The London School of Economics and Political Science

Twin-Constrained Hamiltonian Paths on Threshold Graphs

- an Approach to the Minimum Score Separation Problem

Kai Helge Becker

A thesis submitted to the Department of Management
of the London School of Economics and Political Science
for the degree of Doctor of Philosophy, London, July 2010.

Declaration

I certify that the thesis I have presented for examination for the PhD degree of the London School of Economics and Political Science is solely my own work other than where I have clearly indicated that it is the work of others (in which case the extent of any work carried out jointly by me and any other person is clearly identified in it).

Abstract

The Minimum Score Separation Problem (MSSP) is a combinatorial problem that has been introduced in JORS 55 as an open problem in the paper industry arising in conjunction with the cutting-stock problem. During the process of producing boxes, flat papers are prepared for folding by being scored with knives. The problem is to determine if and how a given production pattern of boxes can be arranged such that a certain minimum distance between the knives can be kept. While it was originally suggested to analyse the MSSP as a specific variant of a Generalized Travelling Salesman Problem, the thesis introduces the concept of twin-constrained Hamiltonian cycles and models the MSSP as the problem of finding a twin-constrained Hamiltonian path on a threshold graph (threshold graphs are a specific type of interval graphs).

For a given undirected graph G(N,E) with an even node set N and edge set E, and a bijective function b on N that assigns to every node i in N a "twin node" b(i)≠i, we define a new graph G'(N,E') by adding the edges {i,b(i)} to E. The graph G is said to have a twin-constrained Hamiltonian path with respect to b if there exists a Hamiltonian path on G' in which every node has its twin node as its predecessor (or successor).

We start with presenting some general findings for the construction of matchings, alternating paths, Hamiltonian paths and alternating cycles on threshold graphs. On this basis it is possible to develop criteria that allow for the construction of twin-constrained Hamiltonian paths on threshold graphs and lead to a heuristic that can quickly solve a large percentage of instances of the MSSP. The insights gained in this way can be generalized and lead to an (exact) polynomial time algorithm for the MSSP. Computational experiments for both the heuristic and the polynomial-time algorithm demonstrate the efficiency of our approach to the MSSP. Finally, possible extensions of the approach are presented.

Acknowledgements

First of all, I am very much indebted to my supervisor Professor Gautam Appa, who has greatly inspired my research, and my life beyond - in fact, to far greater an extent than he is probably aware of. Also, I am very grateful for his permanent support and his understanding of what drives me academically.

Also, I would like to thank the staff members (and later colleagues) at the Operational Research Group at LSE, who all contribute to making the group (and LSE in general) an inclusive and stimulating environment for research that brings to life the spirit that characterises a true university. In particular I am thankful to Dr Barbara Fasolo, Dr Gilberto Montibeller, Dr Alec Morton, Dr Katerina Papadaki, Professor Larry Phillips, Dr Alan Pryor and Professor Paul Williams for interesting insights and both helpful comments and advice on various academic matters in the past years. Moreover, the working environment of the Operational Research Group would have been much less productive and pleasant a place, if the group were not run as excellently on the administrative side as it actually is. I am very grateful to Brenda Mowlam, Jenny Robinson, Richard Szadura and Lucy Underhill for their great support, helpful advice and amazing kindness over all the years, from my first steps at LSE to my time as a member of staff.

I would also like to thank all my fellow PhD students at LSE, especially Dr Nikos Argyris, Florian Gebreiter, Nayat Horozoglu, Attila Marton and Dr Kostas Papalamprou, who all have, in one way or the other, contributed to making LSE my home, on both the academic and the personal level. I am particularly thankful to Dr Nikos Argyris, with whom I undertook the very first steps in analysing the problem that would later become the topic of this thesis; unfortunately, he had to focus his attention on his own thesis after a short time. Also, I am thankful to Dr Kostas Papalamprou for a great introduction into the topic of total unimodularity, and I would like to thank my fellow PhD students for their friendship and great sense of community.

Moreover, I am grateful to Frits Spieksma from the Univeristy of Leuven for his interest in the theoretical aspects of this thesis and several insightful discussions.

Many thanks are also due to British Petroleum who generously funded most of the research undertaken in this thesis.

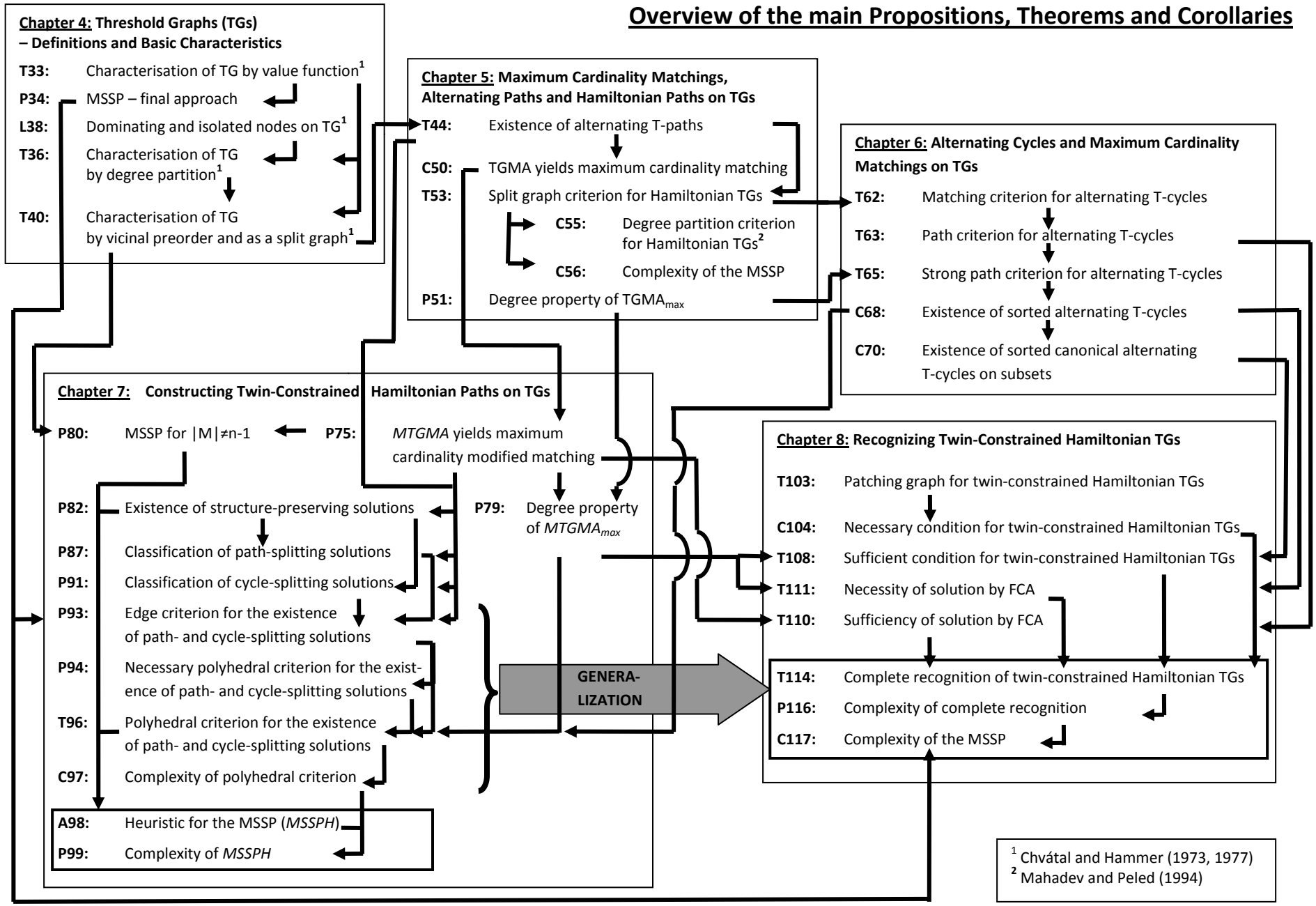
Finally, I would like to thank Professor Gautam Appa, Dr Katerina Papadaki and Edson Franco de Morais for ensuring that I would not lose my thesis out of sight given my commitments as a member of staff at LSE.

Contents

List of Figures

List of Tables

Overview of the main Propositions, Theorems and Corollaries



1 The Minimum Score Separation Problem (MSSP)

The Minimum Score Separation Problem (MSSP) has recently been introduced in the OR literature by Goulimis (2004) in JORS 55 as an open combinatorial problem associated with the cutting stock problem. Goulimis encountered this problem during consultancy projects in the paper and related industries where it arises in the process of producing boxes. Manufacturing boxes involves two steps: first cutting out flat sheets from the raw material and second folding these sheets. The first step of this procedure, which consists of finding a feasible pattern of sheets that minimizes waste, has been well known and investigated for quite some time as the "cutting-stock problem", and is classically solved by delayed column-generation (Gilmore and Gomory 1961, 1963). In contrast to this, the second stage, which, for mechanical reasons, involves an additional constraint, has not received due attention and had not been addressed before the article mentioned.

In particular, as a part of the process of folding, the flat sheets must be prepared by "scoring" them along the fold lines, which is achieved by knives mounted on a bar. Due to technical limitations, the knives cannot be placed at an arbitrary distance to each other, but their distances have to exceed a certain minimum $\alpha \in \mathbb{R}_+^*$ (typically, α could be about 70mm in practice). This implies that a given pattern of flat boxes as a possible outcome of the first stage of the production process is feasible for the second stage only if the boxes can be aligned in a way such that the scores of adjacent boxes are separated by the minimum distance required. The following diagram (Figure 1) illustrates this setting for a possible production pattern that is made up by four (not necessarily different) boxes A, B, C and D:

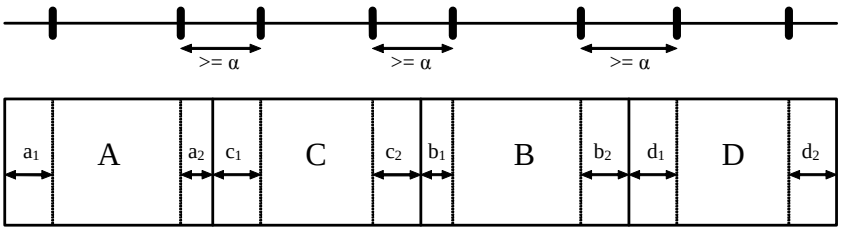


Figure 1: Feasible alignment of boxes I

A possible alignment of boxes in this example consists of arranging the boxes in the order A, C, B and D as shown in Figure 1. Despite the fact that a box may have several scores, only the outer ones matter for the MSSP. Moreover, we can assume that the overall widths of the boxes are large enough that we do not have to take into account the internal distance between the two outer scores of a certain box. Accordingly, the arrangement above is feasible if and only if $a_2 + c_1, c_2 + b_1$, and $b_2 + d_1 \geq \alpha$. (If a box has no score on its left or right side, we will set the corresponding value of the left or right width as α.)

Further combinatorial options arise from the fact that the boxes, when being aligned, can also be rotated by 180^o, as illustrated in Figure 2 where due to rotating box B, the minimum score separation constraint is satisfied if $a_2 + c_1, c_2 + b_2$, and $b_1 + d_1 \geq \alpha$:

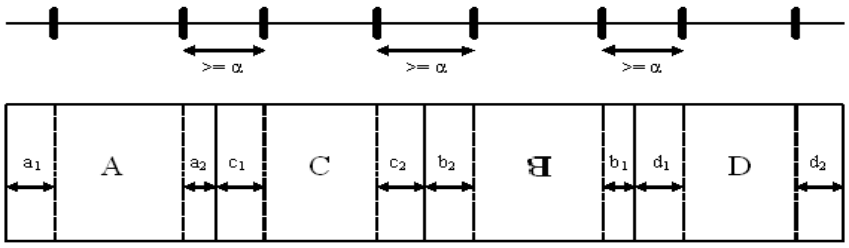


Figure 2: Feasible alignment of boxes II

Given such a setting, the MSSP consists in determining whether or not a certain set of n boxes can be aligned in an order (possibly also by rotating some of the boxes) such that between each pair of boxes the minimum distance requirement is met. As the number of possible arrangements of the boxes including rotations is $O(\frac{n!}{2}2^n)$ - the factor $\frac{1}{2}$ arises due to symmetry - complete enumeration can easily lead to a practically unmanageable combinatorial explosion. (Even in the case of only 10 boxes, a typical value in practice, this would amount to calculating about 1.858×10^9 possible combinations for each candidate production pattern, of which there can be several thousands in the context of generating columns for solving the cutting stock problem.) Though in practical applications, a pattern that turns out to be infeasible in terms of the MSSP is not entirely useless and can still be employed for manufacturing boxes by running the scoring machine at a slower pace, such a situation would cause considerable costs. Therefore, infeasible patterns must be singled out at an early stage before the cutting stock problem is addressed, and be penalised.

In view of this, it must be considered a practically relevant open combinatorial problem to develop an algorithm that, at least as a heuristic for a large percentage of possible production patterns, can quickly determine if a certain arrangement of boxes is (in)feasible in terms of the MSSP, and explicitly generate such an arrangement if this exists at all.

The remainder of this thesis addresses and solves this problem as follows. In chapter 2, we give a precise account of the problem by modelling it in two different ways: first, by describing it on the basis of the concept of a "Travelling Politician Problem" as proposed by Goulimis (2004) in his original description of the MSSP, and second, as an alternative approach, by representing it as a specifically constrained variant of a Hamiltonian Path Problem on a modified version of a so-called "threshold graph". Chapter 3 discusses in detail the relationship of the Minimum Score Separation Problem with the literature on related problems, namely the Hamiltonian Path Problem, the Travelling Salesman Problem, the Clustered Travelling Salesman Problem, and the Generalised Travelling Salesman Problem, and addresses the question of the complexity

of the MSSP. Chapter 4 lays the graph theoretical foundation for our approach to the MSSP. In particular, it introduces the concept of threshold graphs and presents some of their basic characterisations. Proceeding from the perspective provided in the two previous chapters, chapter 5 analyses the relation of paths and maximum cardinality matchings on threshold graphs. In this context, it provides a matching algorithm for threshold graphs that will later form the basis of our algorithm for the MSSP and presents a new polynomial-time algorithm for the Hamiltonian Path Problem on this particular type of graphs. Building on these results, chapter 6 provides several criteria that allow for the construction of alternating cycles on threshold graphs, which will be helpful prerequisites for tackling the MSSP. In chapter 7, we turn our attention to the MSSP itself and analyse the relation of maximum cardinality matchings on threshold graphs and solutions to the MSSP. On this basis, we arrive at criteria that enable us to develop a polynomial-time heuristic for quickly solving a large percentage of instances of the MSSP. Chapter 8 capitalises on the insights gained in the previous chapters and presents an exact polynomial-time algorithm that solves the MSSP. In chapter 9, we demonstrate the efficiency of our approach by giving computational results for a large number of randomly generated instances. The final chapter 10 presents some concluding remarks and an outlook on open questions for further research. Three appendices provide the C++ source codes of the algorithms developed.

2 Two ways of modelling the MSSP

The aim of this section is to provide a precise definition of the Minimum Score Separation Problem by defining it in two different ways: first, by describing it as a variant of the so-called "Travelling Politician Problem" as proposed by Goulimis (2004), and second, alternatively, by representing it as a specifically constrained Hamiltonian Path Problem on what we will later call a "threshold graph". The different notions underlying these two definitions will be illustrated by giving two different MIP representations of the problem. As a point of departure, we will start with a definition of the Hamiltonian Path Problem on an undirected graph and its general MIP representation as a Travelling Salesman Problem (TSP).

2.1 Starting point: the Hamiltonian Path Problem as a TSP

In the following, let $G(N, E)$ be a (finite) *undirected graph* without loops and multi-edges. We will denote its set of *nodes* by $N_G = \{1, 2, ..., n\} \subset \mathbb{N}^*$ and its set of *edges* by $E_G \subseteq N_G \times N_G$. It will be assumed that $E_G \neq \varnothing$ throughout the text.

Definition 1 *(Hamiltonian Path Problem)*

Let $G(N, E)$ be an undirected graph with a node set $N_G = \{1, 2, ..., n\} \subset \mathbb{N}^*$ and a set of edges $E_G \subseteq N_G \times N_G$. Then the Hamiltonian Path Problem consists in finding a path

$$i_1 - i_2 - i_3 - ... - i_{n-1} - i_n$$

with $i_1, i_2, i_3, ..., i_{n-1}, i_n \in N_G$ and $(i_k, i_{k+1}) \in E_G$ for all $k = 1, 2, ..., n-1$, where every node $i_k \in N_G$ occurs in the path once.

By introducing a dummy node $i = 0$, a Hamiltonian Path Problem can routinely be modeled as a TSP. The resulting TSP has constant cost coefficients $c := (0, 0, ..., 0)$ and is defined on the extended graph $G'(N, E)$, with the new node set being given by $N_{G'} := N_G \cup \{0\}$ and the new edge set by $E_{G'} := E_G \cup \{0\} \times N_G \cup N_G \times \{0\}$.

One among several possible MIP representations of the TSP consists in the single commodity flow formulation by Gavish and Graves (1974), which we will employ in the following for the illustrative purpose of this section because it allows for a very transparent description of the subtour elimination constraints. (See Orman and Williams (2004) for a survey of different MIP formulations of the TSP.) Given for all nodes $i, j \in N_{G'}, i \neq j$, the binary variables x_{ij} with $x_{ij} = 1$ iff the edge $(i, j) = (j, i)$ is part of the TSP tour, the continuous variables y_{ij} to denote possible flows on the edges, and binary constants $\delta_{ij} = 1 :\Leftrightarrow (i, j) \in E_{G'}$ representing the edges of the graph, the single commodity flow formulation of the TSP for the general Hamiltonian Path Problem reads as follows:

$$\text{minimize } 0 \tag{1}$$

$$\text{subject to } \sum_{j,j\neq i} x_{ij} = 1 \text{ for all } i \in N_{G'} \tag{2}$$

$$\sum_{i,i\neq j} x_{ij} = 1 \text{ for all } j \in N_{G'} \tag{3}$$

$$y_{ij} \leq n x_{ij} \text{ for all } j \in N_{G'}, i \neq j \tag{4}$$

$$\sum_{j>0} y_{0j} = n \tag{5}$$

$$\sum_{i,i\neq j} y_{ij} - \sum_{k,k\neq j} y_{jk} = 1 \text{ for all } j \in N_{G'} - \{0\} \tag{6}$$

$$x_{ij} \leq \delta_{ij} \text{ for all } i,j \in N_{G'}, i \neq j \tag{7}$$

$$x_{ij} \in \{0;1\}, y_{ij} \geq 0 \text{ for all } i,j \in N_{G'}, i \neq j \tag{8}$$

In this formulation, constraints (2) and (3) represent the assignment relaxation of the TSP. The "flow" y_{ij} imposed by constraints (4), (5) and (6) ensures the elimination of subtours. This is achieved by requiring the tour to start at the dummy node with an initial flow of n units on the first edge, from which one unit is consecutively "dropped" at each node along the tour until the flow finally becomes zero. (In the case of several subtours instead of one "full" tour, the subtours without the dummy node would have no initial flow according to (5) so that consecutively dropping a flow at each node along the subtour would lead to a violation of (6) at the node where the subtour is completed.) Constraints (7) finally impose the structure of the graph on the model.

2.2 First approach: the MSSP as a Travelling Politician Problem

Given this context of the Hamiltonian Path Problem and the TSP, Goulimis (2004), in his problem presentation, suggested to approach the MSSP as a certain type of generalized TSP, namely as what he calls a "Travelling Politician Problem" (TPP). A TPP is a path in a graph consisting of *pairs* of nodes, where the path must pass each node pair exactly once, which can be imagined as a travelling politician who - during her election campaign - has to visit exactly one out of two available cities in each constituency (or state), and return finally to where she started. (Note that Goulimis does not include the idea of returning to the point of departure in his description of the TPP, however consistency demands so in order to maintain a conceptual parallel to the TSP.)

Figure 3 illustrates this problem for $n = 5$ constituencies, with the single node in the ellipse being a dummy node that models the way back to the point of departure. Of course, the general TPP could also be depicted, analogously to the general TSP, without a dummy node, but with a direct way back to the starting point instead. However, as our description here is ultimately intended as a means to represent the MSSP, a dummy node has been included in the diagram.

In terms of the MSSP, the different constituencies represent $n = 5$ different boxes that have to be arranged in a certain order, namely as a path that covers all boxes (all constituencies). The two cities in each constituency denote the two possible ways of including a box ("regular", and after a 180^o rotation) in the alignment. A feasible arrangement of boxes then consists of a path that passes through all constituencies exactly once. (The dummy node in the ellipse has no representational value in the MSSP as such and just ensures that the path is extended to a tour such that we can illustrate this setting down below by building upon the MIP representation of a TSP.)

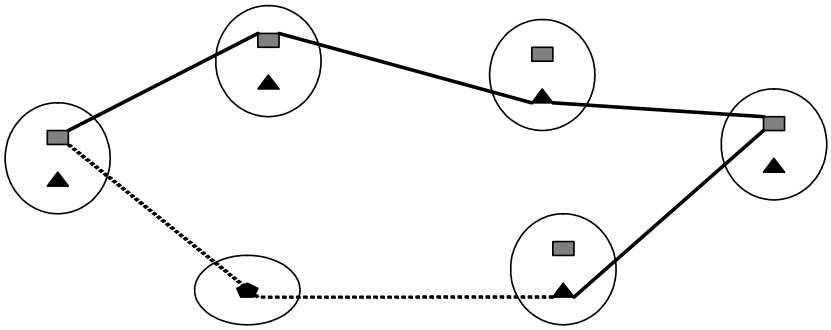


Figure 3: The MSSP as a Traveling Politician Problem

Let us now formalise Goulimis' approach. We can denote the n boxes by two nodes each (for both ways of placing the box into the alignment) such that we obtain a node set $N_G = \{1, 2, 3, 4, ..., 2n - 1, 2n\}$ where the odd numbers represent the "regular" way of placing the boxes, and the even ones the boxes after rotation. Further, let $v_1(i)$ and $v_2(i)$ be the widths of the ("regular") left and the right hand sides of the "regularly" placed boxes, respectively, i.e. for all odd nodes $i = 1, 3, ..., 2n-1$. Conversely, let $v_1(i)$ and $v_2(i)$ be the right and the left hand sides of the rotated boxes, respectively, i.e. for all even nodes $i = 2, 4, ..., 2n$. Consequently, we have $v_1(2k - 1) = v_2(2k)$ and $v_2(2k - 1) = v_1(2k)$ for all $k = 1, 2, ..., n$. Then, given a minimum knife distance $\alpha \in \mathbb{R}^*_+$, the minimum score separation constraint is satisfied for two nodes $i, j \in N_G$ that *do not belong to the same box* iff $v_1(i) + v_2(j) \geq \alpha$ or $v_2(i) + v_1(j) \geq \alpha$.

This yields the following formal definition of the MSSP on a directed graph:

Definition 2 *(Minimum Score Separation Problem - Approach 1)*

Let $G(N, A)$ be a directed graph with the even node set $N_G = \{1, 2, ..., 2n-1, 2n\}$, $\alpha \in \mathbb{R}_+^*$ a positive number (a "minimal value") and $v_p : N_G \to \mathbb{R}_+^*$ with $p \in \{1, 2\}$ a symmetric pair of "value functions" that assigns two positive numbers $v_1(i)$ and $v_2(i)$ to every node $i \in N_G$ such that the symmetry condition

$v_1(2k-1) = v_2(2k)$ *and* $v_2(2k-1) = v_1(2k)$ *for all* $k = 1, 2, ..., n$

holds. Moreover, let the edge set of the graph be defined by the adjacency condition

$A_G := \{(i, j)$ *with* $v_2(i) + v_1(j) \geq \alpha \mid$

$i = 2k \wedge j \neq 2k-1$ *or* $i = 2k-1 \wedge j \neq 2k$ *for some* $k = 1, 2, ..., n\}$.

Then the MSSP consists in deciding whether there exists on G a subpath that contains exactly one node out of each of the node subsets $\{2k-1, k\}$ for all $k = 1, 2, ..., n$.

To state this definition differently: The perspective involved in this approach implies (due to the symmetry between the odd and the even nodes) that the MSSP is feasible if the graph just defined can be partitioned into two subsets S and $\bar{S}$ of nodes of equal cardinality such that (a) for each $k = 1, 2, ..., n$ either $(2k-1) \in S$ and $(2k) \in \overline{S}$ or $(2k) \in S$ and $(2k-1) \in \overline{S}$ and that (b) there exists a Hamiltonian path in one subset (and, due to symmetry, consequently also in the other subset). Conversely, if such a partition does not exist, the MSSP is infeasible. In the diagram above, when leaving aside the dummy node and the dotted edges, this partition is represented by the set of the connected nodes on one side and the set of the unconnected nodes on the other side. Because of the symmetry between "regular" boxes and their rotated counterparts, the path drawn in the diagram ensures that there exists also a path covering the unconnected nodes, and this in the same order of constituencies.

We can illustrate this definition of the MSSP by a MIP formulation gained from the TSP representation based on the Hamiltonian path property of the subsets. This means the following TSP is feasible if and only if there exists a subtour in the graph that covers all and only all of the nodes of a subset S of which for each $k = 1, 2, ..., n$ either the node $(2k)$ or the node $(2k-1)$ is an element. Again, corresponding to the general TSP model for Hamiltonian paths above, we have to introduce a dummy node 0 (the one already depicted in figure 2 above) and define the extended graph $G'(N, A)$, with the new node set given by $N_{G'} := N_G \cup \{0\}$ and the new edge set by $A_{G'} := A_G \cup \{0\} \times N_G \cup N_G \times \{0\}$. Also analogously, introducing for all nodes $i, j \in N_{G'}, i \neq j$, the binary variables x_{ij}, the continuous flow variables y_{ij}, and the binary constants $\delta_{ij} = 1 :\Leftrightarrow (i, j) \in E_{G'}$ yields the following MIP approach to the MSSP.

$$\text{minimize } 0 \tag{1}$$

$$\text{subject to} \sum_{j,j\neq 2i-1,j\neq 2i} (x_{2i-1,j} + x_{2i,j}) = 1 \text{ for all } i \in N_{G'} - \{0\} \tag{2a'}$$

$$\sum_{j>0} x_{j0} = 1 \tag{2b'}$$

$$\sum_{i,i\neq 2j-1,i\neq 2j} (x_{i,2j-1} + x_{i,2j}) = 1 \text{ for all } j \in N_{G'} - \{0\} \tag{3a'}$$

$$\sum_{j>0} x_{0j} = 1 \tag{3b'}$$

$$\sum_{j,j\neq i} x_{ij} - \sum_{j,j\neq i} x_{ji} = 0 \text{ for all } i \in N_{G'} - \{0\} \tag{9}$$

$$y_{ij} \leq n x_{ij} \text{ for all } i,j \in N_{G'}, i \neq j \tag{4'}$$

$$\sum_{j>0} y_{0j} = n \tag{5'}$$

$$\sum_{i,i\neq j} y_{ij} - \sum_{k,k\neq j} y_{jk} = \sum_{i,i\neq j} x_{ij} \text{ for all } j \in N_{G'} - \{0\} \tag{6'}$$

$$x_{ij} \leq \delta_{ij} \text{ for all } i,j \in N_{G'}, i \neq j \tag{7}$$

$$x_{ij} \in \{0;1\}, y_{ij} \geq 0 \text{ for all } i,j \in N_{G'}, i \neq j \tag{8}$$

In this formulation, constraints $(2a'), (2b'), (3a')$ and $(3b')$ are the equivalents of the assignment relaxation constraints of the TSP above. Constraints $(2a')$ and $(3a')$ refer to all nodes representing a box. Corresponding to the set $S \subseteq N_{G'}$ that, for each $k = 1, 2, ..., n$, contains either the node $(2k)$ or the node $(2k-1)$ as an element, only half of the nodes (namely one for each box) must be assigned to one successor and one predecessor. Constraints $(2b')$ and $(3b')$ make sure that the dummy node definitely is included in the tour with one successor and one predecessor. However, in contrast to the general MIP formulation of the Hamiltonian Path Problem, these assignment relaxation constraints must be complemented by constraints (9). Without (9), constraints $(2a')$ and $(3a')$ would allow some nodes to have either only predecessors or only successors. This is avoided by forcing both sums in (9) to either the value 1 or the value 0 for all $i \in N_{G'} - \{0\}$. Doing so ensures that all nodes representing boxes have either (a) both a predecessor and a successor, or (b) neither a predecessor nor a successor, i.e. the nodes are either fully included in or entirely excluded from the tour.

The flow constraints for the elimination of inappropriate subtours $(4')$, $(5')$ and $(6')$ have been only slightly modified compared to the TSP above. Note however that, despite $|N_{G'}| = 2n+1$ here, constraints $(4')$ and $(5')$ still have the constant n on their right-hand sides because we are only interested in a path that covers half of the nodes (and the dummy). The change

regarding the constraints (6′) takes into account that a node may be used for the subtour or not at all. If and only if a node is part of the tour to be found, the right-hand side of (6′) equals 1 and a unit of the flow is "dropped" at the node. Otherwise, the right hand side equals 0 and the corresponding node j is neutral with respect to the flow that enforces the elimination of subtours. Finally, constraints (7) again impose the structure of the graph on the model.

2.3 Second approach: the MSSP as a Twin-Constrained Hamiltonian Path Problem

We will now introduce an alternative approach to the MSSP, which is more intuitive in so far that (as the reader will also observe in terms of notational effort) it goes down more "naturally" on the basis of the concept of Hamiltonian Paths. Still, we model each box by two nodes and look for a certain path connecting the nodes. However, while in the first approach the two nodes for each box represent the two possible positions of a box in the alignment ("regular", or rotated), here the two nodes correspond to the left and the right sides of the boxes. This means that if a "regular" box is described by a pair $(i, j) \in N_G \times N_G$, its rotated counterpart is denoted as (j, i). We will call the two nodes that make up a box "twin nodes" in the following.

Analogously to the preceding subsection, we could imagine the left hand sides of the boxes being represented by the odd node numbers in the node set, and the right hand sides by the even numbers, but we will not require this in the following because the algorithm for the MSSP to be developed later will permute the order of nodes anyway. This is why we will remain more flexible and introduce a "twin node function" as a bijective function $b : N_G \to N_G$ from the set of nodes N_G onto itself that associates each node $i \in N_G$ with its twin node $t := b(i) \in N_G$ (consequently we have $b(b(i)) = b(t) = i$). Then, if a node i represents one side of a certain box, the node $b(i)$ represents the other side of this box, and it does not matter, if i is the right or the left side of the box, or if the box is included in the alignment in a "regular" or a rotated fashion. Note that the notion of "twin nodes" here does not imply that two twin nodes are adjacent to the same set of other nodes because the two sides of a box could differ with respect to the other boxes they can be placed next to. In this model, "twins" are closely attached to each other, but not perceived to be necessarily identical, so to speak.

Similarly to the previous subsection, we will denote the minimum knife distance by a certain positive $\alpha \in \mathbb{R}_+^*$ and introduce a "value function" v to model the widths of the left and right sides of the boxes. However, since we model each side of a box separately from the other side in this alternative approach, one value function $v : N_G \to \mathbb{R}_+^*$ suffices here, and we can do without a "symmetry condition" of the kind imposed on the pair of value functions in the previous subsection. On this simpler basis, two nodes $i, j \in N_G$ fulfil the minimum score separation constraint, i.e. are adjacent in the graph G, if and only if $v(i) + v(j) \geq \alpha$ − provided that they do not belong to the same box, i.e. $j \neq b(i)$.

As both sides of a box must be part of the final arrangement of boxes (we cannot cut the

boxes into halves), both nodes modeling a box must finally be part of the path that represents the alignment of boxes. In other words: this way of describing the MSSP, in contrast to Goulimis' approach, does not require a certain subpath in a partition of nodes for a feasible solution, but instead a (complete) Hamiltonian Path, which (by definition) covers all nodes of the graph. In this perspective, the MSSP finally turns out to be a "normal" Hamiltonian Path Problem on a graph given by the specific adjacency condition $v(i) + v(j) \geq \alpha$ for all nodes $j \neq b(i)$, with the only additional requirement being that each node has its twin node either as its successor or its predecessor ("twin node condition").

This gives rise to the following definition.

Definition 3 *(Twin-Constrained Hamiltonian Path Problem)*

Let $G(N, E)$ be an undirected graph with node set $N_G = \{1, 2, ..., n\} \subset \mathbb{N}^*$, a set of edges $E_G \subseteq N_G \times N_G$, and $b : N_G \rightarrow N_G$ a bijective function that associates every node $i \in N_G$ with a "twin node" $t := b(i) \in N_G$, $t \neq i$. Moreover, let G' be a graph derived from G by

$$N_{G'} := N_G \text{ and } E_{G'} := E_G \cup \{(i, j) : i = b(j)\}.$$

Then the Twin-Constrained Hamiltonian Path Problem on G with respect to b consists in deciding whether there exists on G' a Hamiltonian path of the form

$$i_1 - b(i_1) - i_2 - b(i_2) - ... - i_n - b(i_n),$$

i.e. a Hamiltonian path in which every node is either predecessor or successor of its twin node ("twin node condition"). G is called the underlying graph and b the twin-node function of the Twin-Constrained Hamiltonian Path Problem.

Note that, due to the bijectivity of the twin function b, the twin node condition actually describes *all* possible types of paths in which every node is either predecessor or successor of its twin node. Obviously, equivalent formulations of this condition would be, for example, also

$$b(i_1) - i_1 - b(i_2) - i_2 - ... - b(i_n) - i_n, \text{ and}$$
$$i_1 - b(i_1) - b(i_2) - i_2 - i_3 - b(i_3) - ... - b(i_n) - i_n.$$

The setting of a twin-constrained Hamiltonian path is illustrated for an MSSP with $n = 5$ boxes in Figure 4, in which the two nodes that share a circle (the triangle and the rectangle) correspond to the same box (imagine the rectangles as their right sides and the triangles as their left ones, "normal" position given). The inseparability of the twin nodes in the model is reflected by the bold lines (edges) connecting triangles and rectangles such that any Hamiltonian Path must either enter a circle at the side of the triangle and proceed to the rectangle, or vice versa. Again, the ellipse represents a dummy node to turn the Hamiltonian Path Problem into a TSP.

The mathematical intuition guiding this "twin node" approach can be described as follows: Goulimis' model, as presented in the previous section, requires doubling all nodes in order to account for the fact that boxes can be rotated. Parallel to this, his approach requires a pair of value functions that, due to their symmetry property, also double the number of mathematical

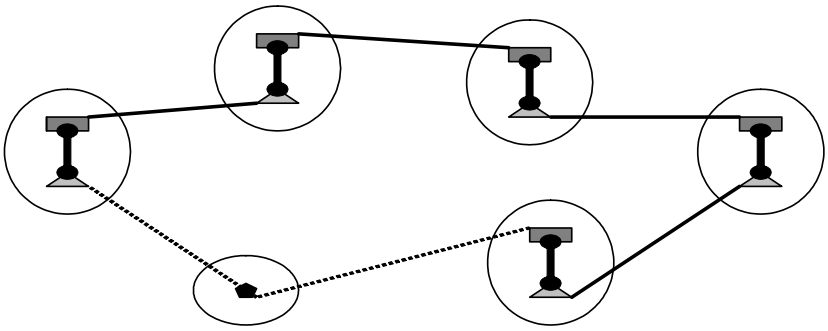


Figure 4: The MSSP as a twin-constrained Hamiltonian path problem

entities for representing rotated boxes. Despite this effort of doubling the mathematical structure, any solution to the MSSP (if there exists one at all for a certain instance) will use only half of the nodes in the model. Such a structural redundancy is theoretically unsatisfying and suggests looking for a more straight forward approach (also in view of the classical methodological principle of "Occam's razor"). This is why our alternative approach models each side of a box separately from the corresponding other side and additionally imposes the "twin node condition". In proceeding in this way, we can describe every solution to the MSSP readily as a Hamiltonian Path in terms of all boxes. Moreover, apart from a certain elegance involved in this description, this perspective of looking at the MSSP opens up a flexible way to exploit the adjacency conditions of the graph for each node separately because we do not have to care about both sides of a box at the same time - at least not in the first instance. In particular, modeling the sides of the boxes separately leads to a "threshold graph", the properties of which can fruitfully be exploited for finding a solution to the MSSP. Finally, the algorithm gained in this way shows a surprisingly effective behaviour, which ultimately justifies this perspective on the MSSP.

Having presented the ideas underlying our alternative approach, we summarize the second model of the MSSP in the following formal definition:

Definition 4 *(Minimum Score Separation Problem - Approach 2)*

Let $G(N, E)$ be an undirected graph with an even set of nodes $N_G = \{1, 2, ..., 2n-1, 2n\} \subset \mathbb{N}^*$, and $b : N_G \to N_G$ a twin node function. Moreover, let $\alpha \in \mathbb{R}_+^*$ a positive constant (a "minimal value") and $v : N_G \to \mathbb{R}_+^*$ be a "value function" that assigns a positive number to each node. The edge set of the graph $E_G \subseteq N_G \times N_G$ be defined by the adjacency condition

$$(i, j) \in E_G :\Leftrightarrow v(i) + v(j) \geq \alpha \text{ for all } i, j \in N_G, j \neq i.$$

Then the Minimum Score Separation Problem (MSSP) consists in deciding if there exists a twin-constrained Hamiltonian path on G with respect to b.

Finally, we will illustrate also this definition of the MSSP by a MIP formulation gained from the TSP presentation of the Hamiltonian Path property of a feasible solution to the MSSP. The reader will observe that this approach comes much closer to modeling the general Hamiltonian Path Problem than Goulimis' approach does. Again, corresponding to the general TSP model for Hamiltonian Paths above, we have to introduce a dummy node 0 (the one already depicted in figure 3), and define the extended graph $G'(N, E)$, with the new node set given by $N_{G'} := N_G \cup \{0\}$ and the new edge set by $E_{G'} := E_G \cup \{0\} \times N_G \cup N_G \times \{0\}$. Also analogously, introducing for all nodes $i, j \in N_{G'}, i \neq j$, the binary variables x_{ij}, the continuous flow variables y_{ij}, and the binary constants $\delta_{ij} = 1 :\Leftrightarrow (i, j) \in E_{G'}$ yields the following MIP approach to the MSSP.

$$\text{minimize } 0 \tag{1}$$

$$\text{subject to } \sum_{j,j\neq i} x_{ij} = 1 \text{ for all } i \in N_{G'} \tag{2}$$

$$\sum_{i,i\neq j} x_{ij} = 1 \text{ for all } j \in N_{G'} \tag{3}$$

$$x_{i,b(i)} + x_{b(i),i} = 1 \text{ for all } i \in N_{G'} - \{0\} \tag{10}$$

$$y_{ij} \leq 2n x_{ij} \text{ for all } i, j \in N_{G'}, i \neq j \tag{4"}$$

$$\sum_{j>0} y_{0j} = 2n \tag{5"}$$

$$\sum_{i,i\neq j} y_{ij} - \sum_{k,j\neq k} y_{jk} = 1 \text{ for all } j \in N_{G'} - \{0\} \tag{6}$$

$$x_{ij} \leq \delta_{ij} \text{ for all } i, j \in N_{G'}, i \neq j \tag{7}$$

$$x_{ij} \in \{0; 1\}, y_{ij} \geq 0 \text{ for all } i, j \in N_{G'}, i \neq j \tag{8}$$

In contrast to the first approach formalised above on the basis of Goulimis' suggestions, our formulation requires only minor changes of the original MIP for the Hamiltonian Path Problem. The assignment relaxation constraints (2), (3) and the subtour elimination constraint (6) remain the same as in the original problem. The only significant difference here is the introduction of the additional "twin node constraints" (10). Complementing the assignment relaxation constraints, they ensure that each node has its twin node either as its predecessor or its successor. The subtour elimination constraints $(4'')$ and $(5'')$ can be kept almost unchanged, with the only change to the general Hamiltonian Path Problem being that they have to account for the fact that the node set of the graph here has cardinality $|N_{G'}| = 2n + 1$. Constraints (7) impose the structure of the graph on the model again.

3 The MSSP, Hamiltonian paths, variants of the TSP and complexity theory

We have seen in the previous chapter that our approach of modelling the Minimum Score Separation Problem leads to a specific type of Hamiltonian path problem and a specific variant of the Travelling Salesman Problem. This chapter explores in more detail of how the Mininum Score Separation Problem is related to existing research about these problems, namely to research about the Hamiltonian Path Problem, the Travelling Salesman Problem, the Clustered Travelling Salesman Problem, and the Generalised Travelling Salesman Problem. Moreover, it addresses the question of the complexity of the MSSP. In doing so, we will proceed from the most specific case, which is the Hamiltonian path problem, to the most general case, namely the Generalised Travelling Salesman Problem. We begin with some preliminary remarks on the notation used in this thesis.

3.1 Notation

In the remainder of this paper, we will mostly follow Schrijver's notation (Schrijver 2003). Whenever specific aspects of threshold graphs are concerned, we apply Mahadev's and Peled's notation, who have written the major source on this topic (Mahadev and Peled 1995).

Let $G\,(N, E)$ be a (finite and simple) *undirected graph* with *node set* $N_G = \{1, 2, ..., n\} \subset \mathbb{N}^*$ and *edge set* $E_G \subseteq N_G \times N_G$ It will be assumed that $E_G \neq \varnothing$ throughout the text. As, in most cases, we prefer to denote edges by pairs of nodes (instead of using the set notation $\{i, j\}$, which can often be found in the case of undirected graphs), we will always take for granted that $(i, j) \in E_G \Leftrightarrow (j, i) \in E_G$ when using this notation. A graph $G(N, E)$ is called *complete* iff we have $E_G = N_G \times N_G$. Given two graphs $G(N, E)$ and $G'(N, E)$ with

$$N_{G'} \subseteq N_G \text{ and } E_{G'} \subseteq E_G,$$

the graph G' is called a *subgraph of* G and G is said to *contain* G'. For

$$N_{G'} := P \subsetneqq N_G \text{ and } E_{G'} := E_G \cap P \times P,$$

the graph $G^{'}(P, E)$ is called the *subgraph of* G *induced by* P.

A *path* in an undirected graph $G(N, E)$ is a sequence

$$(i_0, e_1, i_1, ..., e_n, i_n)$$

where $i_0, i_1, ..., i_n \in N_G$ are distinct nodes and $e_1, e_2, ..., e_n \in E_G$ edges such that e_k is an edge that connects the nodes i_{k-1} and i_k. We will call i_0 and i_n the *end nodes* of the path and say that the path connects the nodes $i_0, i_1, ..., i_n \in N_G$. A sequence

$$(i_0, e_1, i_1, ...e_{n-1}, i_{n-1}, e_n, i_n)$$

is called a *cycle* if $i_0, i_1, ..., i_{n-1} \in N_G$ are distinct nodes, $i_0 = i_n$, and $e_1, e_2, ..., e_n \in E_G$ edges such that e_k is an edge that connects the nodes i_{k-1} and i_k If the context prevents misunderstanding, we refer to a path or a cycle by just providing the corresponding set of edges

$$\{e_1, e_2, ..., e_n\} \subseteq E$$

or the corresponding order of nodes

$$i_0 - i_1 - ... - i_{n-1} - i_n.$$

A graph G is said to be *connected* if for any two of nodes $i_0, i_n \in N_G$ there exists a path the end nodes of which are i_0 and i_n, and it is called *Hamiltonian* if there exists a cycle that connects all nodes in N_G. Such a cycle is called a *Hamiltonian cycle*. By removing one edge from a Hamiltonian cycle, we obtain a *Hamiltonian path*.

For any node $i \in N_G$, the subset of nodes $N(v) \subseteq N_G$ is called a *neighbourhood* of the node i if $N(i)$ contains all nodes that are adjacent to i, i.e.

$$N(i) := \{j \in N_G : (i, j) \in E_G\}.$$

The *closed neighbourhood* $N[i] \subseteq N_G$ of a node $i \in N_G$ is the union of i and the neighbourhood of i, i. e.

$$N[i] := N(i) \cup \{i\}.$$

A node is called *isolated* if its neighbourhood is empty, and *dominating* if its closed neighbourhood is the entire set of nodes.

A subset of nodes $S \subseteq N_G$ is a *stable set* when $i \notin N(j)$ for all nodes $i, j \in S$, and a subset $K \subseteq N_G$ is called a *clique* when $i \in N(j)$ for all nodes $i, j \in K$.

A *preorder* is the pair $(P, \succsim)$ of a set P and a binary relation $\succsim$ on P when the two following properties hold:

$$(i)\ p \succsim p \text{ for all } p \in P\ , \qquad \text{(reflexivity)}$$

$$(ii)\ (p_1 \succsim p_2) \wedge (p_2 \succsim p_3) \Rightarrow (p_1 \succsim p_3)$$
$$\text{for all } p_1, p_2, p_3 \in P\ . \qquad \text{(transitivity)}$$

We will say the preorder is *total* iff all nodes can be compared with respect to the preorder, i. e.

$$p_1 \succsim p_2 \text{ or } p_2 \succsim p_1 \text{ for all } p_1, p_2 \in P.$$

Defining a binary relation $\succsim$ on the set of nodes N_G by

$$i \succsim j :\Leftrightarrow N[i] \supseteq N(j) \text{ for all } i, j \in N_G,$$

clearly yields a preorder, which will be called the *vicinal preorder of* G (and is not total in the general case). If $i \succsim j$, we will call the node i *greater* than j, and j *smaller* than i. (Note that it is possible to have both $i \succsim j$ and $j \succsim i$, which is equivalent to $N(i)\backslash\{j\} = N(j)\backslash\{i\}$.)

The cardinality of the neighbourhood of a node i is called its *degree*, denoted by

$$dg(i) := |N(i)|.$$

Given the family $\delta_1 < \delta_2 < ... < \delta_{m-1} < \delta_m$ of distinct positive degrees of a graph G, and $\delta_0 := 0$ (even if there are no isolated nodes in the graph), we define the family of sets

$$D_k := \{i \in N_G : dg(i) = \delta_k\} \text{ for all } k = 0, 1, ..., m$$

and call

$$N_G = D_0 + D_1 + D_2 + ... + D_m$$

the *degree partition of* G.

For a graph $G(N, E)$ the relation $M \subseteq E_G \subseteq N_G \times N_G$ is called a *matching* iff the relation is (i) *symmetric*, i.e. $(i, j) \in M$ implies $(j, i) \in M$ for all $i, j \in N_G$, (ii) *functional*, i.e. $(i, j) \in M$ implies $(i, k) \notin M$ for all $i, j, k \in N_G$, $k \neq j$, and (iii) $(i, i) \notin M$ for all $i \in N_G$. A matching M^* is said to be a *maximum cardinality matching* if $|M| \leq |M^*|$ for all matchings $M \subseteq E$. If a matching is a *left-total* relation, i.e. for all $i \in N_G$ there exists a $j \in N_G$ such that $(i, j) \in M$, the matching is called *perfect*.

3.2 Hamiltonian paths and alternating Hamiltonian paths

In this section we will address the relationship of the MSSP to research on the Hamiltonian Path Problem and some of its generalisations. As mentioned in the first chapter, for a given undirected graph $G(N, E)$ a *Hamiltonian path* is a path

$i_1 - i_2 - i_3 - ... - i_{n-1} - i_n$
with $i_1, i_2, i_3, ..., i_{n-1}, i_n \in N_G$ and
$(i_k, i_{k+1}) \in E_G$ for all $k = 1, 2, ..., n-1$,
such that every node $i_k \in N_G$ occurs in the path exactly once.

We will speak of a *Hamiltonian cycle* iff the first and the last node of the path are identical, and a graph with at least one Hamiltonian cycle is said to be *Hamiltonian*.

The problem of finding a Hamiltonian path or cycle, while being named after Hamilton (1858), was already described in an earlier paper by Kirkman (1856), who discussed a planar graph that is not Hamiltonian (see Biggs, Lloyd and Wilson (1976) for details on the history of this problem). Since these early publications, more than one thousand papers have been published, providing theoretical insights, algorithms or applications of the problem. Among the theoretical insights there are criteria for Hamiltonicity and non-Hamiltonicity based on certain characteristics of a graph (such as connectedness, toughness or the number of edges), for example, stochastic analyses on the frequency of Hamiltonian graphs, theorems on the Hamiltonicity of graphs that do not contain specific subgraphs, and results on the number of different Hamiltonian cycles that might exists for a particular graph. An overview of the vast amount of literature on these and related topics can be found in Bermond (1978), Gould (1991) and Gould (2003).

Regarding the computational question of how to find a Hamiltonian cycle for a given instance of a graph, various types of general-purpose algorithms have been developed for random graphs (see Posa (1976), Angluin and Valiant (1979), Bollobas, Fenner and Frieze (1987), Frieze (1988), Brunacci (1988), Kocay (1992), Broder, Frieze and Shamir (1994), Shufelt and Berliner (1994), Vandegriend and Culberson (1998), and Wagner and Bruckstein (1999), for example). Some of these algorithms are heuristics for finding a Hamiltonian path or cycle (such as Wagner and

Bruckstein (1999), for example), while others are exact algorithms that give a definite answer to the question of whether a given graph is Hamiltonian (such as Shufelt and Berliner (1994), for example). While the latter come with the disadvantage of a long computational time in the worst case, the former might not find a definite answer on the Hamiltonicity of an input instance (cf. Shields, 2004, for more details on some of the algorithms).

Additionally, research has led to algorithms for specific classes of graphs that can decide on the Hamiltonicity of a graph in a comparably short amount of computational time ("polynomial time", to be precise, see section 4 of this chapter). Among the specific classes of graphs for which such an efficient algorithm exists there are, for example, proper interval graphs (Bertossi, 1983), interval graphs (Keil, 1985), circular-arc graphs (Shih, Chern and Hsu, 1992), threshold graphs (Mahadev and Peled, 1994, see also chapters 4 and 5 of this thesis for more details), graphs without "claws" and "nets" as subgraph (Brandstaedt, Dragan and Koehler, 2000), distance-hereditary graphs (Hung and Chang, 2005), strongly chordal graphs that do not contain the subgraphs

$G_1(N, E_1)$ and $G_2(N, E_2)$

with the (common) node set $N := \{a, b, c, d, e\}$ and the edge sets

$E_1 := \{(a, b), (a, c), (b, c), (c, d), (c, e)\}$, $E_2 := E_1 + \{(d, e)\}$

and have an order of at least 5 (Abueida and Sritharan, 2006), and quasi-adjoint graphs (Blazewicz, Kasprzaka, Leroy-Beaulieuc and de Werra, 2008). Note that the references given for efficient algorithms here, refer to the first published paper to tackle algorithmically the question of the Hamiltonicity of a particular class of graphs. For many of these graph classes, later research led to the development of improved algorithms that solve the Hamiltonian cycle (or path) problem with less computational effort.

Apart from the result on recognizing Hamiltonian threshold graphs, to which we will come back in chapter 5 of this thesis, going more into the details of these streams of research on the Hamiltonicity of graphs is not necessary for discussing our problem, the MSSP. As mentioned in the previous chapter, the MSSP is a Hamiltonian path problem with one additional type of constraint, namely the constraints that each node must have its twin-node as a successor (or predecessor) in the Hamiltonian path. If we would like to arrive at an algorithm that exploits the specific structure of the MSSP, we can not expect the results of the research on the (ordinary) Hamiltonian path or cycle problem to be particularly helpful for us. For this reason, let us turn to a variant of the problem of recognizing Hamiltonian graphs that has a more specific structure: the problem of finding *alternating* (Hamiltonian) paths and cycles.

Definition 5 *(Alternating Hamiltonian cycles and paths on 2-edge-coloured graphs)*

For a given graph $G(N, E)$, we colour some edges "red" and some edges "blue". The graph G is said to have an alternating Hamiltonian cycle (path) if there exists a Hamiltonian cycle (path) on G such that successive edges differ in colour. G is said to be alternating Hamiltonian iff there exists an alternating Hamiltonian cycle on G.

Remark 6 *We note that the twin-constrained Hamiltonian path problem, and hence the MSSP, is obviously a special case of the problem of finding an alternating Hamiltonian path, as we can imagine the edges of our underlying graph G as "blue" edges and the edges $(i, b(i))$ that connect a pair of twin nodes as "red" edges.*

The following proposition attributed to Häggkvist (1979) states that, in a certain sense, the problem of finding an alternating Hamiltonian cycle (path) on a 2-edge-coloured graph generalizes the problem of finding an ordinary Hamiltonian cycle (path).

Proposition 7 *(Reducibility of Hamiltonicity to alternating Hamiltonicity)*
An algorithm that can decide on the alternating Hamiltonicity of graphs is able to decide on the Hamiltonicity of graphs.

Proof. For a given (uncoloured) graph $G(N, E)$ we define a new graph G' by (i) introducing for each edge $(i, j) \in E$ two new nodes k and l, (ii) replacing (i, j) by the four edges (i, k), (k, j), (i, l) and (l, j), and (iii) colouring the edges (i, k) and (l, j) "red", while colouring the edges $(k.j)$ and (i, l) "blue". If we can find an alternating Hamiltonian path (cycle) on G', we just have to replace the four edges (i, k), (k, j), (i, l) and (l, j) by the original edge (i, j) and remove the nodes k and l in order to construct a Hamiltonian path (cycle) on G. Conversely, any Hamiltonian path (cycle) on G obviously corresponds to an alternating Hamiltonian path (cycle) on G'. ■

The concept of alternating paths goes back, as do an astonishing number of graph theoretical problems, to Petersen (1891). (See also Mulder (1992) for a discussion of Petersen's results in the light of contemporary graph theory.) The problem of alternating Hamiltonicity is likely to have first been introduced by Bankfalvi and Bankfalvi (1968), going back to a problem stated by Erdös (cf. Bang-Jensen and Gutin, 1997). In their paper, Bankfalvi and Bankfalvi gave a criterion according to which the Hamiltonicity of a 2-edge coloured complete(!) graph with an even node set depends on the sum of the degrees of the nodes in certain pairs of disjunct subsets of the node set. Apart from this theorem, three other early results on alternating cycles (paths) were presented in Daykin (1976), Bollobás and Erdös (1976) and Chen and Daykin (1976), who gave criteria for the existence of alternating cycles of certain lengths for a complete graph no node of which is incident to more than k edges of the same colour, provided that the number of nodes exceeds a certain threshold depending on k.

Up to now, these early results have fostered a stream of research of some hundred papers (see Bang-Jensen and Gutin, 1997 for a survey). While most contributions to this topic address the case of 2-edge coloured alternating paths (cycles) and alternating Hamiltonian paths (cycles), the scope of research has, more recently, been extended to explore also, among other cases, the case of alternating cycles on r-edge coloured graphs with $r > 2$ (see Yao (1996) and Abouelaoualim et al (2009), for example) and the question of whether there exist, on an edge

coloured graph, subgraphs other than paths and cycles (such as trees, or node partitions with specific properties) such that all edges have the same colour or differ in colour (see Kano and Li, 2008, for a survey).

The majority of this stream of research on alternating subgraphs focusses almost exclusively on graphs that are complete and/or bipartite, in particular those papers that deal with Hamiltonian paths (cycles). As will become clearer in section 4 of this chapter and in section 3 of chapter 5, we are well advised to exploit the specific structure of the graph G underlying our MSSP according to Definition 3. Therefore we cannot expect to benefit from more details of the literature on alternating paths (cycles) and we will stop our survey on this body of research here.

3.3 The Travelling Salesman Problem and generalisations

The Hamiltonian path problem of the previous chapter can be generalized to the Travelling Salesman Problem (TSP).

Definition 8 *(Travelling Salesman Problem - TSP)*

For an undirected graph $G(N, E)$ and a ("cost") function $c : E \to \mathbb{R}$ the Travelling Salesman Problem consists in finding a Hamiltonian cycle

$$i_1 - i_2 - i_3 - ... - i_{|N_G|} - i_1, \text{ with } i_k \in N_G \text{ for } 1 \leq k \leq |N_G|$$

on G such that

$$\sum_{1 \leq k \leq |N_G|} c[(i_{k-1}, i_k)] + c[(i_{|N_G|}, i_1)]$$

is minimal.

Remark 9 *(1) In the literature the TSP is often defined only for complete graphs. This is not a restriction as we can transform a TSP on an arbitrary graph $G(N, E)$ into a TSP on a complete graph by defining*

$$c(f) := \sum_{e \in E} c(e) + 1 \text{ for all } f \in (N_G \times N_G) - E.$$

Then the TSP on the graph G has a solution if and only if we have

$$\sum_{1 \leq k \leq |N_G|} c[(i_{k-1}, i_k)] + c[(i_{|N_G|}, i_1)] \leq \sum_{e \in E} c(e)$$

for an optimal solution of the TSP on the complete graph.

(2) Obviously, the problem of deciding on the Hamiltonicity of a graph is a special case of the TSP as a graph is Hamiltonian if and only if the TSP on this graph has a feasible solution.

The TSP was probably first stated in a German handbook for traveling salesmen (Voigt, 1831; cf. Müller-Merbach 1983) and found its way into the mathematical literature in the 1930s (see Hoffman and Wolfe (1985) and Schrijver (2003) for more details on the history of the problem). Since then, it has become one of the most thoroughly investigated combinatorial problems. Detailed surveys on the development of research on the topic, with respect to the mathematical structure of the TSP, its applications as well as algorithms for solving it, can be found in Bellmore and Nemhauser (1968), Lawler, Lenstra, Rinnooy Kan and Shmoys (1985), Jünger, Reinelt and Rinaldi (1995), Burkhard, Deineko, van Dal, van der Veen and Woeginger (1998), and Gutin and Punnen (2002). Additionally, Gutin (2009) provides an excellent introductory overview, and Orman and Williams (2004) compare various different ways of modelling the TSP as an Integer Programming problem.

Obviously, our MSSP is not immediately a TSP because the MSSP requires us to find a Hamiltonian path that satisfies the additional constraint that the successor (or predecessor) of each node is its twin-node. There are several generalisations of the TSP in the literature that originate from adding a specific type of constraint to the TSP. Among these there are the TSP with time-windows, in which some nodes have to be visited during a certain period of time (see Dumas, Desrosier, Gelinas and Solomon (1995), for example), the TSP with precedence constraints, in which certain nodes can only be visited after certain other nodes have been visited (see Balas, Fischetti and Pulleyblank (1995), for example), and the TSP with pickup and delivery, in which each node is associated with a "pickup quantity" and a "delivery quantity" and a feasible Hamiltonian cycle satisfies the additional condition that the quantity transported along the cycle does not exceed a certain capacity (see Gendreau, Laporte, Vigo (1999), for example). A large variety of further generalisations can be found in the book chapters by Balas (2002), Barvinok, Gimadi and Serdyukov (2002), Fischetti, Salazar-Gonzáles and Toth (2002), and Kabadi and Punnen (2002).

A particular one among these generalisations of the TSP is of interest for us as the MSSP can be considered a direct subcase of it: the Clustered Traveling Salesman Problem (CTSP). The CTSP was introduced into the literature by Chisman (1975). A short overview of the literature and of several applications can be found in Laporte and Palekar (2002).

Definition 10 *(Clustered Traveling Salesman Problem - CTSP)*

For an undirected graph G, a partition of the node set into sets ("clusters") N_i, $1 \leq i \leq n$, with $N_1 + N_2 + ... + N_n = N_G$ and a function $c : N_G \times N_G \rightarrow \mathbb{R}$, the Clustered Traveling Salesman Problem consists in finding an optimal solution of the TSP on G under the additional constraint that the nodes of each cluster appear in the Hamiltonian cycle in a consecutive order.

Remark 11 *(1) Also the CTSP is typically defined on a complete graph in the literature. Remark* 9(1) *applies analogously.*

(2) The twin-constrained Hamiltonian path problem on a graph G with respect to the twin node function b (and hence the MSSP) can directly be stated as an CTSP on a graph G' that results from adding to the underlying graph G a twin-node pair of dominating nodes and partitioning the node set of G' into clusters of cardinality 2 *such that each cluster contains one of the nodes $i \in N_{G'}$ of the graph and its twin-node $b(i)$. Then the twin-constrained Hamiltonian path problem on G with respect to b is feasible if and only if the CTSP on G' has a feasible solution.*

Despite its specific structure of the node set and the additional constraint that all nodes within a cluster must be visited consecutively, the CTSP is eventually equivalent to the TSP in the sense that any instance of a TSP can be transformed into an instance of the CTSP, and vice versa.

Proposition 12 *(Reducibility of CTSP to TSP and vice versa)*

Let $G(N, E)$ be an undirected graph, a cost function $c : E \to \mathbb{R}$, and a partition $N_1 + N_2 + ... + N_n = N_G$ for the CTSP. An algorithm that is able to solve the TSP to optimality is also able to solve the CTSP to optimality, and vice versa.

Proof. An instance of the TSP can trivially be transformed into an instance of the CTSP by partitioning the node set into clusters N_i with $|N_i| = 1$ for $1 \leq i \leq |N_G|$. Conversely, an instance of the CTSP can be transformed into an instance of the TSP in the following way. We add to the cost of all edges between clusters the constant

$$M := \sum_{e \in E} c(e) + 1.$$

As the CTSP has n clusters, a feasible solution of the CTSP must contain n inter-cluster edges. Therefore, if and only if the TSP on G with the redefined cost function has a solution with

$$\sum_{1 \leq k \leq |N_G|} c[(i_{k-1}, i_k)] + c[(i_{|N_G|}, i_1)] \leq (n+1)M - 1,$$

the CTSP with the original cost function has a feasible solution and the Hamiltonian cycle found by the TSP algorithm is also the optimal solution of the CTSP. ■

Remark 13 *Note that a case parallel to the preceding proposition would be a transformation of the alternating Hamiltonian path problem into a Hamiltonian path problem. However, such a transformation is not possible because we were able to transform the CTSP into the TSP only by virtue of a redefined cost function.*

Despite the fact that the CTSP is equivalent to the TSP in the sense of the preceding proposition, solving a CTSP as a TSP cannot necessarily be considered the method of choice

as doing so would (at least partly) disregard the particular cluster structure of the CTSP. Therefore specific algorithms and heuristics for tackling the CTSP have been developed, which can be found in Jongens and Volgenant (1985), Arkin, Hassin and Klein (1994), Laporte, Potvin and Quilleret (1997), Renaud and Boctor (1998), Anily, Bramel, and Hertz (1999), Guttmann-Beck, Hassin, Khuller and Raghavachari (2000), and Dinga, Cheng and He (2007). We will not go further into the details of these algorithms here as we have reasons (see the following section) to focus on exploiting both the specific structure of the graph underlying the MSSP and the specific structure given by the twin-node function − two aspects that, to the best of our knowledge, have not been addressed in the literature.

Another variant of the TSP is worth being addressed here due to its close relation with our MSSP: the Generalized Traveling Salesman Problem (GTSP). More details on this problem can be found in Laporte, Asef-Vaziri and Sriskandarajah (1996).

Definition 14 *(Generalized Traveling Salesman Problem − GTSP)*

For an undirected graph G, a partition of the node set into sets ("clusters") N_i, $1 \leq i \leq n$, with $N_1 + N_2 + ... + N_n = N_G$ and a function $c : N_G \times N_G \rightarrow \mathbb{R}$, the Generalized Traveling Salesman Problem consists in finding a minimum cost cycle on G that passes through each cluster N_i, $1 \leq i \leq n$, exactly once.

Remark 15 (1) *In a different version of the GTSP, the minimum cost cycle must pass through each cluster at least once (see Laporte, Asef-Vaziri and Sriskandarajah ,1996).*

(2) *The GTSP can be reduced to a CTSP by doubling all nodes and redefining the cost function c in an appropriate manner (see Laporte and Semet (1999) for details). Consequently, the GTSP can also be reduced into a TSP. Conversely, each TSP can trivially be transformed into a GTSP by partitioning the node set into clusters N_i with $|N_i| = 1$ for $1 \leq i \leq |N_G|$.*

(3) *The original model for the MSSP by Goulimis (see chapter 2.2) can be seen as a GTSP with clusters of cardinality 2.*

(4) *More generally, any alternating Hamiltonian cycle problem can be reduced to a GTSP. For doing so, we replace each node of the alternating Hamiltonian cycle problem by a cluster of nodes. Each node in such a cluster represents a way of visiting the original node such that we arrive at that node via taking a blue edge and depart from that node via a red edge (or vice versa).*

Finally, for the sake of comprehensiveness, we mention two more related problems, Network Design Problem and Generalized Network Design Problem (Feremans, Labbé and Laporte, 2003; Feremans, Labbé, Letchford and Salazar, 2009). Network Design Problems consist of finding a minimum cost subgraph of a given graph, while an optimal solution of a Generalized Network Design Problem is a minimum cost subgraph of a given graph such that, for a given partition of the node set, the subgraph consists of exactly one node (or at least one node, or at most

one node, depending on the variant of the problem) from each cluster of the node set partition. Clearly, Network Design Problems are a relaxation of the TSP, while Generalized Network Design Problems are a relaxation of the GTSP. The theoretical relevance of the concepts of the Network Design Problem and the Generalized Network Design Problem lies in the fact that theoretical results (such as polyhedral, algorithmic, and complexity-related results) about these problems can be applied to a variety of subgraph problems, such as the (Generalized) Minimum Spanning Tree Problem (Feremans, C., M. Labbé and G. Laporte, 2002), of which the Network Design Problem and the Generalized Network Design Problem are relaxations.

3.4 Relevant results of complexity theory

In this section we first review some definitions and results from complexity theory and, in a second step, apply these results to the problems presented in the preceding two sections. Doing so will give us an insight into the computational "difficulty" of the twin-constrained Hamiltonian path problem. Our review of definitions and results from complexity theory mainly follows the presentation in Johnson and Papdimitrou (1985, pp. 42-58) - albeit not always in the order of their presentation and apart from some references to other sources when we go slightly more into detail. A rigorous formal treatment based on the concept of the Turing machine can be found in Jongen, Meer and Triesch (2004, chapters 18-22), for example.

We begin by distinguishing between two types of problems in the computational complexity of which we are interested.

Definition 16 *(Decision and optimization problems)*

(1) *A problem that can be solved by an algorithm that produces only a "yes" or "no" answer is called a decision problem.*

(2) *A problem is referred to as an optimization problem if solving it means finding an optimal feasible solution to the problem with respect to some objective function. For a given optimization problem that is a minimization problem and a given number b, we call the decision problem of whether there exists a feasible solution to the optimization problem with an objective function value less than or equal to b the decision problem version of the optimization problem.*

The branch of complexity theory we deal with here is concerned with analysing the worst case behaviour of an algorithm with respect to running time. The following definition introduces a way of describing the running time of an algorithm and introduces the class of decision problems that is the most relevant for the present thesis. It goes back to a suggestion by Edmonds (1965a) and, independently, Cobham (1965), according to which an algorithm should be considered "good" (or "efficient") for practical purposes if the number of computer operations needed to solve it depends polynomially on the input data.

Definition 17 *($O(f(n)$-notation and the class of polynomial-time decision problems $-$ $\mathbf{P}$)*

(1) *For a function $f : \mathbb{N} \to \mathbb{R}$, we say that the running time of an algorithm is $O(f(n))$ iff there exists a constant $c > 0$ such that the number of steps that an algorithm needs to solve a problem for all instances of size n has an upper bound of $cf(n)$ if n is sufficiently large.*

(2) *An algorithm for which such a polynomial function f exists is said to be efficient or a polynomial-time algorithm. The class of all decision problems that can be solved by polynomial-time algorithms is denoted by $\mathbf{P}$.*

Remark 18 *In this and all following chapters we will equate the input size n with the cardinality of the node set of the graph that our problem is based on, and we will count as one, single (unit time) step all summations, multiplications, and all operations that compare the size of two given numbers (such operations are called elementary arithmetic operations). This simplification is justified because the number of steps needed for multiplications and comparisons is polynomially bounded by the number of steps needed for summations and we are only concerned with the question of whether there exists, or is likely to exist, an efficient, i.e. polynomial time algorithm for a problem. This simplification also implies that we will disregard the details of how the actual number of steps needed for an elementary arithmetic operation depends on the numerical size of the input data. In line with most of the literature on combinatorial optimization (cf. Papadimitrou and Steiglitz,1998, chapter 8), it suffices for us to know that this actual number of steps, and the memory space needed for carrying them out, are bounded by a polynomial in the numerical size of the input data. More precisely speaking, an algorithm that requires only polynomial time with respect to elementary arithmetic operations and in which the space needed for carrying out each of these operations is bounded by a polynomial in the numerical size of the input data is called strongly polynomial (cf. Schrijver, 2003).*

The subsequent definition presents two concepts that are important tools for comparing the complexity of two algorithms.

Definition 19 *(Polynomial-time reducibility and polynomial-time transformability)*

(1) *A problem A is called polynomial-time reducible to a problem B if there exists an algorithm for A that uses an algorithm for B as a subroutine and the algorithm for A runs in polynomial time if we count each call of the subroutine as a unit time step.*

(2) *A decision problem A is called polynomial-time transformable to a problem B if A is polynomial-time reducible to B and the algorithm for A calls only once the subroutine that solves B.*

Remark 20 (1) *It immediately follows from the definition that, if there exists a polynomial-time algorithm for problem B and problem A is polynomial-time reducible to problem B, there exists a polynomial-time algorithm for problem A.*

(2) *It follows also by means of the concept of polynomial-time reducibility that if the value of the objective function of the optimization problem can be calculated in polynomial time and the numerical size of the optimal solution is bounded by a polynomial in the numerical size of the input data, the decision problem version of an optimization problem can be solved in polynomial time if and only if the optimization problem can be solved in polynomial time.*

(3) *We note that the property of polynomial-time transformability is transitive.*

We now introduce two more classes of problems. The first class of problems was introduced by Cook (1971) and Karp (1972), while the second class was first described by Edmonds (1965b) who referred to problems in this class as problems with "good characterisations".

Definition 21 *(Polynomial-time nondeterministic decision problems – **NP**)*

An algorithm that is able to carry out an instruction of the type

***goto both** label 1, label 2,*

i.e. that can carry out arithmetic operations in an exponential number of branches of a search tree in parallel at the same time, is called a nondeterministic algorithm. A nondeterministic algorithm is said to solve a decision problem with input size n in polynomial time iff there exists a polynomial function f such that the number of steps taken in each branch of the search tree is $O(f(n))$. The class of decision problems that can be solved by polynomial-time nondeterministic algorithms is denoted by **NP**.

The following second class of decision problems consists, loosely speaking, of those decision problems A for which all mathematical objects S that are solutions of A are sufficiently small (i.e. bounded by a polynomial in the size of the instance of A) and there exists a (certificate-checking) algorithm C that can verify in polynomial time for every S that this S is indeed a "yes" instance of the decision problem A.

Definition 22 *(Decision problems with the succinct certificate property)*

A decision problem A is said to have the succinct certificate property iff there exists a polynomial-time algorithm for another decision problem C whose instances are given by an instance of A and object S whose size is bounded by a polynomial in the size of the instance of A, such that any instance of A is a "yes" instance for problem A if and only if there exists an S such that the instance of C (given by S and the instance of A) is a "yes" instance of C.

A famous theorem by Cook (1971) states that the two classes of decision problems previously introduced are, in fact, equivalent.

Theorem 23 *For a decision problem A the following three statements are equivalent:*

(1) *The problem A has the succinct certificate property.*

(2) *The problem A is an element of* **NP**.

(3) *The problem A is polynomial-time transformable to the decision problem version of a Binary Integer Programming problem.*

Proof. See Cook (1971), or Papadimitrou and Steiglitz (1998), pp. 353-358. ■

Two more complexity classes are worth considering here. They consist of problems that, regarding their complexity, must be considered particularly "difficult".

Definition 24 *(***NP***-complete and* **NP***-hard problems)*

(1) *A decision problem is called* **NP***-complete if it is an element of* **NP** *and if every problem in* **NP** *is polynomial-time transformable to it.*

(2) *A problem is referred to as* **NP***-hard if it is not an element of* **NP***, and all problems in* **NP** *are polynomial-time reducible to it.*

Remark 25 (1) *It follows directly from this definition that* $\mathbf{P} = \mathbf{NP}$ *if and only if there exists a polynomial-time algorithm for an* **NP***-complete problem. Up to now neither has such an algorithm been found, nor could it be proved that such an algorithm does not exist.*

(2) *The previous theorem implies that Binary Integer Programming is an* **NP***-complete problem.*

We now apply these concepts and results from complexity theory to the problems presented in the previous two sections. Obviously, the Hamiltonian path (cycle) problem, the alternating Hamiltonian path (cycle) problem and the twin-constrained Hamiltonian path problem (hence also the MSSP) are decision problems, while the TSP, the CTSP and the GTSP are optimization problems. The following statements address the complexity of these problems. The first theorem, which was a milestone in the theory of complexity for combinatorial optimization problems, is due to Karp (1972).

Theorem 26 *(Complexity of the Hamiltonian path (cycle) problem)*

For a given undirected graph $G(N, E)$*, the Hamiltonian path (cycle) problem is* **NP***-complete.*

Proof. We have seen in chapter 2.1 that the Hamiltonian path (cycle) problem can be modeled as a Binary Integer Programming problem, hence we know from Theorem 23 that the Hamiltonian path (cycle) problem is in **NP**. A proof that every problem in **NP** can be transformed to the Hamiltonian path (cycle) problem can be found in Karp (1972) or Papdimitrou and Steiglitz (1998, chapter 15). ■

Proposition 27 *(Complexity of the alternating Hamiltonian path (cycle) problem)*

For a given undirected graph $G(N, E)$ and an edge colouring with 2 colours, the alternating Hamiltonian path (cycle) problem is an **NP***-complete problem.*

Proof. Based on our model in chapter 2.1 it is easy to see that the alternating Hamiltonian path (cycle) problem can be modeled as a Binary Integer Programme, hence (Theorem 23) the problem is in **NP**. The proof of Proposition 7 presents a polynomial-time transformation to the Hamiltonian path (cycle) problem. Taking into account the preceding theorem finishes the proof. ■

Proposition 28 *(Complexity of TSP, CTSP and GTSP)*

For a given undirected graph $G(N, E)$, a function $c : E \to \mathbb{R}$, and, if applicable, a partition

$$N_1 + N_2 + ... + N_n = N_G,$$

the TSP, the CTSP and the GTSP are **NP***-hard.*

Proof. Being optimization problems, TSP, CTSP and GTSP are not in **NP**. As noted in Remark 9(2), the Hamiltonian cycle problem can be reduced to TSP in polynomial time. Then Theorem 26 implies that TSP is **NP**-hard. Consequently, because of Proposition 12, CTSP is **NP**-hard. With Remark 15(2), or, alternatively, Remark 15(4) we can conclude that GTSP is **NP**-hard. ■

Theorem 29 *(Complexity of the twin-constrained Hamiltonian path problem)*

For a given undirected graph $G(N, E)$ with $N = \{1, 2, ..., 2n\}$ and a twin-node function b, the twin-constrained Hamiltonian path problem on G with respect to b is **NP***-complete in the general case.*

Proof. The twin-constrained Hamiltonian path problem on G with respect to b can be modeled as a Binary Integer Programme (chapter 2.3). Hence we can conclude from Theorem 23 that it is in **NP**. For a given graph $G'(N', E')$ with $N' = \{1, 2, ..., n\}$ and an (arbitrary) edge set E', we define the graph $G(N, E)$ with $N = \{1, 2, ..., 2n\}$ and

$$E := E' + \{(i, j) \in N \times N : (i - n, j - n) \in E'\}$$

and define the twin-node function $b : N \to N$ by virtue of

$$b(i) := i + n \text{ for all } i \in N'$$

and

$$b(i) := i - n \text{ for all } i \in N - N'.$$

Then the twin-constrained Hamiltonian path problem on G with respect to b is feasible if and only if the Hamiltonian path problem on G' is feasible. (The feasibility of the Hamiltonian path problem on G' follows from the feasibility of the twin-constrained Hamiltonian path problem on G by contracting the edges given by the twin-node function; the converse is trivial.) The fact that our construction of G from G' can be carried out in polynomial time implies that

there exists a polynomial-time transformation from the Hamiltonian path problem on G' to a twin-constrained Hamiltonian path problem (on G, with respect to the twin-node function b we have defined). Given this situation, the **NP**-completeness of the twin-constrained Hamiltonian path problem follows from Theorem 26. ■

Remark 30 *For an undirected graph G', the graph G we have just defined is, when we include in E the edges $(i, b(i))$ given by our twin-node function, the Cartesian product of G' with the complete graph K_2 and is called the prism over G'. The Hamiltonicity of the prism over a graph is a necessary condition for the Hamiltonicity of a graph and plays a significant role in the theory of Hamiltonicity (Kaiser, Ryjáček, Král, Rosenfeld and Voss, 2007). We have just shown that the twin-constrained Hamiltonicity of the prism over a graph (with the twin-node function given by the edges that have been "generated" by K_2) is a sufficient condition for the existence of a Hamiltonian path on a graph.*

We conclude from the preceding theorem that the twin-constrained Hamiltonian path problem on threshold graphs, i.e. our MSSP, might well be an **NP**-complete problem. Therefore it makes sense for us, instead of further following the path of the literature on problems related to our MSSP (such as the alternating Hamiltonian path problem, the TSP, the CTSP and the GTSP), to proceed our analysis by exploring more in detail the particularities of the graph underlying our MSSP.

4 Threshold graphs: definition and basic characteristics

In the past chapter, the twin-constrained Hamiltonian path problem (and hence the MSSP) was looked at in the context of existing research and we clarified the respect in which it is a specific type of Hamiltonian path problem, a specific variant of the TSP and how it can be reduced to these problems. One specific aspect of the MSSP that we have not considered yet, but should consider in view of the final section of the previous chapter, consists in the fact that the underlying graph has a very specific structure: it is a so-called "threshold graph". This chapter introduces the concept of threshold graphs, presents some basic properties of threshold graphs that will be useful for our analysis in the following chapters and, in doing so, provides the theoretical background on which we will build our approach for solving the problem of recognising twin-constrained Hamiltonian threshold graphs in later chapters.

4.1 Definition and examples

The term "threshold graph" was coined by Chvátal and Hammer (1973), who were interested in the type of graphs whose stable sets can be distinguished from unstable sets by a single hyperplane in the space of the characteristic vectors for all subsets of nodes of a graph. Independently from Chvátal and Hammer, Ecker and Zaks (1977) described the same mathematical structure in their studies of graph labeling for open shop scheduling, while Henderson and Zalcstein (1977) called the same type of graphs "PV_c-definable graphs" when analysing the flow of information in parallel processing. Another early use of threshold graphs is Koren's (1973) work, who came across the concept in his studies of certain degree sequences of graphs. Also the author of the present thesis, not knowing about the existing literature on threshold graphs at that point of time, "re-invented" the concept and first studied threshold graphs under the notion of "graphs with monotonic neighbourhoods".

The standard monograph on the subject is Mahadev and Peled (1995), which lists more than 100 papers related to threshold graphs, most of which were published during a comparably short period of 10 years. This monograph includes also further applications of threshold graphs, to problems such as cyclic scheduling and Guttman scales. Apart from practical applications, research on threshold graphs has primarily been motivated by the fact that threshold graphs have a beautiful structure due to which they are closely connected to other important types of (sub-)graphs and therefore a helpful tool for studying their structure. The mathematical relevance of threshold graphs is illustrated by the fact that the survey of graph classes by Brandstädt, Le and Spinrad (1999) devotes an entire chapter (one out of altogether 14 chapters) to "Threshold Graphs and Related Concepts".

The definition of a threshold graph provided here is the original one given by Chvátal and Hammer (1973).

Definition 31 *(Threshold Graph)*

An undirected graph $G(N, E)$ is called a threshold graph iff there exist positive "node weights" $w_i \in \mathbb{R}_+^*$ for all $i \in N_G$ and a threshold $t \in \mathbb{R}_+^*$ such that for all subsets $S \subseteq N_G$

$$w(S) := \sum_{i \in S} w_i \leq t \text{ if and only if } S \text{ is a stable set.} \tag{11}$$

Figure 5 provides three examples of threshold graphs with the appropriate node weights and thresholds (example (c) is given in Golumbic, 1980).

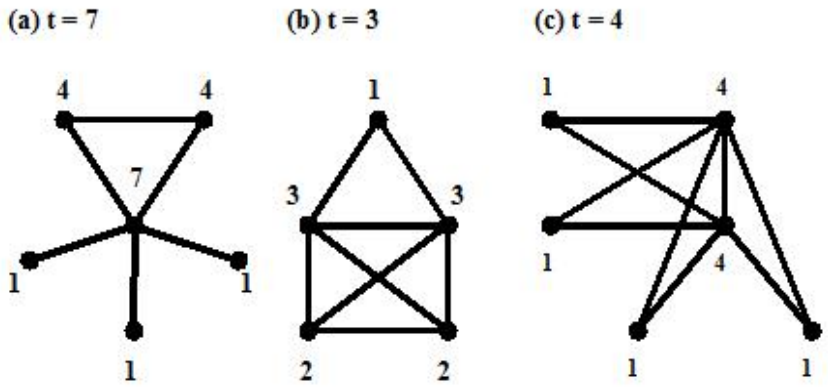


Figure 5: Examples of threshold graphs

There are also some rather trivial graphs that do not qualify as threshold graphs, among which are all paths P_n for $n \geq 4$, all chordless cycles C_n for $n \geq 4$, and the matching $2K_2$. Four examples of non-threshold graphs can be found in Figure 6. That the depicted graphs violate the threshold property (11) can easily be seen on the basis of the nodes i, j, k and l. As $\{i, k\}$ and $\{j, l\}$ are stable sets, any assignment of node weights and a threshold would imply

$$w_i + w_k \leq t \text{ and } w_j + w_l \leq t,$$

while the cliques $\{i, l\}$ and $\{j, k\}$ required the inequalities

$$w_i + w_l > t \text{ and } w_j + w_k > t,$$

which would lead to a contradiction when summing up the weights of these four nodes.

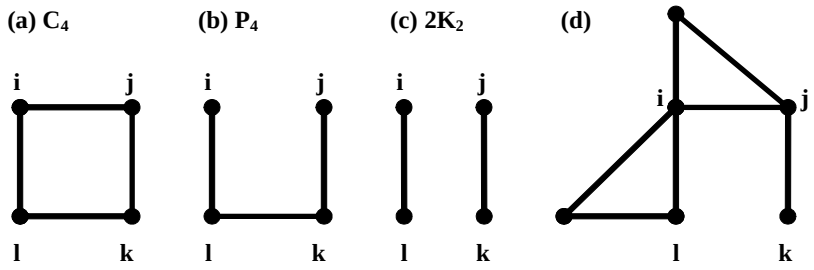


Figure 6: Examples of non-threshold graphs

Some complementary remarks may further clarify the definition of the threshold graph given above:

Remark 32 *(a) The threshold property* (11) *can be interpreted on the basis of the following question. If, for a graph with node set* $N_G = \{1, 2, ..., n\}$*, every subset of nodes* $P \subseteq N_G$ *is described by its characteristic vector* $x = (x_1, x_2, ..., x_n)$ *with*

$$x_i = 1 \text{ if } i \in P, \text{ and } 0 \text{ otherwise, for all } i = 1, 2, ..., n,$$

does there exist a single hyperplane (a "separator")

$$\sum_{i \in N_G} w_i x_i \leq t$$

that separates the characteristic vectors of all stable sets from those of all non-stable sets? If and only if the answer is affirmative, the graph is threshold.

(b) Orlin (1977) has shown that for every threshold graph, the family of node weights w_i *for which the threshold* t *is minimal is unique and integer.*

(c) It follows directly from the definition that every induced subgraph of a threshold graph is also threshold. Consequently, threshold graphs do not contain the path P_4*, the chordless cycle* C_4 *and the matching* $2K_2$*. Moreover, Chvátal and Hammer (1973) have demonstrated that this property entirely characterizes threshold graphs: any graph without the aforementioned induced subgraphs is a threshold graph.*

4.2 Basic characteristics of threshold graphs

The following theorem characterizes threshold graphs on the basis of a value function on the set of nodes, and establishes a very convenient approach to analysing threshold graphs. This theorem (with a slight modification) has first been provided by Chvátal and Hammer (1973) and, independently, by Henderson and Zalcstein (1977).

Theorem 33 *(Characterisation of threshold graphs by a value function)*

For a graph $G(N, E)$ the following statements are equivalent:

(i) G is a threshold graph.

(ii) There exists a minimum value $\alpha \in \mathbb{R}_+^*$ and a value function $v : N_G \to \mathbb{R}_+^*$ that assigns a positive $v(i) \in \mathbb{R}_+^*$ to every node $i \in N_G$ such that for all nodes $i, j \in N_G$ with $i \neq j$

$$(i, j) \in E_G \text{ if and only if } v(i) + v(j) \geq \alpha \ . \tag{12}$$

Proof. See Chvátal and Hammer (1977), Golumbic (1980), or Mahadev and Peled (1995). ■

This directly allows us to reformulate the MSSP. We will stick to this way of looking at the MSSP for the reminder of this thesis.

Proposition 34 *(Minimum Score Separation Problem - Final Approach)*

Let $G(N, E)$ be a threshold graph and $b : N_G \to N_G$ a twin node function. Then the Mimimum Score Separation Problem consists in solving the Twin-Constrained Hamiltonian Path Problem on G with respect to b.

Proof. Obviously, condition (12) is the one imposed on the graph in our second model of the MSSP (see definition 4). ■

Note, however, that the graph on which we look for a Hamiltonian path in order to solve the MSSP affirmatively is *not* a threshold graph in the general case. Instead, in the MSSP the twin node property partly destroys the threshold structure because we have not made any further assumptions about the twin node function b. In the general case, the twin node function will not induce edges between twin nodes in a way that the threshold property is respected. So despite the fact that the underlying graph G of the Twin-Constrained Hamiltonian-Path Problem in question is a threshold graph, we will have to look for a Hamiltonian path on a non-threshold graph However, we will leave this out of account for the moment, concentrate on the threshold condition, and come back to the twin node condition in a later chapter, namely chapter 7.

Remark 35 *The preceding theorem directly implies that the complement of a threshold graph is also threshold. After transforming the values $v(i)$ and α into integers, the complement of G can be obtained by introducing new values $\hat{v}(i) := t - v(i)$ for all nodes $i \in N_G$, and a new minimal value $\overset{\wedge}{\alpha} := \alpha - 1$. As a consequence, the approach suggested in this paper would work also for a "Maximum Score Separation Problem".*

Another characterization of threshold graphs that we will use in the following also goes back to early work of Chvátal and Hammer (1973).

Theorem 36 *(Characterisation of threshold graphs by their degree partition)*

For a graph $G(N, E)$ the following statements are equivalent:

(i) G is a threshold graph.

(ii) The indices of the degree partition $N_G = D_0 + D_1 + ... + D_m$ of G provide the full information about adjacency for all nodes $i \in D_k$ and $j \in D_l$ by virtue of

$$(i, j) \in E_G \text{ if and only if } k + l > m \ . \tag{13}$$

Proof. See Chvátal and Hammer (1977), Golumbic (1980), or Mahadev and Peled (1995). ■

Figure 7, taken from Golumbic (1980), illustrates the general structure of a threshold graph according to Theorem 36. Lines between the node sets D_0, D_1, D_2, ..., D_m indicate adjacency. The dotted set of isolated nodes D_0 may be empty, and the dotted set $D_{\lceil \frac{m}{2} \rceil}$ does not exist if m is even.

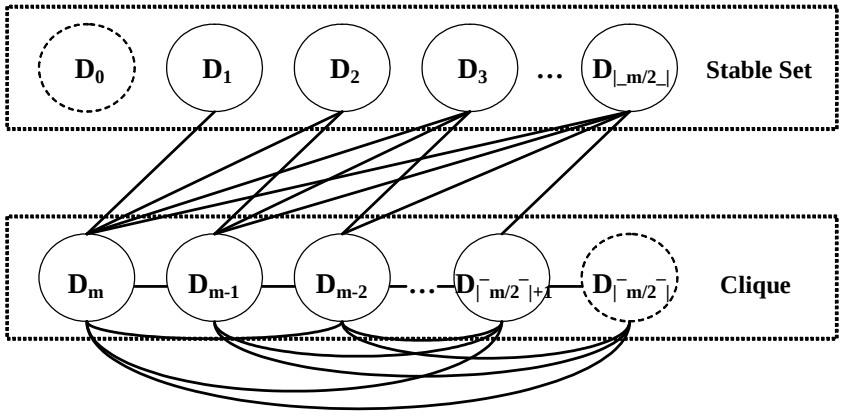


Figure 7: Degree partition of a threshold graph

Remark 37 *If m is even, the sets $D_m, D_{m-1}, ..., D_{\lceil \frac{m}{2} \rceil + 1}$ contain all nodes $i \in N_G$ with $v(i) \geq \frac{\alpha}{2}$, while the nodes $j \in N_G$ with $v(j) < \frac{\alpha}{2}$ are in the sets $D_0, D_1, ..., D_{\lfloor \frac{m}{2} \rfloor}$. If m is odd, the sets $D_m, D_{m-1}, ..., D_{\lceil \frac{m}{2} \rceil + 1}, D_{\lceil \frac{m}{2} \rceil}$ contain all nodes $i \in N_G$ with $v(i) \geq \frac{\alpha}{2}$, and, additionally, the set $D_{\lceil \frac{m}{2} \rceil}$ contains also exactly one node $j \in N_G$ with $v(j) < \frac{\alpha}{2}$ that is adjacent to all nodes $i \in N_G$ with $v(i) \geq \frac{\alpha}{2}$ (provided there exists any).*

We finally provide two more characterisations of threshold graphs (first given in Chvátal and Hammer, 1977), on which we will build our approach to the Minimum Score Separation Problem in the following. Further fundamental properties of threshold graphs can be found in the literature by Chvátal and Hammer (1973, 1977), Ecker and Zaks (1977), Henderson and

Zalcstein (1977), Golumbic (1980), Hammer, Ibaraki and Simeone (1981), and Mahadev and Peled (1995).

In order to give the reader an idea of how some basic structural elements of threshold graphs are related to each other, we will present here a proof for the subsequent two characterisations of threshold graphs. (These two characterisations are also those that the author of this thesis found before becoming aware of the literature on threshold graphs.) For our proof, we need the following lemma that was first stated by Chvátal and Hammer (1977).

Lemma 38 *Let $G(N, E)$ be a threshold graph with node set $N_G = \{1, 2, ..., n\}$, a certain family of weights (w_i) and a threshold t, from which we build a new graph G' by adding a node $n+1$. If the node $n+1$ is (i) dominating or (ii) isolated, the new graph with the node set $N_{G'} = \{1, 2, ..., n, n+1\}$ remains a threshold graph.*

Proof. (i) We will assign the weight $w_{n+1} := t$ to the new node $n+1$ and consider the new subsets $P \subseteq N_G + \{n+1\}$ with $n+1 \in P$. Among these, the stable subset $\{n+1\}$ certainly fulfils condition (11). Any other subset $P \subseteq N_G$ with $n+1 \in P$ is not stable because $n+1$ is a dominating node. As, by definition, all weights of a threshold graphs are required to be positive, also in this case property (11) still holds.

(ii) We will double all weights $(w_i)_{1 \leq i \leq n}$ of the "old" nodes and increase the threshold to $2t+1$. The new node $n+1$ will be given the weight $w_{n+1} := 1$. This leads to the following setting: formerly stable subsets obviously remain stable. Previously unstable sets $U \subseteq N_G$ had at least a sum of weights of $w(U) = t+1 > t$, which has become $2t+2$ in the new setting, such that the weight of these sets still exceeds the new threshold $2t+1$. In the case of the new sets $U + \{n+1\}$ for formerly unstable sets U, instability is maintained, and the new sum of weights is at least

$$w(U + \{n+1\}) = 2w(U) + w_{n+1} = 2(t+1) + 1 > 2t + 1.$$

Clearly, also the stable subset $\{n+1\}$ fulfils property (11). Finally, for all formerly stable set $S \subseteq N_G$, the new sets $S + \{n+1\}$ remain stable with the isolated node $n+1$, and the threshold is not violated because of

$$w(S + \{n+1\}) = 2w(S) + w_{n+1} \leq 2t + 1. \blacksquare$$

Remark 39 *The preceding lemma can be used to construct threshold graphs by successively adding (one or more) isolated or dominating nodes. Such a procedure gives rise to the adjacency structure that Theorem* 36 *describes. Consequently, Lemma* 38 *can be used as the main ingredient of a proof of Theorem* 36.

Theorem 40 *(Further characterisations of threshold graphs)*

For a graph $G(N, E)$ the following statements are equivalent:

(i) G is a threshold graph.

(ii) The vicinal preorder of G is total, i.e. for every pair of nodes $i, j \in N_G$ we have $i \precsim j$ or $j \precsim i$.

(iii) The graph G is a split graph (i.e. the node set N_G can be partitioned into a clique K and a stable set S), and the neighbourhoods of the stable nodes are nested, i.e. there exists a permutation of the stable nodes $i_1, i_2, ..., i_{|S|}$ such that $N(i_1) \subseteq N(i_2) \subseteq ... \subseteq N(i_{|S|})$.

Proof. $(i) \Rightarrow (ii)$: Let us assume that the vicinal preorder is not total, i. e. there exists a pair of nodes $i, j \in N_G$ with neighbourhoods $N(i)$ and $N(j)$ and another pair of nodes $k, l \in N_G$ such that $k \in N(i)$, $k \notin N(j)$ and $l \in N(j)$, $l \notin N(i)$. Because G is a threshold graph, it follows from Theorem 6 that $v(i) + v(k) \geq \alpha$ and $v(j) + v(l) \geq \alpha$, but that $v(j) + v(k) < \alpha$ and $v(i) + v(l) < \alpha$, which is a contradiction.

$(ii) \Rightarrow (iii)$: Let $i, j \in S$ be a pair of nodes from the stable set, hence $j \notin N(i)$. Then $N[j] \supseteq N(i)$ yields $N(j) \supseteq N(i)$, i. e. the neighbourhoods of the stable nodes are nested. In order to show that G is a split graph, it is sufficient to show that the graph G' that we obtain from G by removing all isolated nodes is a split graph. Now, let all nodes $i_1, i_2, ..., i_n \in N_{G'}$ be numbered in any order that corresponds to the vicinal preorder, i.e. $i_j \succsim i_k$ for $j \geq k$. Let $l \in N_{G'}$ be a node adjacent to the second highest node in the order, i.e. $i_{n-1} \in N(l)$. Clearly, $i_n \succsim l$. Hence the vicinal preorder property implies that $i_{n-1} \in N(i_n)$, i.e. there exists a pair of immediately subsequent nodes $i_k, i_{k+1} \in N_{G'}$ that are adjacent to each other.

Let j be the *smallest* number so that i_j is adjacent to its immediate successor. We observe that i_j is adjacent also to all nodes $i_k \succsim i_{j+1}$ because $i_j \in N(i_{j+1})$ leads to $i_j \in N(i_k)$ due to the vicinal preorder. Furthermore, all $i_k, i_l \succsim i_j$ are adjacent to each other because $i_k \in N(i_j)$ implies $i_k \in N(i_l)$. Hence, the set $K := \{i \in N_{G'} : i \succsim i_j\}$ is a clique.

If we can show that the set $N_{G'} - K$ is stable, we have finished. Let us assume the opposite would be the case, i. e. that there were two adjacent nodes $i_k, i_l \in N_{G'}$ with $i_k, i_l \not\succsim i_j$ and $i_l \in N(i_k)$. We may assume without loss of generality that $i_l \succsim i_k$. Then, due to the vicinal preorder, $i_l \in N(i_k)$ implies $i_l \in N(i_{l+1})$ as we obviously have $i_{l+1} \succsim i_k$. However, this result contradicts the fact that i_j is the smallest node that is adjacent to its immediate successor. Consequently, $S := N_{G'} - K$ is a stable set.

$(iii) \Rightarrow (i)$: If G has an isolated node, we can remove it and will still get a split graph with nested neighbourhoods for the stable nodes. If G has no isolated node, we will choose a node $j \in N_G$ that is a neighbour of the stable node with the smallest neighbourhood, namely $i_1 \in N_G$. Clearly $j \in K$. Because the neighbourhoods of the stable nodes are nested, $j \in N(i_1)$ yields $j \in N(s)$ for all stable nodes $s \in S$. Moreover, as $j \in K$, it follows that j is adjacent to all nodes in N_G, hence j is a dominating node. If we remove j from the node set of G, we will still have a split graph with nested neighbourhoods for all stable nodes. In sum: we can deconstruct the whole graph by repeatedly removing isolated or dominating nodes. If we

rebuild the graph by reversing this procedure, we can apply Lemma 38 such that G is shown to be threshold. ■

Remark 41 *(1) The partition of the node set of G into a clique K and a stable set S is not necessarily unique. In fact, there can be a subset $F \subset N_G$ the nodes in which are adjacent to all $k \in K \backslash F$, to no $s \in S \backslash F$, and not to any other $f \in F$ either. At most one of these nodes can be placed in the clique, while all others must go into the stable set. So the maximal clique $K_{\max}$ and the maximal stable set $S_{\max}$ are unique except for the placement of one node from F.*

(2) The way in which we have constructed the clique K in the preceding proof ensures at least that K is maximal, i.e. exactly one $f \in F$ has been placed in the clique iff $F \neq \varnothing$.

(3) With respect to the characterization of threshold graphs by the degree partition $N_G = D_0 + D_1 + ... + D_m$ we have the following setting: if and only if m is odd, the maximal clique is given by

$$K_{\max} = \bigcup_{i=\lceil \frac{m}{2} \rceil}^{m} D_i,$$

and the sets

$$\bigcup_{i=0}^{\lfloor \frac{m}{2} \rfloor} D_i \cup \{f\} \text{ with } f \in F := D_{\lceil \frac{m}{2} \rceil}$$

are all maximal stable sets of the graph. Conversely, if and only if m is even, the maximal stable set is given by

$$S_{\max} = \bigcup_{i=0}^{\frac{m}{2}} D_i,$$

and all maximal cliques are given by

$$\bigcup_{i=\frac{m}{2}+1}^{m} D_i \cup \{f\} \text{ with } f \in F := D_{\frac{m}{2}}.$$

(4) With respect to the characterization of threshold graphs by a value function we have: if $K_{\max}$ is a maximal clique in G, all nodes $i \in N_G$ with $v(i) \geq \frac{\alpha}{2}$ are an element of $K_{\max}$, and iff there exist nodes $j_0 \in N_G$ with $v(j_0) < \frac{\alpha}{2}$ that are adjacent to all nodes $i \in N_G$ with $v(i) \geq \frac{\alpha}{2}$, also one of these nodes is in $K_{\max}$. All other nodes $j \in N_G$ with $v(j) < \frac{\alpha}{2}$ constitute a stable set.

(5) Note that we have $k \succsim s$ and $k \not\precsim s$ for all $k \in K$ and $s \in S$.

(6) If $(i,j) \in E_G$, then $(i \in K_{\max} \vee j \in K_{\max})$, and also $(i \in K \vee j \in K)$ for any $K := N_G - S_{\max}$ with $S_{\max}$ being a maximal stable set.

5 Maximum cardinality matchings, alternating paths and Hamiltonian paths on threshold graphs

Having explored some fundamentals of the general structure of threshold graphs, we are now prepared to address the topic of matchings on threshold graphs and the way in which these are related to different types of paths. In particular, the following two subsections will analyse the relation of matchings with alternating paths and Hamiltonian paths. First of all, this topic is interesting as such because we will lead to some new insights into the structure of threshold graphs. The main reason for proceeding in this way, however, consists in the structure of the MSSP.

Let us recall our (alternative) model of the MSSP. According to our final definition, a solution to the MSSP consists in a twin-constrained Hamiltonian path on a threshold graph with the adjacency between two twin nodes being given by a twin node function $b : N_G \rightarrow N_G$. This means that the Hamiltonian path must have the structure

$$i_1 - b(i_1) - i_2 - b(i_2) - i_3 - b(i_3) - ... - i_{n-1} - b(i_{n-1}) - i_n - b(i_n) \ . \tag{14}$$

Unfortunately, we must do without any further information about the twin node function. However, as also the underlying threshold graph structure induces adjacency (or non-adjacency) between any two nodes, we can be sure nevertheless that the pairs $b(i_1) - i_2$, $b(i_2) - i_3$, ..., and $b(i_{n-1}) - i_n$ constitute a matching on a threshold graph. (While all the other edges of the Hamiltonian path (14) are given by the twin node property.) In other words: every solution to the MSSP contains a matching on a threshold graph, the pairs of which are "glued" together by the twin node function. Hence, if we were able to obtain a proper overview of all possible perfect matchings on a threshold graph, we would know immediately whether or not a particular instance of the MSSP can be solved.

This is the reason why it seems to be promising to try to approach the MSSP on the basis of a matching first, and then take up from there the question of how to construct certain paths on the graph that might be fruitful for tackling the MSSP.

We will finish this chapter by drawing some conclusions regarding the complexity of the MSSP.

5.1 Alternating paths and maximum cardinality matchings

For threshold graphs, there exists a very efficient and straight-forward maximum cardinality matching algorithm (also mentioned in Mahadev and Peled, 1995, without a proof). This subsection addresses the maximum cardinality property of this algorithm on the basis of the augmenting path theorem. Moreover, we will show that the matching gained by the algorithm reveals so much information about the structure of the threshold graph in question that we can obtain rather strong results about the existence of alternating paths in general.

The matching algorithm proceeds as follows:

Algorithm 42 *(TGMA - Threshold Graph Matching Algorithm)*

Let $G(N, E)$ be an undirected graph with the set of nodes $N_G = \{1, 2, ..., n\}$ and neighbourhoods $N(i)$ for all $i \in N_G$.

[01] *Sort all nodes in an order of non-decreasing degrees. Set node $i := 0$, the matching list $M := \varnothing$, and the set of matched nodes $I := \varnothing$.*
[02] *Increase i by 1. If $i = n$ then STOP.*
[03] *If $i \in I$ or $N(i) \backslash I = \varnothing$ then go to step* [02].
[04] *Pick a node $j \in N(i) \backslash I$, add (i, j) to matching list M, add i, j to set of matched nodes I, go to* [02].

For further results, we will distinguish between different types of alternating paths. In contrast to the usual definition of alternating paths, our concept explicitly accounts for the subset of the matching that the alternative path in question is derived from.

Definition 43 *(Alternating T-paths)*

Let $G(N, E)$ be a graph, M a matching on G, and $T \subseteq M$. An alternating T-path relative to M is a path

$$i_0 - i_1 - i_2 - i_3 - ... - i_{n-1} - i_n$$

such that every consecutive pair of edges in the path contains one edge from T and one edge that is not an element of the matching, and all elements of T are edges of the path.

a) If n is even and the node i_0 is not incident to any edge in the matching, the path is called an even T-path. The node i_0 is called the exposed node of the path.

b) If n is odd and the number of edges that are element of the matching is greater than the number of those edges that are not in the matching, we will call the path a matching-dominated T-path.

c) If n is odd and both the nodes i_0 and i_n are not incident to any edges in the matching, the path is called an augmenting T-path. The nodes i_0 and i_n are called the exposed nodes of the path.

The succeeding Theorem 44 provides criteria for the existence of alternating (T-)paths on threshold graphs.

One way of looking at the concept of alternating paths consists in seeing the existence of alternating T-paths as an affirmative answer to the question of whether a set T of some edges that match a certain subset of nodes can be complemented by other edges such that there exists a path that connects all nodes of the subset (as in the case of a matching-dominated T-path)),

or such that there exists a path that connects all nodes of the subset *and* one or two additional nodes (as in the cases of an even T-path or an augmenting T-path, respectively).

This is why the following theorem will later be the backbone of our effort of constructing paths that are solutions to the MSSP. Moreover, it must be considered interesting as such that for threshold graphs, as Theorem 44 shows, the information about the existence of alternating paths can be gained almost solely on the basis of the aforementioned matching algorithm. Finally, our theorem directly implies a proof for the fact that TGMA terminates with a maximum cardinality matching.

Theorem 44 *(Alternating T-paths on threshold graphs)*

Let $G\,(N, E)$ be a threshold graph and M a matching on G that has been obtained from TGMA. Then for all $T \subseteq M$ the following statements hold:

- (i) *There exists a matching-dominated T-path relative to M.*
- (ii) *There exists an even T-path relative to M if and only if its exposed node is adjacent to some node from each edge in T.*
- (iii) *There exists no augmenting T-path relative to M.*

Proof. (i) Let the elements of T be given by $(i_k, j_k) \in T$ for $1 \leq k \leq |T|$. Since the vicinal preorder of G is total, we can renumber the edges such that

$$i_1 \precsim i_2 \precsim ... \preceq i_{|T|-1} \precsim i_{|T|},$$

which yields $j_k \in N(i_k) \subseteq N[i_{k+1}]$ for $1 \leq k \leq |T| - 1$. Hence

$$i_1 - j_1 - i_2 - j_2 - ... - i_{|T|-1} - j_{|T|-1} - i_{|T|} - j_{|T|}$$

is a matching-dominated T-path relative to M.

(ii) Let $i_0 \in N_G$ be the exposed node of the path.

a) $\Leftarrow$: Let the elements of T be given by $(i_k, j_k) \in T$ for $1 \leq k \leq |T|$ in a way such that

$$i_1 \precsim i_2 \precsim ... \preceq i_{|T|-1} \precsim i_{|T|}, \text{ and}$$
$$i_k \precsim j_k \text{ for all } 1 \leq k \leq |T|.$$

The latter condition yields $i_0 \in N(j_k)$ for all $1 \leq k \leq |T|$ because i_0 is adjacent to at least one node of each pair in T. Hence

$$i_1 - j_1 - i_2 - j_2 - ... - i_{|T|-1} - j_{|T|-1} - i_{|T|} - j_{|T|} - i_0$$

is an even T-path relative to M.

b) $\Rightarrow$: Let the even T-path be given as

$$i_0 - i_1 - j_1 - ... - i_{|T|-1} - j_{|T|-1} - i_{|T|} - j_{|T|}\ , \tag{15}$$

and $K_{\max}$ denote a maximal clique in in N_G.

Case (1) : $i_0 \in K_{\max}$. Then we have finished because in a threshold graph, at least one among the two nodes incident to a given edge is an element of $K_{\max}$, cf. Remark 41(6).

Case (2) : $i_0 \notin K_{\max}$. We assume that $i_0 \in N(i_k)$ for some k with $1 \leq k \leq |T| - 1$, which is certainly the case for $k = 1$. From $i_0 \in N(i_k)$ and $i_0 \notin K_{\max}$ follows $i_0 \precsim i_k$, and $i_0 \not\succsim i_k$, due

to Remarks 41(5) and 41(6). Further, condition (15) implies that i_k has been matched with j_k. As TGMA proceeds in a non-decreasing order of nodes and i_0, though being adjacent to i_k, has remained unmatched, while j_k has been matched with i_k, we must have $j_k \precsim i_0$. This yields $i_{k+1} \in N(j_k) \subseteq N[i_0]$. By induction, we obtain $i_0 \in N[i_k]$ for all k with $1 \leq k \leq |T|$.

(*iii*) First let us note that when TGMA terminates there cannot be left two unmatched nodes that are adjacent to each other. We now assume the opposite of the statement to be shown, i.e. that there exists an augmenting T-path

$$i_0 - i_1 - j_1 - ... - i_{|T|-1} - j_{|T|-1} - i_{|T|} - j_{|T|} - i_{|T|+1}.$$

Furthermore let $K_{\max}$ be a maximal stable set in N_G. Then the nodes i_0 and $i_{|T|+1}$ cannot be both elements of $K_{\max}$ because there were two adjacent unmatched nodes otherwise. Therefore, let us assume without loss of generality that $i_0 \notin K_{\max}$. The path $i_0 - i_1 - j_1 - ... - i_{|T|} - j_{|T|}$ is an even T-path relative to M. Hence (and because of $i_0 \notin K_{\max}$) we can apply the same line of reasoning as in part (*ii*) *b*) *Case* (2) of the proof, which yields by induction $i_0 \in N[i_{|T|}]$, and even $i_0 \in N[i_{|T|+1}]$. This contradicts our observation that TGMA does not leave two adjacent nodes unmatched when terminating. Consequently, the assumption that there exists an augmenting T-path is wrong. ■

Remark 45 *Note that statement* (*i*) *and the sufficiency of condition* (*ii*) *have been derived without referring to the way in which TGMA operates.*

For what follows in the succeeding sections, it is helpful to distinguish between two different variants of the algorithm $TGMA$. The two algorithms mainly differ with respect to the node they choose in step [04]. The first variant, $TGMA_{\min}$ always picks a node with the lowest degree that is still available, provided this node is adjacent to the node to be matched.

Algorithm 46 *($TGMA_{\min}$)*

[01] *Sort all nodes in an order of non-decreasing degrees.*
Set node $i := 0$*, the matching list* $M := \varnothing$*, and*
the set of matched nodes $I := \varnothing$*.*

[02] *Increase* i *by* 1*. If* $i = n$ *then STOP.*

[03] *If* $i \in I$ *or* $N(i) \backslash I = \varnothing$ *then go to step* [02].

[04] *Set* $j := i + 1$*. Increase* j *by* 1 *until* $j \in N(i) \backslash I$*.*
Add (i, j) *to matching list* M*,*
add i*,* j *to set of matched nodes* I*, go to step* [02].

The second version of the algorithm, $TGMA_{\max}$, always chooses in step [04], if possible, the node with the highest degree.

Algorithm 47 *($TGMA_{\max}$)*

[01] *Sort all nodes in an order of non-decreasing degrees.*
Set node $i := 0$, node $j := n$,
and matching list $M := \varnothing$.

[02] *Increase i by 1. If $i \geq j$ then STOP.*

[03] *If $i \notin N(j)$ then go to step* [02].

[04] *Add (i, j) to matching list M,*
decrease j by 1, and go to step [02].

Definition 48 *(Modest and greedy matchings on threshold graphs)*

Let $G(N, E)$ be a threshold graph with neighbourhoods $N(i)$ for all $i \in N_G$. A matching that has been obtained by the algorithm $TGMA_{\min}$ is called modest, and a matching that results from running $TGMA_{\max}$ greedy.

Remark 49 *The distinction between modest and greedy matchings on threshold graphs should not be confused with the well-known concept of the greedy algorithm in matroid theory (cf. e.g. Nemhauser and Wolsey 1999, chapter III.3). In fact, from the latter perspective, all variants of $TGMA$ can be considered greedy in a basic sense as they simply work along the list of nodes (in an order of non-decreasing degrees) and pick some suitable available node without "worrying" about the consequences for the choices to follow at later stages.*

Corollary 50 *If G is a threshold graph, TGMA terminates with a maximum cardinality matching, and there exists a maximum cardinality matching algorithm that terminates after $O(n \log_2 n)$ steps.*

Proof. The maximum cardinality property is directly implied in Theorem 44(*iii*) on the basis of the well-known augmenting path theorem going back to Petersen (1891). With regard to computational complexity, we observe that $TGMA_{\max}$ is the best version of $TGMA$ with regard to computational complexity as, when using $TGMA_{\max}$, we do not have to keep track of the nodes that have already been matched. Further, we note that steps [02], [03], and [04] of $TGMA_{\max}$ occur at most n times. Thus, the most expensive operation is the sorting of nodes according to their degrees, which can be accomplished easily in $O(n \log_2 n)$ time. ■

The algorithm $TGMA_{\max}$ is not only favourable in terms of computational complexity; it also provides us with matchings that have a useful property that will turn out to be useful when we later will analyse alternating cycles on threshold graphs.

Proposition 51 *(Degree property of $TGMA_{\max}$)*

Let $G(N, E)$ be a threshold graph and M a matching on G that has been obtained from $TGMA_{\max}$. Then for all $(i_1, j_1) \in M$ there exists no edge $(i_2, j_2) \in M$ with $dg(i_1) > dg(i_2)$ and $dg(j_1) > dg(j_2)$.

Proof. The property follows directly from the fact that $TGMA_{\max}$ proceeds along the list of nodes in a non-decreasing order and always chooses a mate for the matching that has the highest degree among all remaining nodes that have not been matched yet. ■

Remark 52 *(1) The fastest available algorithm for the general non-bipartite cardinality matching problem with n nodes and m edges, which has been developed by Micali and Vazirani (1980), requires $O(m\sqrt{n})$ time. As the vicinal neighbourhood property means "carrying forward" adjacency from one node to the next stronger one, threshold graphs typically tend to have many edges, i. e. $m > n$, especially when there are many non-isolated nodes in the maximal stable set, or when the amount of nodes in the corresponding clique is rather large. (Note, for example, that we already have at least n (undirected) edges if the smallest node is not isolated, and that k nodes in the clique imply at least $\frac{k(k-1)}{2}$ edges representing only the mutual adjacencies among these nodes.). Therefore, and since $\sqrt{n} > \log_2 n$ for $n > 16$, the algorithm presented here provides a significant computational advantage in the case of threshold graphs. Also, the obviously simple structure of the TGMA makes it particularly easy to implement.*

(2) If we always pick the smallest node $j \in N(i)\backslash I$ in step [04]*, we will arrive at a matching algorithm for interval graphs, which has been proven to yield maximum cardinality matchings by Moitra and Johnson (1989). In fact, every threshold graph is also an interval graph, as can be seen easily on the basis of the a characterization of interval graphs given by Ramaligam and Rangan (1988). According to this criterion, a graph $G(N, E)$ is an interval graph if and only if its nodes can be numbered such that*

$$(i,k) \in E_G \Rightarrow (j,k) \in E_G \text{ for all } i,j,k \in N_G \text{ and } i < j < k.$$

Clearly, numbering the nodes of a threshold graph in a non-decreasing order with respect to the vicinal preorder fulfils this condition.

(3) The general, not modified version of TGMA, leaves open the question of how to pick the node $j \in N(i)\backslash I$ in step [04]*. In the general case, different rules of how to choose j will lead to different maximum cardinality matchings on G. If we would like to generate* ***all*** *possible maximum cardinality matchings on a given threshold graph, we have to take into account two aspects: (a) Given a certain maximum cardinality matching on a threshold graph, every unmatched node could be matched at the expense of any smaller node that has been matched, due to the vicinal preorder of threshold graphs. The smaller node would then become unmatched by "handing over" its mate to the larger one such that the overall cardinality of the matching would not be affected. (b) If we always choose in step* [04] *a node $j \in N(i)\backslash I$ that is among the smallest nodes available, we clearly get a matching in which the unmatched nodes are as large as possible because TGMA matches in a non-decreasing order of nodes. Because of these two aspects (a) and (b), all maximum cardinality matchings on a certain threshold graph can be generated in the following way: first compute a matching on the basis of TGMA by always choosing in step* [04] *one node j from the smallest nodes available in $N(i)\backslash I$. In a second step, all other matchings can be derived from this initial matching by picking a subset of unmatched*

nodes and matching these nodes at the expense of some subset that consists of matched nodes of a lower or equal degree.

5.2 Hamiltonian paths and maximum cardinality matchings

We have noted at the beginning of this section that every Hamiltonian Path that is a solution to the MSSP contains a matching on a threshold graph. This property suggests to ask the question of how matchings and Hamiltonian Paths are related in the general case of threshold graphs. Though this implies stepping back from the MSSP because it means leaving aside the issue of twin nodes, analysing the relation of matchings and Hamiltonian paths on threshold graphs will be helpful for a better understanding of the MSSP and, in fact, lay the foundation of our approach to solving it.

In the general case of an arbitrary graph, the existence of Hamiltonian *paths* is closely related to the question of the Hamiltonicity of a graph, i.e. to the question of whether there exists a Hamiltonian *cycle* on that graph. In particular, any both necessary and sufficient condition for the Hamiltonicity of a graph directly implies both a necessary and a sufficient criterion for the existence of Hamiltonian paths. In order to determine if a graph has a Hamiltonian path, one simply has to add a dominating node to the graph in question and then check whether the resulting graph is Hamiltonian.

In the case of threshold graphs, also the opposite of the above statement is true, i.e. every necessary and sufficient criterion for the existence of a Hamiltoninan path directly leads to a condition for the Hamiltonianicity of the graph. In order to see this, one has to consider that threshold graphs are among those graphs that either have an isolated or a dominating node, which is an immediate implication of the fact that the vicinal preorder of threshold graphs is total (cf. also Lemma 38). As a consequence, the Hamiltonicity of a threshold graph without an isolated node can be determined by dropping one node among those with the largest neighbourhood and checking if there exists a Hamiltonian path on the subgraph induced by the remaining nodes. Concludingly, all necessary and sufficient criteria for the existence of Hamiltonician paths on threshold graphs are equivalent to characterisations of Hamiltonian threshold graphs in the sense that each of these criteria yields a characterization, and vice versa.

The issue of the Hamiltonicity of threshold graphs has already been addressed in the literature. Four different characterisations of the Hamiltonianicity of threshold paths have been given, namely in Chvátal and Hammer (1977), Golumbic (1980), Harary and Peled (1987), and Mahadev and Peled (1994), which all directly lead to a polynomial-time algorithm for recognising Hamiltonian threshold graphs. Among these characterisations, the criterion provided by Mahadev and Peled (1994) is the only one related to matchings. It is derived from their study of the longest cycles on threshold graphs and uses the characterization of threshold graphs on the basis of their degree partitions.

In the following, we will provide an alternative polynomial-time criterion by drawing on our alternating path theorem for threshold graphs and the structure of TGMA. In contrast to Mahadev and Peled's criterion, the one presented here is not based on the degree partition of threshold graphs, but instead it can be seen as the split graph counterpart to their theorem. It will be shown at the end of this subsection that our split graph criterion indeed provides a rather immediate proof of Mahadev and Peled's degree partition-based characterization of Hamiltonian threshold graphs.

Theorem 53 *(Split graph criterion for Hamiltonian paths on threshold graphs)*

If G is a threshold graph with n nodes, then there exists a Hamiltonian path on G if and only if

(i) for n even, there exists a matching M on G with cardinality $|M| = \frac{n}{2}$,

(ii) for n odd, there exists a matching M on G with cardinality $|M| = \frac{n-1}{2}$ and the unmatched node is an element of a maximal clique of G.

Proof. (i) : The existence of a matching with cardinality $\frac{n}{2}$ obviously is a relaxation of the Hamiltonian Path Problem and thus a necessary condition. Regarding sufficiency, the existence of a matching of cardinality $|M| = \frac{n}{2}$ implies according to Theorem 44(i) that there exists a matching-dominated M-path on G that covers all nodes of G.

(ii) : *a*) Concerning the sufficiency of the condition, let us recall that at least one node incident to any edge in a threshold graph is an element of $K := N_G - S_{\max}$, with $S_{\max} \subseteq N_G$ being a maximal stable set. Therefore, the unmatched node is adjacent to some node from each edge of the matching, and because of Theorem 44(ii) there exists an even M-path on G that covers all nodes.

b) Regarding the necessity of the condition, every Hamiltonian Path on $G(N, E)$ with an odd number of nodes obviously presupposes the existence of a matching of cardinality $\frac{n-1}{2}$. With respect to the second part of the condition, let us assume that there exists a matching of cardinality $\frac{n-1}{2}$ with $i_n \in N_G$ being the unmatched node after TGMA has terminated, and that there exists a subset of the maximal clique $K_0 \subseteq K_{\max}$ to which the node i_n is *not* adjacent, i.e. $i_n \notin N(k)$ for all $k \in K_0$. We will demonstrate that this implies the infeasibility of the Hamiltonian Path Problem on G. (Note that the fact that our line of argument will refer to TGMA is not a restriction. As we have argued in Remark 53(3), there exists a special case of TGMA that returns the largest unmatched node possible. In the end, our line of argument will imply that the Hamiltonian Path Problem turns out to be infeasible in any case if it is proven to be infeasible with such a large unmatched node.) For $N(i_n) = \varnothing$, there is nothing to show, therefore $N(i_n) \neq \varnothing$ in the following. The remainder of the proof will introduce node set partitions, an illustration of which can be found in Figure 8 for a better overview.

(1) In a first step, we will analyse the particular structure of the set of nodes N_G in this case. Clearly, $i_n \in S := N_G - K_{\max}$ and (by definition of i_n) $N(i_n) \subsetneqq K_{\max}$. Hence the node i_n induces a partition of the clique $K_{\max}$ such that

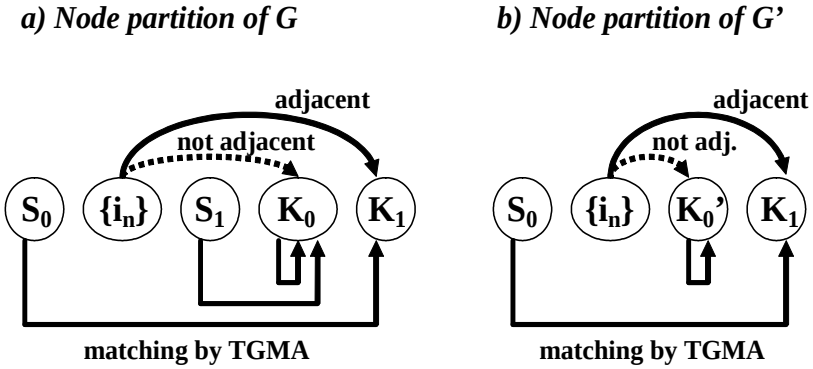


Figure 8: Structures of G and G'

$K_{\max} = K_0 + K_1$
with $K_1 = N(i_n)$ and $k \notin N(i_n)$ for all $k \in K_0$.

Due to $N(i_n) \neq \varnothing$, and because of the order of nodes in which the algorithm tries to find a matching for nodes, there must be a (non-empty) stable set $S_0 \subseteq S$ whose elements are smaller than the node i_n, i.e. $s \precsim i_n$ for all $s \in S_0$. The vicinal preorder implies that $\bigcup_{s \in S_0} N(s) \subseteq K_1$. Since the node i_n has not been matched by TGMA, we have $|\bigcup_{s \in S_0} N(s)| = |K_1|$ and therefore $\bigcup_{s \in S_0} N(s) = K_1$, i.e. all nodes in S_0 can only be and actually have been matched with a node in K_1, and all nodes in K_1 have been matched with a node in S_0. Hence we have $|S_0| = |K_1|$.

Moreover, there might be a stable set $S_1 \subseteq S$ with nodes that are greater than the node i_n, i.e. $s \succsim i_n$ for all $s \in S_0$. Since all nodes $k \in K_1$ have been matched with nodes $s \in S_0$, all nodes $k \in K_1$ must have been matched with some nodes $k \in K_0$. Finally, we note that those nodes $k \in K_0$ that have not been matched with nodes $s \in S_1$ must have been matched with other nodes $k \in K_0$.

In sum, the set of nodes N_G is partitioned as follows (the sum being written according to the preorder):

$N_G = S_0 + \{i_n\} + S_1 + K_0 + K_1$
with $S_0, K_0, K_1 \neq \varnothing$, $\bigcup_{s \in S_0} N(s) = K_1 = N(i_n)$,
and all nodes $s \in S_1$ matched with nodes $k \in K_0$.

(2) Let $k_0 \in K_0$ be one among the largest nodes in K_0, i.e. $k \precsim k_o$ for all $k \in K_0$, and thus $s \precsim k_0$ for all $s \in S_1$. By adding edges (if necessary at all), we construct a new graph G' with $N_{G'} := N_G$ such that $N(i) := N(k_0)$ for all $i \in S_1 \cup K_0 =: K_0'$ (i.e. K_0' induces a complete subgraph), with all other edges remaining unchanged.

Hence the set of nodes of the new graph can be written in the following way:

$N_{G'} = S_0 + \{i_n\} + K'_0 + K_1$
with $S_0, K'_0, K_1 \neq \varnothing$, and
$N(k_1) = N(k_2) \subsetneqq N(k_3)$ for all $k_1, k_2 \in K'_0$ and $k_3 \in K_1$.

Obviously, i_n would be an unmatched node also on G' after TGMA has terminated, and, since $E_G \subseteq E_{G'}$, a Hamiltonian Path Problem on G is infeasible if it is infeasible on G'. As the nodes $k_0 \in K'_0$ are adjacent only to all nodes $k \in K'_0 \cup K_1$, every Hamiltonian Path on G' will contain (one or more) subpaths

$$... - k_i - k_{i+1} - ... - k_{i+m} - k_{i+m+1} - ...$$
$$\text{with } k_i, k_{i+m+1} \in K_1, k_{i+1}, ..., k_{i+m} \in K'_0 \text{ and } m \geq 1 \ , \tag{16}$$

and, iff the Hamiltonian Path ends with a node $k_0 \in K'_0$, also a subpath

$$... - k_i - k_{i+1} - ... - k_{i+m}$$
$$\text{with } k_i \in K_1, k_{i+1}, ..., k_{i+m} \in K'_0 \text{ and } m \geq 1 \ . \tag{17}$$

(3) Consider the subgraph G'' of G' that is induced by
$N_{G''} = S_0 + \{i_n\} + \{k_0^*\} + K_1$ with some $k_0^* \in K'_0$.

We can directly transform a Hamiltonian Path on G' into a Hamiltonian Path on G'' by contracting all subpaths in (16) and (17) to the subpaths $... - k_i - k_{i+m+1} - ...$ and $... - k_i$, respectively, and adding the single node k_0^* to any among the contracted subpaths such that we get $... - k_i - k_0^* - k_{i+m+1} - ...$ and $... - k_i - k_0^*$ for the former subpaths, respectively. Therefore, the Hamiltonian Path Problem on G' is infeasible if it is infeasible on G''.

As $|S_0| = |K_1|$, we have $|N_{G''}| = 2|K_1| + 2$. Because it holds that $N(i) = K_1$ for all $|K_1| + 2$ nodes $i \in S_0 + \{i_n\} + \{k_0^*\}$, a maximum cardinality matching on G'' will have cardinality $|K_1| < \frac{|N_{G''}|}{2}$. Therefore, the Hamiltonian Path Problem is infeasible on G'', hence on G', and so on G. ■

Remark 54 *We have seen that the alternating path theorem for threshold graphs introduced in this section is a rather strong result that is closely related to the existence of Hamiltonian paths. It would be an interesting question for further research to explore whether this theorem can lead to an alternative characterization of threshold graphs.*

As mentioned above, we will now give a proof of the degree partition-based criterion that Mahadev and Peled (1994, Theorem 3.1) have derived from determining the longest cycle in threshold graphs.

Corollary 55 *(Degree partition criterion for Hamiltonian threshold graphs)*
If $G(N,E)$ is a threshold graph with degree partition

$$N_G = D_0 + D_1 + D_2 + ... + D_m,$$

then G is Hamiltonian if and only if there exists a (bipartite) matching

$$\text{from } S := \bigcup_{i=1}^{\lfloor \frac{m}{2} \rfloor} D_0 \text{ into } K := N_G - S$$

that has cardinality $|S|$, and
(i) for m even, M matches some node $j \in D_{\frac{m}{2}}$ with some node $k \in D_m$,
(ii) for m odd, M leaves some node $k \in D_m$ unmatched.

Proof. We define $n := |N_G|$ and assume that G has no isolated nodes (otherwise there is nothing to show).

a) As the corollary is about Hamiltonian cycles, let us prove the sufficiency of the conditions by adding a dominating node i_{n+1} to the graph first, which will not affect the size m of the degree partition. The bipartite matching from S into K covers all nodes in the maximal stable set if m is even, and in the complement of the maximal clique if m is odd, cf. Remark 41(3). Consequently, the nodes in $N_G \cup \{i_{n+1}\}$ that are unmatched up to now are adjacent to each other and can be matched into pairs. If $n+1$ is even, condition (i) of Theorem 53 applies, if $n+1$ is odd, one unmatched node remains, which certainly is an element of a maximal clique, so that condition (ii) applies.

b) Regarding the necessity of the conditions in the corollary, let us assume that the conditions of Theorem 53 hold after we have removed a dominating node $k \in D_m$. Then, in the original graph, all nodes in G are matched except the node k and, if $n-1$ is odd, except an additional node $j \in \bigcup_{i=\lceil \frac{m}{2} \rceil}^{m} D_i$. If m is odd, we have $j \notin S$ or $n-1$ is odd. In either case, our matching already contains a bipartite matching from S into K that has cardinality $|S|$ and leaves $k \in D_m$ unmatched. For even m, we have either $j \in D_{\frac{m}{2}}$ or $j \notin S$, or $n-1$ is odd. If $j \in D_{\frac{m}{2}}$, we can add the edge (j,k) to the matching and our matching will contain a bipartite matching in accordance with the corollary. If $j \notin S$ or $n-1$ is odd, we can easily fulfil the conditions of the corollary by matching the dominating node k with some arbitrary node from $D_{\frac{m}{2}}$ and discarding the latter node's mate. ■

5.3 Summary and a remark on the complexity of the MSSP

Our aim in this chapter was to take a first step towards analysing the MSSP by finding out more about matchings on threshold graphs. We presented a class of maximum cardinality matching algorithms on threshold graphs, introduced and discussed the concepts of even, matching-dominated and augmenting T-paths relative to a matching on a threshold graph, and found a new criterion for the existence of a Hamiltonian path on a threshold graph. In this section we

will briefly look at the meaning of the latter result in the light of the last section of chapter 3 on complexity theory

Corollary 56 *(Preliminary statement about the complexity of the MSSP)*

(1) *The twin-constrained Hamiltonian path problem on a threshold graph (i.e. the MSSP) is in* **NP**.

(2) *A special case of this problem, the Hamiltonian path problem on a threshold graph, is in* **P**.

(3) *A generalization of this problem, the twin-constrained Hamiltonian path problem, is* **NP***-complete.*

Proof. Statements (1) and (3) follow from Theorem 29; statement (2) is a consequence of Remark 30 in conjunction with Theorem 53. ■

We can conclude from this corollary that, for threshold graphs, the twin-constrained Hamiltonian path problem is at the frontier between two complexity classes, namely problems in **P** and **NP**-complete problems. At the moment, the question of whether or not the MSSP is solvable in polynomial time is open, and the answer to this question must be considered nontrivial. This means that as a reasonable strategy in the following we should continue to exploit the structure of the underlying threshold graph as much as possible, in the hope that we might find a polynomial-time approach to the MSSP. And yet, we should be open to the fact that we might have to conclude the endeavor of this thesis not with an exact algorithm, but instead with a polynomial-time heuristic. For the moment we can neither expect nor exclude the possibility of an exact polynomial-time algorithm for the MSSP. Therefore, it seems to be advisable to keep all options open and pursue a two-track policy.

6 Alternating cycles and maximum cardinality matchings on threshold graphs

There is a final type of path that we have to discuss here in relation to maximum cardinality matchings in order to be prepared for our analysis of the MSSP, namely what we call alternating T-cycles. These can be seen as a natural extension of our concept of alternating T-paths in so far as they feature, in contrast to augmenting T-paths, two end nodes that have been matched as one pair in the matching given, or, alternatively, as they can be considered as a special case of even T-paths (where the two end nodes coincide), or as a matching-dominated T-path where the end nodes are adjacent. Therefore, when discussing alternating T-cycles we will benefit a great deal from the results of the previous chapter.

We begin with a section that introduces the concept of alternating T-cycles and explains why they are an important tool for analysing all perfect matchings on threshold graphs. The succeeding section provides two (both necessary and sufficient) criteria for the existence of alternating T-cycles. Based on this, the third section sharpens one of the criteria by using the properties of the matching algorithm $TGMA_{\max}$ and, on this basis, presents a criterion for merging two alternating T-cycles. In the final section of this chapter, i.e. before we turn our attention to the MSSP from the next chapter on, we give a summary of what has been achieved by our discussion of threshold graphs in the preceding and the present chapter.

6.1 Definition and relevance of alternating T-cycles

We start with a definition of alternating T-cycles. Again, as it was the case with alternating T-paths, our definition directly refers to the subset of the matching that the cycle is build upon.

Definition 57 *(Alternating T-cycle)*

Let $G(N, E)$ be a graph, M a matching on G, and $T \subseteq M$. An alternating T-cycle relative to M is a cycle on G with an even number of edges such that every consecutive pair of edges in the cycle contains one edge from T and one edge that is not an element of the matching M, and all elements of T are edges of the cycle.

The following definition leads to a way of looking at the problem of analysing all perfect matchings on a threshold graph that will allow us to draw on the results of the preceding subsection.

Definition 58 *(Matching generator)*

Let $G(N, E)$ be a simple undirected graph and $\mathbf{M} \neq \varnothing$ the set of all perfect matchings on G.

(1) *For a given matching $M \in \mathbf{M}$, a pair $(g_M, \mathbf{S}_M)$ of a set $\mathbf{S}_M$ and a function*

$$g_M : \mathbf{S}_M \to \mathbf{M}$$

is called a matching generator relative to M, and M is called the basis of the matching generator.

(2) *For a given matching generator $(g_M, \mathbf{S}_M)$ relative to M and a particular subset $M^* \in \mathbf{M}$, an element $S \in \mathbf{S}_M$ is called an M^*-generator relative to M iff*

$$g_M(S) = M^*,$$

i.e. iff i "produces" the matching M^* under g_M.

(3) *A matching generator is said to be complete iff g_M is surjective, i.e. iff there exists an M^*-generator relative to M for all $M^* \in \mathbf{M}$.*

A matching generator relative to a certain perfect matching M allows us to begin with a specific matching and construct from there all possible perfect matchings on a given graph.

We proceed with a simple example of a matching generator.

Example 59 *Let us consider a complete graph $G(N, E)$ with $N_G = \{1, 2, 3, 4\}$, for which the set of all possible perfect matchings is*

$$\mathbf{M} = \{\{(1,2),(3,4)\},\{(1,3),(2,4)\},\{(1,4),(2,3)\}\}.$$

We choose the basis $M := \{(1,2),(3,4)\}$ and define a set $\mathbf{S}_M$ as follows:

$$\mathbf{S}_M := \{\quad \varnothing,\quad \{(1,2),(2,3),(3,4),(1,4)\},$$
$$\{(1,2),(2,4),(3,4),(1,3)\}\quad \}.$$

Furthermore, a function $g_M : \mathbf{S}_M \to \mathbf{M}$ be defined by

$$g_M(S) := S \backslash M + M \backslash (S \cap M) \text{ for all } S \in \mathbf{S}_M.$$

Then we have

$$g_M(\varnothing) = \varnothing + M,$$
$$g_M(\{(1,2),(2,3),(3,4),(1,4)\}) = \{(2,3),(1,4)\} + \varnothing,$$
$$g_M(\{(1,2),(2,4),(3,4),(1,3)\}) = \{(2,4),(1,3)\} + \varnothing,$$

which implies that g_M is surjective and hence $(g_M, \mathbf{S}_M)$ a complete matching generator relative to M.

The following proposition generalizes the preceding example by using the concept of alternating T-cycles and establishes the relevance for alternating T-cycles for analysing all perfect matchings on a graph.

Proposition 60 *(Alternating T-cycles as matching generators)*

Let $G(N,E)$ be a simple undirected graph, $M \in \mathbf{M} \neq \varnothing$ a basis from the set of all perfect matchings on G. Further, let $\mathbf{C}_M$ be the set of families that contains the family $(C_*) := (\varnothing)$, i.e. the family the only member of which is the empty set, and all disjunct families $(C_i)_{i\in I}$ of cycles on G for which there exists a disjunct family $(T_i)_{i\in I}$ of subsets $T_i \subseteq M$ such that C_i is an alternating T_i-cycle relative to M for all $i \in I$. Then $(g_M, \mathbf{C}_M)$, by virtue of

$$T_* := \varnothing \text{ and}$$
$$g_M : \mathbf{C}_M \to \mathbf{M}$$
$$\text{with } (C_i)_{i\in I} \mapsto g_M((C_i)_{i\in I}) := \sum_{i\in I}(C_i \backslash T_i) + M\backslash(\sum_{i\in I} T_i)$$
$$\text{for all } (C_i)_{i\in I} \in \mathbf{C}_M,$$

is a complete matching generator relative to M.

Proof. We have to show that *a*) for all families $(C_i)_{i\in I} \in \mathbf{C}_M$ the image

$$g_M((T_i)_{i\in I}) = \sum_{i\in I}(C_i\backslash T_i) + M\backslash(\sum_{i\in I} T_i)$$

is a perfect matching on G and *b*) there exists an M'-generator relative to M for all $M' \in M$.

a) Trivially, the image of $(C_*) = (\varnothing)$ with $T_* = \varnothing$ is a perfect matching.

For all other families $(C_i)_{i\in I}$ we consider the two disjunct sets that the image $g_M((C_i)_{i\in I})$ consists of separately. The second term

$$M\backslash(\sum_{i\in I} T_i) = M\backslash(\sum_{i\in I} C_i)$$

is the set of all edges in the matching M that are not incident to nodes connected via the alternating cycles C_i and is certainly a perfect matching on the set of these nodes.

The subgraph given by the first term

$$\sum_{i\in I}(C_i\backslash T_i) = (\sum_{i\in I} C_i)\backslash M$$

consists of alternating cycles from which every consecutive edge is removed, i.e. the term describes a subgraph that is a perfect matching on the set of nodes to which the edges from the alternating cycles are incident. As the image $g_M((C_i)_{i\in I})$ contains edges such that all nodes are incident to some edges, it is a perfect matching on G.

b) Trivially, $(C_*) = (\varnothing)$ is an M-generator relative to M.

For all other $M' \in M$, $M' \neq M$, we consider the set $(M \cup M')\backslash(M \cap M')$, which consists of edges incident to nodes each of which is an endpoint of one edge from M and one edge from M'. Hence, the set $(M \cup M')\backslash(M \cap M')$ consists of disjunct cycles C_i such that

$$\sum_{i\in I} C_i = (M \cup M')\backslash(M \cap M') \ . \tag{18}$$

If we define a family $(T_i)_{i\in I}$ of sets

$$T_i := C_i\backslash M' \text{ for all } i \in I \ , \tag{19}$$

the family $(C_i)_{i\in I}$ consists of alternating T_i-cycles relative to M. Hence $(C_i)_{i\in I} \in \mathbf{C}_M$, and due to (18) and (19) we have

$$\begin{aligned} g_M((C_i)_{i\in I}) &= \sum_{i\in I}(C_i\backslash T_i) + M\backslash(\sum_{i\in I} T_i) \\ &= \sum_{i\in I}(C_i\backslash(C_i\backslash M')) + M\backslash(\sum_{i\in I}(C_i\backslash M') \\ &= (\sum_{i\in I} C_i)\backslash((\sum_{i\in I} C_i)\backslash M') + M\backslash((\sum_{i\in I} C_i)\backslash M') \\ &= ((M\cup M')\backslash(M\cap M'))\backslash(((M\cup M')\backslash(M\cap M'))\backslash M') \\ &\quad + M\backslash(((M\cup M')\backslash(M\cap M'))\backslash M') \\ &= ((M\cup M')\backslash(M\cap M'))\backslash(M\backslash M') + M\backslash(M\backslash M') \\ &= M'\backslash M + (M\cap M') = M', \end{aligned}$$

i.e. the family $(C_i)_{i\in I}$ is an M'-generator relative to M. ■

The consequence of the preceding proposition is that, when studying perfect matchings on graphs, we can start with one single matching M and then operate on the set of alternating T-cycles for analysing all other matchings on our graph. In the case of threshold graphs this means in particular that we can start with a convenient implementation of $TGMA$ (with $TGMA_{\max}$, for example) and develop our theory by analysing alternating T-cycles relative to the matching M we obtained from our implementation of $TGMA$. This is exactly the path we will follow in the next two sections.

6.2 Criteria for the existence of alternating T-cycles

Having introduced the concept of alternating T-cycles and demonstrated its relevance in the previous section, we will now develop criteria for the existence of alternating T-cycles on threshold graphs. The criteria for the existence of certain *paths* on threshold graphs, namely Hamiltonian paths and different types of alternating paths that we have presented in the preceding chapter, all refer to the characteristics of a certain given matching M. This begs the question of whether it is possible to have also such a criterion of the existence of alternating T-cycles. On the basis of our Theorem 53 for Hamiltonian paths, we can establish such a criterion indeed

We will proceed in three steps. As a preparation for what follows, we will start with a basic general observation on the relation of two perfect matchings on the same graph, which is expressed in the succeeding lemma. On this basis and by means of Theorem 53, we will show in a second step that in the case of threshold graphs, alternating T-cycles and Hamiltonian cycles on the corresponding subgraph are subject to similar conditions. Finally, we derive from there a statement that characterizes the matching M relative to which we would like to construct an alternating T-cycle.

Lemma 61 *Let $G(N,E)$ be a simple undirected graph, $T, T' \subseteq E_G$ perfect matchings, and $(i_0,j_0) \in E_G$ an edge such that $(i_0,j_0) \in T'$, but $(i_0,j_0) \notin T$. Then there exists an alternating $(C\backslash T')$-cycle $C \subseteq T \cup T'$ relative to T with $(i_0,j_0) \in C$.*

Proof. We consider all nodes in N_G that have different mates with respect to the matchings $T \subseteq M$ and T', i.e. the set

$$N_{T \oplus T'} := \{i \in N_G : (i,j) \in T \text{ and } (i,j) \notin T' \text{ for some } j \in N_G\}.$$

As every node of the subgraph $G'(N,E)$ given by

$$N_{G'} := N_{T \oplus T'} \text{ and } E_{G'} := T \oplus T'$$

(where $T \oplus T'$ denotes the symmetric difference of T and T') has exactly two neighbours, namely its mate under the matching T and its mate under T', the components of G' are cycles, i.e. G' can be fully described by a family of (alternating) cycles $(C_l)_{l \in L}$ with $E_l \subseteq E_{G'}$ for all $l \in L$ such that

$$\sum_{l \in L} C_l = T \oplus T'$$

and

$$\sum \{i \in N_G : (i,j) \in C_l \text{ for some } l \in L \text{ and } j \in N_G\} = N_{T \oplus T'}.$$

Because of $(i_0, j_0) \notin T$, but $(i_0, j_0) \in T'$, the edge $(i_0, j_0) \in E_{G'}$ is an element of one of these cycles. ■

We can now make a first statement about the relation between matchings and alternating T-paths on a threshold graphs. Basically, the following theorem means that we will find an alternating T-cycle if and only if the subgraph induced by T is so "rich" in edges that we can "afford" to construct a perfect matching on the subgraph in which two nodes with a rather high degree have been singled out to be each others' mates.

Theorem 62 *(Matching criterion for alternating T-cycles on threshold graphs)*
Let $G(N,E)$ be a threshold graph, M a matching on G, and $T \subseteq M$. Let

$$U := \{i \in N_G : (i,j) \in T \text{ for some } j \in N_G\}$$

denote the set of all nodes incident to edges in T. Then there exists an alternating T-cycle relative to M if and only if there exists a perfect matching T' on the subgraph induced by U such that a dominating node in U is matched with an element of a maximal clique in U.

Proof. $\Rightarrow$: Obviously, the existence of a perfect matching on the subgraph induced by U is a necessary condition for the existence of an alternating T-cycle. As for the necessity of the second part of the condition, let $i_0 \in U$ be a dominating node in U. The existence of an alternating T-cycle trivially implies that there is a Hamiltonian path on the odd set $U - \{i_0\}$. Hence the node remaining from a matching T' according to Theorem 53 is a member of a maximal clique in U and can be matched with i_0.

$\Leftarrow$: Let $i_0 \in U$ be a dominating node in U with a mate $j_0 \in U$ that is a member of a maximal clique in U, and T' a perfect matching on U with $(i_0, j_0) \in T'$.

$Case(1) : (i_0, j_0) \in T$. We consider all other pairs $(i_k, j_k) \in T$ with $1 \leq k \leq |T| - 1$, assume without loss of generality that

$$j_k \precsim i_k \text{ for } 1 \leq k \leq |T| - 1 \ , \tag{20}$$

and renumber the nodes such that

$$j_{|T|-1} \precsim j_{|T|-2} \precsim ... \preceq j_2 \precsim j_1,$$

which yields $i_k \in N(j_k) \subseteq N[j_{k-1}]$ for $2 \leq k \leq |T| - 1$. Hence

$$i_1 - j_1 - i_2 - j_2 - ... - i_{|T|-2} - j_{|T|-2} - i_{|T|-1} - j_{|T|-1}$$

is a matching-dominated $(T - \{(i_0, j_0)\})$-path relative to M with the node i_1 being in a maximal clique in U because of (20). Thus we have $(j_0, i_1) \in E_G$ and, since i_0 is a dominating node in U, also $(j_{|T|-1}, i_0) \in E_G$, i.e.

$$i_0 - j_0 - i_1 - j_1 - i_2 - j_2 - ... - i_{|T|-2} - j_{|T|-2} - i_{|T|-1} - j_{|T|-1} - i_0$$

is an alternating T-cycle relative to M.

$Case(2) : (i_0, j_0) \notin T$. Applying Lemma 61 to the subgraph induced by U, leads to a cycle $C \subseteq T \oplus T'$ with $(i_0, j_0) \in C$. By deleting the edge (i_0, j_0) from C, we obtain a matching dominated $(C \backslash T')$-path relative to $M \supseteq C \backslash T'$ with the endnodes i_0 and j_0.

Now we consider all nodes in U that are not incident to this path, namely the node set

$$U - \{i \in U : (i, j) \in T \backslash C \text{ for some } j \in N_G\},$$

for which the edge set

$$T \backslash C = T - C \backslash T' \subseteq M$$

is a perfect matching. On this basis we can construct a matching dominated $(T \backslash C)$-path in the same fashion as in $Case(1)$ above, which has a member of a maximal clique in

$$\{i \in N_G : (i, j) \in T \backslash C \text{ for some } j \in N_G\}$$

(and, again because of (20), also in U) as one of its endpoints. By connecting this endpoint to j_0 and connecting the other endpoint to the dominating node i_0, we obtain an alternating $(T \backslash C + C \backslash T')$-cycle relative to M. ■

For the construction of an alternating T-cycle, the preceding theorem required that we find a certain new matching T' on the subgraph induced by the set $T \subseteq M$. We will now provide a statement that gives a characterization of alternating T-paths by directly drawing on some property of the subset T.

Theorem 63 *(Path criterion for alternating T-cycles on threshold graphs)*

Let $G(N, E)$ be a threshold graph, M a matching on G, and $T \subseteq M$. Let

$$U := \{i \in N_G : (i, j) \in T \text{ for some } j \in N_G\}$$

denote the set of all nodes incident to edges in T. Then there exists an alternating T-cycle relative to M if and only if there exists a subset $P \subseteq T$ and a matching-dominated P-path relative to M the endnodes of which are a dominating node in U and a node in a maximal clique of U.

Proof. We will show that the claim of this theorem is equivalent to the preceding theorem. In the following, i_0 is a dominating node in U and j_0 a member of a maximal clique in U.

$\Rightarrow$: Let T' be a perfect matching on U with $(i_0, j_0) \in T'$. If $(i_0, j_0) \in T$, we can set $P := \{(i_0, j_0)\}$ and are done. If $(i_0, j_0) \notin T$, we know from Lemma 34 that there exists an

alternating $(C \backslash T')$-cycle relative to T for some set $C \subseteq T \cup T'$ with $(i_0, j_0) \in C$, and define $P := C - \{(i_0, j_0)\}$.

$\Leftarrow$: Let i_0 and j_0 be the two endpoints of the matching dominated P-path relative to M. If we denote this path by Q,

$$T' := (Q - P) + \{(i_0, j_0)\} + (T - P)$$

is a perfect matching on the subgraph induced by U, which fulfils the condition of Theorem 35. ■

Remark 64 *Note that the theorem refers to a dominating node and a member of the maximal clique in U. Of course, if this node set contains a dominating node and a member of the maximal clique in N_G, these nodes maintain the same properties also in U. The converse, however, is not necessarily the case.*

6.3 Alternating T-cycles and the case of greedy matchings

The path criterion provided above gives a characterization of alternating T-cycles on the basis of some property of a certain subset $P \subseteq T$. From a combinatorial point of view, such a result is rather unsatisfying because it requires us to check all possible subsets P to find out if there exists one with the desired property. This suggests the idea to tighten this theorem, possibly in a way that allows us to impose the same conditions that are relevant for the subset P on the set T *as a whole*. For reaching this aim, it is inevitable that the matching M from which we proceed is richer in structure. This can be achieved if we assume that the matching is greedy, i.e. has been obtained from the version $TGMA_{\max}$ of our Threshold Graph Matching Algorithm.

Theorem 65 *(Strong path criterion for alternating T-cycles on threshold graphs)*

Let $G(N, E)$ be a threshold graph, M a matching on G that has been obtained from $TGMA_{\max}$ and $T \subseteq M$. Let

$$U := \{i \in N_G : (i, j) \in T \text{ for some } j \in N_G\}$$

the set of all nodes incident to edges in T. Then there exists an alternating T-cycle relative to M if and only if there exists a matching-dominated T-path relative to M the endnodes of which are a dominating node of U and a node in a maximal clique of U.

Proof. The sufficiency of the condition is trivial as we can simply connect the endnodes of the matching-dominated T-path. Regarding the necessity of the condition let there be a matching-dominated P-path relative to M with endnodes i_0 and j_0 according to Theorem 63. We will construct, by inserting the edges in $T - P$ into the matching-dominated P-path, a matching-dominated T-path that fulfils the conditions of our theorem. The proof proceeds in three steps.

STEP 1: Assume that it is possible to rearrange all edges $(i_k, j_k) \in P$ such that we arrive at a matching-dominated P-path relative to M that has the properties of Theorem 63 and is represented by

$$i_0 - j_0 - i_1 - j_1 - i_2 - j_2 - ... - i_{|P|-2} - j_{|P|-2} - i_{|P|-1} - j_{|P|-1} \tag{21}$$

such that

$$i_k \gtrsim j_k \text{ for } 0 \leq k \leq \frac{|P|}{2} - 1 \tag{22}$$

and

$$i_0 \gtrsim i_1 \gtrsim ... \gtrsim i_{\frac{|P|}{2}-2} \gtrsim i_{\frac{|P|}{2}-1} \ . \tag{23}$$

Now take any edge $(i, j) \in T - P$. We assume $i \succsim j$. Consider all edges in P and choose, if possible, a k^* such that $i_{k*} \succnsim i \succsim i_{k*+1}$. Because of $i \succsim i_{k*+1}$, we have $j_{k*} \in N(i_{k*+1}) \subseteq N(i)$, i.e. $(j_{k*}, i) \in E_G$. Moreover, we have $j \succsim j_{k*}$ due to $i_{k*} \succnsim i$ and the degree property of $TGMA_{\max}$ (Proposition 51), which leads to $(j, i_{k*+1}) \in E_G$. Hence, we can insert (i, j) into the matching-dominated P-path between the edges (i_{k*}, j_{k*}) and (i_{k*+1}, j_{k*+1}). If it is not possible to find such a k^*, the node i is a dominating node in U or we have $i_{|P|-1} \succnsim i$. In the former case we add the edge (i, j) at the beginning of the path (21), which is possible because the path starts with a dominating node. In the latter case, we add (i, j) at the end of the path (21), which is possible as $j_{|P|-1}$ is a member of a maximal clique in U and $i \succsim j$. Proposition 51 of $TGMA_{\max}$ implies that $j \succnsim j_{|P|-1}$, i.e. also the new endnode j is a member of a maximal clique in U. It all three cases considered, we obtain at a matching-dominated $(P + \{(i, j)\})$-path that preserves the properties of the matching-dominated P-path regarding the order of nodes and the particular end nodes. By induction over $P - T$, we finally arrive at a matching-dominated T-path with the properties required.

STEP 2: It remains to show that it is indeed possible to construct the path (21), which we will do in the following two steps. In this step, we will prove that for all edges $(i, j) \in P$ (assume in the following $i \gtrsim j$) to which no dominating node in U is incident, there exists an edge $(k, l) \in P$ (assume in the following $k \succsim l$) such that $k \succnsim i$ and l is adjacent to i.

We proceed from assuming that the opposite is true, i.e. that there exists an edge (i^*, j^*), with i^* not being a dominating node in U, such that $(l, i^*) \notin E_G$ for all edges $(k, l) \in P$ with $k \succnsim i^*$. Because of the vicinal preorder, this implies

$$(l, i) \notin E_G \text{ for all } (i, j), (k, l) \in P \text{ with } k \succnsim i^* \text{ and } i^* \succsim i,$$

which yields

$$j \succnsim l \text{ for all } (i, j), (k, l) \in P \text{ with } k \succnsim i^* \text{ and } i^* \succsim i.$$

Therefore, the edge $(i, j) \in P$ induces a partition $P = P_1 + P_2$ of the set of edges P such that

$$(i^*, j^*) \in P_1 \ ,$$

$$k \succsim i \succsim j \succsim l \text{ for all } (i,j) \in P_1, (k,l) \in P_2$$
$$\text{with } i \succsim j \text{ and } k \succsim l \ , \tag{24}$$

and

$$(l,i) \notin E_G \text{ and } (l,j) \notin E_G \text{ for all } (i,j) \in P_1, (k,l) \in P_2$$
$$\text{with } i \succsim j \text{ and } k \succsim l \ . \tag{25}$$

(24) implies that P_2 contains all edges from P to which the dominated nodes in U are incident. Hence, the matching-dominated P-path, which has to begin with a dominated node in U, starts with an edge from P_2. Because of (25), all following edges from P in the matching dominated P-path must be from P_2. This contradicts the fact that the matching dominated P-path must also contain all elements of P_1, in particular $(i^*, j^*) \in P_1$. Consequently, our initial assumption that that there exists an edge $(i^*, j^*) \in P$, with i^* not being a dominating node in U, such that $(l, i^*) \notin E_G$ for all edges $(k,l) \in P$ with $k \succsim i^*$ has been proven wrong.

STEP 3: We can now construct the path (21) in a straight forward manner. Arrange all edges $(i_k, j_k) \in P$ such that (22) and (23) hold. Furthermore, due to the degree property of $TGMA_{\max}$ (Proposition 51), it is possible to sort the nodes j_k for $0 \leq k \leq \frac{|P|}{2} - 1$ in the order

$$j_{\frac{|P|}{2}-1} \succsim j_{\frac{|P|}{2}-2} \succsim \ldots \succsim j_1 \succsim j_0 \tag{26}$$

such that this remains consistent with (22) and (23). As a consequence,

$$i_{\frac{|P|}{2}-1} - j_{\frac{|P|}{2}-1}$$

is a path on G that ends with a node in a maximal clique of U.

Assume now we have constructed a path

$$i_{k*} - j_{k*} - i_{k*+1} - j_{k*+1} - \ldots - i_{\frac{|P|}{2}-2} - j_{\frac{|P|}{2}-2} - i_{\frac{|P|}{2}-1} - j_{\frac{|P|}{2}-1}$$

on G for some $k^* \in \{1, \ldots, \frac{|P|}{2} - 2, \frac{|P|}{2} - 1\}$. If the node i_{k*} is a dominating node in U, all other nodes in $\{i_0, i_1, \ldots, i_{k*-2}, i_{k*-1}\}$ are also dominating nodes because of (23), which implies that (21) is a feasible path on G. If i_{k*} is not a dominating node in U, there exists an edge $(i_k, j_k) \in P$ with $k < k^*$ such that $(j_k, i_{k*}) \in E_G$ according to *STEP* 2 of this proof and (23). Property (26) yields $(j_{k*-1}, i_{k*}) \in E_G$, i.e.

$$i_{k*-1} - j_{k*-1} - i_{k*} - j_{k*} - i_{k*+1} - j_{k*+1} - \ldots - i_{\frac{|P|}{2}-1} - j_{\frac{|P|}{2}-1}$$

is a path on G. Induction over $\{\frac{|P|}{2} - 1, \frac{|P|}{2} - 2, \ldots, 1, 0\}$ leads to (21), which starts with a dominating node by virtue of (22) and (23). ■

The preceding proof made use of the assumption that M is a greedy matching in two of its three steps. The following example illustrates on the basis of the most simple case possible that this condition is necessary indeed.

Example 66 *Consider the threshold graph $G(N, E)$ given by*

$$N_G := \{1, 2, 3, 4, 5, 6, 7, 8\}$$

and

$$\begin{aligned} E_G := \{&\{1,7\},\{1,8\},\{2,6\},\{2,7\},\{2,8\},\\ &\{3,4\},\{3,5\},\{3,6\},\{3,7\},\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\\ &\{5,6\},\{5,7\},\{5,8\},\{6,7\},\{6,8\},\{7,8\}\}, \end{aligned}$$

for which

$$1, 2, 3, 4, 5, 6, 7, 8$$

clearly is an ordering of nodes with non-decreasing degrees,

$$K_{\max} = \{3, 4, 5, 6, 7, 8\}$$

is the (only) maximal clique on G, and

$$D = \{7, 8\}$$

is the set of dominating nodes. Running $TGMA_{\max}$ on G yields

$$M_{greedy} = \{\{1,8\},\{2,7\},\{3,6\},\{4,5\}\},$$

while $TGMA_{\min}$ constructs the matching

$$M_{\mathrm{mod}\,est} = \{\{1,7\},\{2,6\},\{3,4\},\{5,8\}\}.$$

We focus on the cases $T_{greedy} := M_{greedy}$ and $T_{\mathrm{mod}\,est} := M_{\mathrm{mod}\,est}$ and examine all alternating T_{greedy}-cycles and $T_{\mathrm{mod}\,est}$-cycles, respectively.

(a) In the case of a greedy matching, the only alternating T_{greedy}-cycles are

$$8-1-7-2-6-3-4-5-8 \quad \textit{and}$$
$$8-1-7-2-6-3-5-4-8,$$

which contain the matching-dominated T_{greedy}-paths

$$8-1-7-2-6-3-4-5 \quad \textit{and}$$
$$8-1-7-2-6-3-5-4,$$

respectively. In line with the preceding theorem, both paths start with dominating nodes and end with members of $K_{\max}$. (Note that there exist other matching-dominated T_{greedy}-paths relative to M_{greedy}. These however do not fulfil the properties required by the preceding theorem.)

(b) In contrast to this, the modest matching allows for the alternating $T_{\mathrm{mod}\,est}$-cycles

$$7-1-8-5-3-4-6-2-7 \quad \textit{and}$$
$$7-1-8-5-4-3-6-2-7.$$

One can see immediately that on G there exist only the following four matching-dominated $T_{\mathrm{mod}\,est}$-paths relative to $M_{\mathrm{mod}\,est}$ that start with a dominating node in N_G:

$$7-1-8-5-3-4-6-2, \qquad 8-5-3-4-6-2-7-1$$
$$7-1-8-5-4-3-6-2, \qquad 8-5-3-4-6-2-7-1,$$

none of which ends with a member of $K_{\max}$. Hence, the strong path criterion for the existence of an alternating T-cycle given by the preceding theorem cannot be applied for a matching obtained from $TGMA_{\min}$.

However, note that, in line with the general path criterion for the existence of alternating T-cycles (Theorem 63), there exists a subset

$$P := \{\{7,1\},\{8,5\},\{3,4\}\} \subseteq T_{\mathrm{mod}\,est},$$

and a matching-dominated alternating P-path

$$7-1-8-5-3-4$$

relative to $M_{\mathrm{mod}\,est}$ that starts with a dominating node and ends with member of $K_{\max}$, which ensures the existence of an alternating $T_{\mathrm{mod}\,est}$-cycle relative to $M_{\mathrm{mod}\,est}$ according to Theorem 63.

The proof of the preceding theorem justifies the following definition.

Definition 67 *(Sorted alternating T-cycles)*

Let $G(N,E)$ be a threshold graph, M a matching on G and $T \subseteq M$. An alternating T-cycle relative to M is called sorted iff deleting one of its edges yields a matching-dominated T-path for which properties (21), (22), (23), and (26) hold.

We will now derive three direct corollaries about sorted alternating T-cycles from the preceding theorem, which will turn out to be useful in the following chapters. The assumption that the underlying matching M has been generated by $TGMA_{\max}$ must be considered necessary here again.

Corollary 68 *(Existence of sorted alternating T-cycles)*

Let $G(N,E)$ be a threshold graph, M a matching on G that has been obtained from $TGMA_{\max}$ and $T \subseteq M$. There exists an alternating T-cycle relative to M if and only if there exists a sorted alternating T-cycle relative to M.

Proof. If there exists an alternating T-cycle relative to M, there exists a matching-dominated T-path relative to M the endnodes of which are a dominating node of U and a node in a maximal clique of U according to Theorem 65. We sort the path $P := T$ in the fashion of $STEP$ 2 and 3 of the preceding theorem and connect its end nodes. ■

The following concept is intended only to simplify our manner of expression later.

Definition 69 *(Canonical alternating T-cycle)*

Let $G(N,E)$ be a simple undirected graph and $M \subseteq E$ a matching. If, for a given set of edges $T \subseteq M$ written in the form $T = \{(i_1,j_1),(i_2,j_2),...,(i_k,j_k)\}$, there exists an alternating T-cycle relative to M of the form

$$i_1 - j_1 - i_2 - j_2 - ... - i_k - j_k - i_1,$$

we will call this cycle the canonical alternating T-cycle.

Our second corollary enables us to construct alternating cycles from certain subsets of a given set $T \subseteq M$.

Corollary 70 *(Existence of sorted canonical alternating cycles on subsets)*

Let $G\,(N, E)$ be a threshold graph, M a matching on G that has been obtained from $TGMA_{\max}$, and $T \subseteq M$ such that there exists an alternating T-cycle relative to M. Then there exists a sorted canonical alternating T-cycle according to $(21), (22), (23), (26)$ *and for all sets*

$$T_{kl} := \{(i_k, j_k), (i_{k+1}, j_{k+1}), ..., (i_{l-1}, j_{l-1}), (i_l, j_l)\} \subseteq T$$
$$\text{with } 0 \leq k < l \leq |T| - 1$$

there exists the sorted canonical alternating T_{kl}-cycle relative to M.

Proof. We know from the proof of the preceding corollary that any ordering of the nodes and edges from T according to $(21), (22), (23), (26)$ leads to a sorted canonical alternating T-cycle of the form

$$i_0 - j_0 - i_1 - j_1 - ... - i_{|T|-2} - j_{|T|-2} - i_{|T|-1} - j_{|T|-1} - i_0$$

with

$$T = \{(i_0, j_0), (i_1, j_1), ..., (i_{|T|-1}, j_{|T|-1})\}.$$

Hence

$$i_k - j_k - i_{k+1} - j_{k+1} - ... - i_{l-1} - j_{l-1} - i_l - j_l$$

is a matching dominated T_{kl}-path relative to M the end nodes of which can be connected because of the vicinal preorder of G and due to (26), which yield

$$i_k \in N(j_k) \subseteq N(j_l). \quad \blacksquare$$

Finally, we note a corollary of the preceding theorem that makes a statement about the condition under which we can "merge" two alternating T-cycles relative to the same matching.

Corollary 71 *(Sorted alternating $(T_1 \cup T_2)$-cycles on threshold graphs)*

Let $G\,(N, E)$ be a threshold graph, M a matching on G that has been obtained from $TGMA_{\max}$, and $T_0, T_1 \subseteq M$ two subsets of edges such that there exists an alternating T_0-cycle and an alternating T_1-cycle relative to M. Then there exists an alternating $(T_0 \cup T_1)$-cycle relative to M if and only if for $k = 0, 1$ and

$$U_k := \{i \in N_G : (i, j) \in T_k \text{ for some } j \in N_G\}$$

the dominating nodes in U_k are adjacent to the members of a maximal clique in U_{1-k}.

Remark 72 *Note that we trivially have at least for $k = 0$ OR $k = 1$ the case that the dominating nodes in U_k are adjacent to the members of a maximal clique in U_{1-k} because the vicinal preorder ensures that at least one of the sets U_k contains a dominating node in $U_0 \cup U_1$. The condition of the corollary states that we need this property for both $k = 0$ and $k = 1$.*

Proof. $\Leftarrow$: Let us assume without loss of generality that the dominating nodes in U_0 have at least the same degree as the dominating nodes in U_1. We construct the sorted canonical alternating T_k-cycles of the sorted sets T_k and insert the edges from $T_1 \backslash T_0$ in the fashion of $STEP$ 1 in the proof of Theorem 65, however such that we proceed in the sorted order of edges given in $T_1 \backslash T_0$. If necessary, we will always be able to add an edge at the beginning of the path (21) because the dominating nodes in U_0 are dominating also in U_1. Moreover, if necessary, we will always be able to add an edge at the end of the path (21) because the dominating nodes in U_1 are adjacent to the members of a maximal clique in U_0. If the path (21) finally ends with a node in U_0, we can connect the end nodes because the canonical alternating T_0-cycle exists. Otherwise, we know that the end nodes of (21) are adjacent because the dominating nodes in U_0 are dominating also in U_1.

$\Rightarrow$: For demonstrating the necessity of the condition, we assume that it is not fulfilled. Without loss of generality, this is equivalent to assuming that (a) the dominating nodes in U_0 have at least the same degree as the dominating nodes in U_1 and (b) the dominating nodes in U_1 are *not* adjacent to the members of a maximal clique in U_0. We will sort the set $T_0 \cup T_1$ according to (21), (22), (23), (26), which is possible because the matching M is greedy. Because of (a), we can assume the sorted set $T_0 \cup T_1$ to start with an edge from T_0. Due to (b), the first edge from T_0 that appears in the set $T_0 \cup T_1$ will follow after the last edge from T_1. Moreover, assumption (b) implies that we cannot connect the larger node incident to the first edge from T_1 to the smaller node incident to the last edge from T_0. Consequently, there exists no sorted canonical alternating $(T_0 \cup T_1)$-cycle, and thus no alternating $(T_0 \cup T_1)$-cycle relative to M. ■

6.4 Summary of our results about maximum cardinality matchings on threshold graphs

We have started the preceding chapter with a simple matching algorithm for threshold graphs, which, in its different implementations, can provide all possible matchings on a given threshold graph. Using this algorithm, we analyzed the way in which we can construct alternating paths on the basis of a certain matching and could show that the matching algorithm always leads to maximum cardinality matchings. Building on this, we were able to present a direct proof for a new criterion for Hamiltonian paths on threshold graphs, which led to a new proof of a well-known theorem by Mahadev and Peled. Based on our new criterion, we derived several necessary and sufficient conditions for the existence of alternating cycles relative to a given matching, and finally arrived at a rather tight characterization in the case of alternating cycles

relative to a matching generated by $TGMA_{\max}$. In particular, it has been shown that the existence of alternating T-cycles relative to a given matching M is equivalent to the existence of a matching-dominated T-path with certain properties on the subset of nodes incident to edges in T. In view of the concept of matching generators as introduced at the beginning of the present chapter, we can say that studying alternating cycles has led our analysis on the relation of matchings and different types of paths back to the question that was our initial point of departure − in the sense that the concept of alternating cycles provides an answer to the question of all possible (perfect) matchings on a threshold graph.

By proceeding in this way, we have developed a useful method for analysing perfect matchings on threshold graphs. At the beginning of the previous chapter we had to derive their properties from possible outcomes of the matching algorithm. Instead, we can now proceed from a single matching M generated by the computationally most efficient matching algorithm, i.e. $TGMA_{\max}$. The concept of the matching generator then enables us to derive statements about all possible other matchings by examining the set of all generators relative to M, and, in doing so, we are able to draw on the convenient properties of the matchings generated by $TGMA_{\max}$. As we will see in a later chapter, this can, depending on the circumstances, even imply that we can restrict our attention to the subset of $\mathbf{C}_M$ that consists only of disjunct families of *sorted* alternating T_i-cycles relative to a greedy matching gained by $TGMA_{\max}$, i.e. those types of cycles for which we have developed the sharpest characterization.

7 Constructing twin-constrained Hamiltonian paths on threshold graphs

Having analyzed in detail the properties of matchings and various type of paths on threshold graphs, we now have all necessary prerequisites and will turn our attention to the Minimum Score Separation Problem, i.e. the problem of finding a *twin-constrained* Hamiltonian path on a threshold graph. The aim of this chapter is threefold: first, to gain a basic understanding of the general structure of the twin-constrained Hamiltonian path problem on threshold graphs and of how it is related to matchings; second, to solve the MSSP for a number of specific cases; and third, by doing so, to prepare ourselves for finding a general solution of the MSSP. We will begin by making some general considerations on twin-constrained Hamiltonian paths in the first section. The succeeding section introduces the concept of the "twin-induced structure" of a matching and analyses both the case in which we have a perfect matching on the threshold graph and the case in which we have at least 4 unmatched nodes. The third section addresses the case of 2 unmatched nodes, three sub-cases of which we will analyse more in detail in further sections: "structure-preserving solutions" and two types of "non structure-preserving solutions", namely "path-split solutions" and "cycle-split solutions". The final section sums up the results of this chapter by presenting a heuristic that solves the MSSP for a large percentage of instances.

Unfortunately, the convenient structure of a simple Hamiltonian Path Problem on a threshold graph does not translate directly into a solution to the MSSP. This is due to the *twin node* condition of the MSSP, which partly destroys the structure that has been discussed in the preceding section. In the following, we will proceed from our results for threshold graphs and analyse the relation of matchings and paths with respect to the structure given by the MSSP. The aspects to be discussed will establish the general approach for developing a heuristic algorithm for the MSSP afterwards.

7.1 General considerations, modified matchings

We will start with some general considerations on the MSSP, i.e. on recognizing twin-constrained Hamiltonian threshold graphs.

In the remainder of this chapter, let $G(N, E)$ be a threshold graph with the node set $N_G = \{1, 2, ..., 2n\}$ and an edge set E_G. Given a (bijective) twin-node function $b : N_G \to N_G$, the graph $G'(N, E)$ with $N_{G'} := N_G$ and

$$E_{G'} := E_G \cup \{(i, j) \in N_{G'} \times N_{G'} : j = b(i)\} \tag{27}$$

then constitutes a MSSP, which is feasible if and only if there exists a Hamiltonian path on G' that is of the form

$$i_1 - b(i_1) - i_2 - b(i_2) - ... - i_n - b(i_n) \ , \tag{28}$$

i.e. if and only if there exists on G a twin-constrained Hamiltonian path with respect to b

Let us consider what distinguishes the problem of finding out whether there exists a twin-constrained Hamiltonian path from the problem of determining whether there exists a Hamiltonian path on G (as discussed in chapter 5.2).

As we have made no further assumptions on the twin-node function b, the twin-node condition (28) is in general not compatible with the specific structure of a threshold graph: the fact that condition (28) requires every second edge of the Hamiltonian path to be an edge connecting twin nodes partly destroys the structure that has been helpful in chapter 5.2. We will briefly recall the proof of Theorem 53 in order to see why condition (28) changes the situation fundamentally.

The very reason why we could provide a direct matching-based characterization of the Hamiltonicity on threshold graphs in the preceding section consists in the fact that the vicinal preorder of threshold graphs is total. Due to this feature, a simple matching is sufficient to draw far-reaching conclusions about the structure of the graph. In particular, a node $i \in N_G$ can be connected to all nodes that have a higher value $v(j)$ than its mate j in the matching. In other words: the totalness of the vicinal preorder allows us to gain immediate information about the neighbourhood of the matching mate of a certain node. This means that, when trying to construct a Hamiltonian path on a threshold graph, we can look "two steps ahead" so to speak. I.e., when selecting a node as the next one along the Hamiltonian path, we do not only know the neighbourhood of the node in question but also have some information about the neighbourhoods of the nodes in that neighbourhood. In the end, it is this very feature of the vicinal neighbourhood of threshold graphs that turned out to translate the non-polynomial-time complexity of the general Hamiltonian path problem into a polynomial one for threshold graphs.

In the proof of Theorem 53, this feature was used to prove both the sufficiency and the necessity of the condition for Hamiltonicity. For the sufficient part, it allowed us to construct a complete Hamiltonian path just by ordering the pairs of the matching according to the vicinal preorder. With regard to the necessity of the condition provided in Theorem 53, we could use the information given by the vicinal preorder (in conjunction with the matching algorithm) for arguing that under certain conditions not even a graph with a larger edge set than the graph in question would permit the construction of a Hamiltonian path.

In contrast to this, the subset of edge set (27) that is induced by the twin node function does not preserve the convenient structure of the vicinal preorder in the general case as we have not made any assumptions about the structure of the twin node property. This means that, when building a Hamiltonian Path that satisfies property (28), every edge between twin nodes "interrupts" our inference about further possible connections between nodes such that we do

not have any immediate information about the node that possibly follows two steps later in the twin-constrained Hamiltonian path to be constructed. On the basis of this line of reasoning, we must take into account that the MSSP might be truly combinatorial in nature (i.e. NP-complete as the general Hamiltonian Path Problem) and cannot rely on finding a polynomial time algorithm for solving the MSSP.

However, despite these rather discouraging considerations about the MSSP, we can still benefit from our analysis in the previous chapters and try to exploit the fact that the graph underlying a MSSP contains a threshold graph as a subgraph. Therefore, though Theorem 53 is not of direct help here, we should not carelessly discard the general idea of approaching the MSSP on the basis of a matching on a threshold graph. After all, it still is the case that such a matching is contained in every solution of a MSSP according to (28). In fact, as it will be demonstrated in the following two chapters, the threshold property, when used in conjunction with an appropriate matching algorithm, is powerful enough to arrive at substantial results for the MSSP.

For solving the twin-constrained Hamiltonian Path Problem, we need to consider only those matchings in which nodes are *not* matched with their twin-nodes because every edge in the path (28) is either given by the twin-node function or an edge that matches two nodes that are not twin-nodes to each other. That is, for a given threshold graph $G(N, E)$ we have to find a (suitable) maximum cardinality matching on the edge set

$$E' := E \backslash \{(i, j) \in N_{G'} \times N_{G'} : j = b(i)\} \ . \tag{29}$$

As in the general case the graph $G'(N, E')$ is not a threshold graph anymore, we have to modify our matching algorithm TGMA appropriately.

Definition 73 *(Modified Matching on a Threshold Graph)*
Let $G(N, E)$ be a threshold graph, and $b : N \to N$ a twin-node function. A matching on G that contains only edges from the set E' as defined by (29) is called modified (with respect to b).

Algorithm 74 *(MTGMA - Modified Threshold Graph Matching Algorithm)*

Let $G\,(N, E)$ be an undirected graph with the (even) set of nodes
$N_G = \{1, 2, ..., 2n\}$, neighbourhoods $N(i)$ for all $i \in N_G$,
and $b : N_G \to N_G$ a twin-node function.

[01].*Sort all nodes in an order of non-decreasing degrees.*
Set node $i := 0$,
the matching list $m(i) := 0$ for all $i \in \{1, 2, ..., 2n\}$,
the set of all matched nodes $I := \varnothing$, and
the "modified neighbourhoods"

$N'(i) := N(i)\backslash\{b(i)\}$ *for all* $i \in N_G$.

[02] *Increase* i *by* 1. *If* $i = 2n$ *then STOP.*

[03] *If* $i \in I$ *or* $N(i)\backslash I = \varnothing$ *then go to* [02].

[04] *[* $i \notin I$ *and* $N(i)\backslash I \neq \varnothing$ *]:*

If $N'(i)\backslash I \neq \varnothing$ *then*

pick a node $j \in N'(i)\backslash I$,

set matching list $m(i) := j$ *and* $m(j) := i$,

add i, j *to set of matched nodes* I, *and go to* [02].

[05] *[* $i \notin I$ *and* $N(i)\backslash I = \{b(i)\}$ *]:*

If $i > 1$ *and* $i - 1 \in I$ *then*

if $m(i-1) \in N(i)$ *and* $b(i) \in N(i-1)$ *then*

set matching list as follows:

$m(i) := m(i-1)$ *and* $m(m(i-1)) := i$,

$m(i-1) := b(i)$ *and* $m(b(i)) := i - 1$,

and add i, $b(i)$ *to the set of matched nodes* I.

[06] *Go to* [02].

Proposition 75 *Let* $G(N, E)$ *be a threshold graph and* $b : N \to N$. *Then MTGMA yields a maximum cardinality modified matching on* G.

Proof. The basic idea of MTGMA is identical to TGMA (Algorithm 42), which has been proven in Corollary 50 to yield a maximum cardinality matching: we arrange the nodes in an order of non-decreasing degrees and, proceeding from the lowest to the highest node, we match each node that has not yet been matched ($i \notin I$) with some other unmatched node in its neighbourhood, provided $N(i)\backslash I \neq \varnothing$, i.e. provided such a node exists at all (see steps [01] to [04] and step [06] in this algorithm). MTGMA differs from TGMA in the following respect: as we would like to achieve a matching on the set of edges E' in (29), we do not permit in step [04] that a node i be matched with its twin-node $b(i)$ and hence try to find a mate in the neighbourhood $N'(i)\backslash I$. Obviously, as long as the set of so far unmatched neighbours $N(i)\backslash I$ contains either no node or at least one node that is not the twin node $b(i)$, the algorithm proceeds in the same way as TGMA and leads to a maximum cardinality matching according to Corollary 50.

However, whenever an unmatched node i has no unmatched neighbour except its twin-node $b(i)$ (step [05] in this algorithm: the case $i \notin I$ and $N(i)\backslash I = \{b(i)\}$), MTGMA checks whether i is not the first node ($i > 1$) and whether the preceding node $i-1$ has been matched ($i-1 \in I$). If this is the case and the twin node $b(i)$ is a neighbour of the preceding node ($b(i) \in N(i-1)$), the node i is matched with the original mate of the preceding node and the preceding node is matched with the twin node $b(i)$ instead. We will refer to this procedure as a "swap of mates"

in the following and have to prove that attempting a swap of mates in step [05] is an appropriate means for arriving at a maximal cardinality matching using the edges in the set E'.

First, note that we do not have to attempt any matching in step [05] if $i = 1$ and $N(i) = N(i)\backslash I = \{b(i)\}$, as it is, trivially, impossible under these conditions to match the node $i = 1$ using the edges in E'. Second, note that, for $i > 1$, a successful swap of mates in step [05] immediately increases the cardinality of the matching achieved so far and will definitely not reduce the overall cardinality of the matching at a later stage of running the algorithm because Corollary 50 for TGMA ensures that it does not matter which of the so far unmatched nodes $N(i)|I$ we choose for matching a certain node (as long as we proceed in a non-decreasing order of nodes).

It remains to show that there is no other way of increasing the overall cardinality of the matching if, in step [05], our attempt at swapping mates with node $i-1$ is not successful and we leave the node i unmatched as a consequence of this. We make the following observations:

(1) For matching a node i subject to the edge set E', we do not have to attempt more than a mere swap of mates with one single node (in particular we could not do better by trying out a permutation of the mates of several nodes) because, according to Corollary 50 about TGMA, at any stage of constructing a matching, the overall cardinality of a matching on threshold graphs is not affected by the particular nodes in $N(i)\backslash I$ that we have already chosen as mates for certain nodes i.

(2) Attempting a swap of mates with a (so far unmatched) node of a higher degree than the node i will never be successful because, due to the vicinal preorder of a threshold graph, we would never be able to match i with any so far unmatched neighbour of such a node (except with the common neighbour $b(i)$).

(3) Attempting a swap of mates with a so far unmatched node of the same degree as the node i will never be successful because, due to the vicinal preorder, the only unmatched neighbour of such a node is $b(i)$.

(4) Attempting a swap of mates with a matched node of a lower degree than the node $i-1$ will, due to the vicinal preorder, be unsuccessful if attempting a swap of mates with the node $i-1$ is unsuccessful.

(5) Attempting a swap of mates with a matched node of the same degree as the node $i-1$ will, due to the vicinal preorder, be successful if and only if attempting a swap of mates with the node $i-1$ is successful.

(6) If a swap of mates with node $i-1$ is not possible due to the fact that node $i-1$ has remained unmatched, $i-1$ must be of a lower degree than node i (otherwise $i-1$ would have been matched with $b(i)$) and no node of the same degree as $i-1$, or of a lower degree, would be a suitable candidate for a swap of mates.

In sum we have shown that node $i-1$ is the most suitable candidate for a swap of mates and that there is no other way of improving the cardinality of the matching if a swap of mates with node $i-1$ turns out to be impossible. ■

Remark 76 *MTGMA is an adapted version of TGMA (Algorithm* 42*) for computing modified matchings, i.e. maximum cardinality matchings on the edge set* E'*. Analogously, we will consider in the following also modified matchings based on the algorithms* $TGMA_{\min}$ *and* $TGMA_{\max}$ *(Algorithms* 46 *and* 47*). In doing so, we will have to take into account that the degree property of* $TGMA_{\max}$ *(Proposition* 51*) does not hold in general anymore. (See Proposition* 79*, however.).*

7.2 Twin-induced structure and the case $|M| \neq n-1$

Based on the preceding general considerations, let us make a first step towards a heuristic for the MSSP. In this section, we will proceed from a maximum cardinality matching on the threshold graph underlying a MSSP, as provided in chapter 5.1 and the algorithm of the previous section. In particular we will introduce the concept of the "twin-induced structure" of a matching and analyse the easy cases in which the cardinality of our maximum cardinality matching is less than or greater than $n-1$ (with n being the number of twin-node pairs, as in the previous chapters).

Let us observe what happens if we add to a (modified) maximum cardinality matching $M \subseteq E'_{G'}$on the threshold graph those edges that are generated by the twin node function. Figure 9 illustrates this setting for $n = 13$ pairs of twin nodes, with edges from the matching M being depicted as dashed lines, the edges arising from the twin node property as regular lines, and unmatched nodes being represented by small circles.

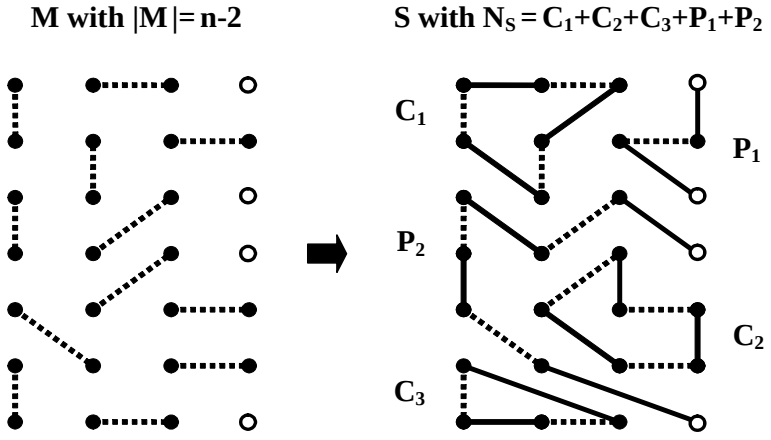


Figure 9: The twin-induced structure of a matching

Obviously, the subgraph resulting from the combination of the matching and the twin node connections is split into different components, each of which is a cycle or a path. Other components are not possible because every node is incident to either one or two edges: only one edge (the twin node edge) if the node has not been matched under M, and two edges if the node has been matched. All *cycles* consist only of nodes that have been *matched* in M, while a *path* results if and only if two *unmatched* nodes are connected via some edges from the matching and some edges due to the twin node property. (Note that the number of unmatched nodes is always even because so is the node set of the threshold graph underlying the MSSP.) This observation gives rise to the following definition.

Definition 77 *(Twin-induced structure of a matching)*

Let $G'(N,E)$ be the graph of an MSSP with twin node function $b : N_G \to N_G$, moreover $G(N,E)$ its threshold subgraph, and $M \subseteq E'_G$ a modified maximum cardinality matching on G under the constraint (29)*.*

Then the subgraph $S(N,E)$ of $G'(N,E)$ with $N_S := N_{G'}$ and

$$E_S := M + \{(i,j) \in N_{G'} \times N_{G'} : j = b(i)\}$$

is called the twin-induced structure of the matching M (with respect to b) and will be represented by the node set partition

$$N_S = C_1 + C_2 + ... + C_d + P_1 + P_2 + ... + P_q,$$

where the sets $C_1, ..., C_d$ and $P_1, ..., P_q$, respectively, consist of the nodes from the cycles and paths that are the components of S. Alternatively, the twin-induced structure will be represented by a partition of the matching, i.e.

$$M = C_1 + C_2 + ... + C_d + P_1 + P_2 + ... + P_q,$$

where the sets $C_1, ..., C_d$ and $P_1, ..., P_q$, respectively, consist of those edges from the cycles and paths of S that are also elements of the matching.

Remark 78 (1) *Note that we have $q = n - |M|$ for $|N_G| = 2n$.*

(2) *If the twin-induced structure consists only of one single path, i.e. $N_S = P_1$, or only of one single cycle, i.e. $N_S = C_1$, then the MSSP obviously is feasible, i.e. there exists a path that fulfills condition* (28)*.*

(3) *Observe that every cycle of S has at least length* 4*.*

(4) *For achieving a transparent presentation we will represent the twin-induced structure of a matching by means of the node set partition thoughout the present chapter, while in chapter* 8 *it will be more convenient to represent the twin-induced structure by subsets of the matching.*

Having introduced the concept of the twin-induced structure of a matching, let us note, before we proceed with the main topic of this chapter, a prerequisite for the last section of the present chapter and for chapter 8. The following proposition about a modified greedy

matching, i.e. a matching computed by $MTGMA_{\max}$, is the counterpart of the degree property of $TGMA_{\max}$ (Proposition 51), expressed by means of the twin-induced structure of a matching.

Proposition 79 *(Degree property of $MTGMA_{\max}$)*

Let $G(N, E)$ be a threshold graph, b a twin-node function, and M a modified matching on G that has been obtained from $MTGMA_{\max}$. Then for all (i_1, j_1), if there exists an edge $(i_2, j_2) \in M$ with $dg(i_1) > dg(i_2)$ and $dg(j_1) > dg(j_2)$, the edges (i_1, j_1) and (i_2, j_2) belong to the same cycle or path of the twin-induced structure of M with respect to b.

Proof. We know from Proposition 51 that the case $dg(i_1) > dg(i_2)$ and $dg(j_1) > dg(j_2)$ does not occur with matchings obtained by $TGMA_{\max}$. The only difference between $TGMA_{\max}$ and $MTGMA_{\max}$ lies in the fact that the latter occasionally performs "swaps of matching mates" (cf. the proof of Proposition 75), which can disturb the degree property of $TGMA_{\max}$ for the pairs of nodes involved in the swap. As such a swap of mates implies that a node j_1 is matched with the twin node $i_1 = b(j_2)$ of a node j_2, the edges (i_1, j_1) and (i_2, j_2) must belong to the same cycle or path of the twin-induced structure of M. (Note, however, that a swap of mates does not automatically lead to a situation in which $dg(i_1) > dg(i_2)$ and $dg(j_1) > dg(j_2)$.) ■

In the remainder of this chapter, we will analyse what the matching M and its twin-induced structure S tell us about the feasibility or infeasibility of the MSSP. Some conclusions are immediately at hand when we look at the cardinality of M.

Being a special case of the general Hamiltonian path problem, every solution (28) of the MSSP presupposes a matching on the threshold subgraph G of a cardinality of at least $n - 1$. Hence we can immediately decide on the infeasibility of an MSSP if our maximum cardinality matching is of cardinality $|M| < n - 1$.

In contrast to this, let us consider the case of a perfect matching, i.e. a matching with $|M| = n$.

Proposition 80 *(MSSP in the case of $|M| \neq n - 1$)*

Let $G'(N, E)$ be the graph of an MSSP, $G(N, E)$ its threshold subgraph, and $M \subseteq E'_G$ a modified maximum cardinality matching on G.

(a) If $|M| < n - 1$, the MSSP is infeasible.

(b) If M is a perfect matching, i.e. $|M| = n$, the MSSP is feasible.

Proof. It remains to show (b). According to Remark 78(1), the twin-induced structure S of the matching consists only of cycles $C_1, ..., C_d$. If $d = 1$, we directly have a solution of the MSSP. Otherwise, we exploit the threshold structure of the subgraph. Let us arbitrarily pick one node from each cycle of the matching structure S, i.e. let us choose nodes $i_1, i_2, ..., i_d \in N_S$ with $i_k \in C_k$ for $1 \leq k \leq d$. Because the vicinal preorder of G is total, we can renumber the node and cycles such that

$$i_1 \precsim i_2 \precsim ... \precsim i_d.$$

As the matching has cardinality $|M| = n$, there exist nodes $j_k \in N_S$ with $(i_k, j_k) \in M \subseteq E_S$ for all $k = 1, ..., d$, which, due to the vicinal preorder yields $j_k \in N(i_k) \subseteq N[i_{k+1}]$, i.e.$(j_k, i_{k+1}) \in E_G$ for $1 \leq k \leq d-1$, and even $(j_k, i_{k+1}) \in E'_G$. Therefore, we can define a new matching M' by

$$M' := M - \{(i_k, j_k) : k = 1, ..., d\} \\ +\{(j_k, i_{k+1}) : k = 1, ..., d-1\},$$

which is of cardinality $|M'| = n - 1$ and the twin-induced structure of which clearly fulfils condition (28). (See Figure 10 for an illustration of the process for the case $d = 3$, where the dotted thin edges are removed from M, and the curved edges are added when constructing M'.) Hence, every MSSP for which there exists a perfect matching on the underlying threshold graph is feasible. ■

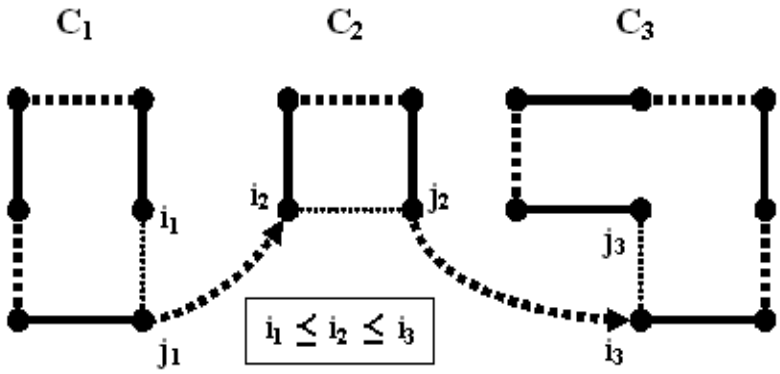


Figure 10: The case |M|=n

7.3 The case $|M| = n - 1$

While the cases of matchings with cardinality $|M| < n - 1$ or $|M| = n$ obviously can directly be solved in polynomial time for all possible instances, the case with cardinality $|M| = n - 1$ does not allow for an immediate treatment and brings back the combinatorial challenge into the problem. (Compare this with the setting of a general Hamiltonian path problem where the case $|M| = n$ is the only interesting one).

According to what has been said above (and this actually is the only information that we have so far), the instances with $|M| = n - 1$ are characterized by the fact that they are those MSSPs for which every maximum cardinality matching has a twin-induced structure of the form

$$N_S = C_1 + C_2 + ... + C_d + P_1 \text{ with } d \geq 0 \ . \tag{30}$$

Let us proceed by examining some examples to get a better insight into the nature of such a setting.

First of all, as Figure 11 shows, there obviously are both feasible and infeasible cases among the instances with $|M| = n-1$, which depends on the twin node structure as well as the options for a matching. The infeasibility of the second twin-induced structure in Figure 11 can easily be seen by looking at the degree partition of the underlying threshold graph and observing that the matching given in Figure 11 is the only maximum cardinality matching that exists on the graph. (The numbers in the circles represent the values $v\,(i)$ of the nodes; adjacency is defined by the threshold $\alpha = 70$.)

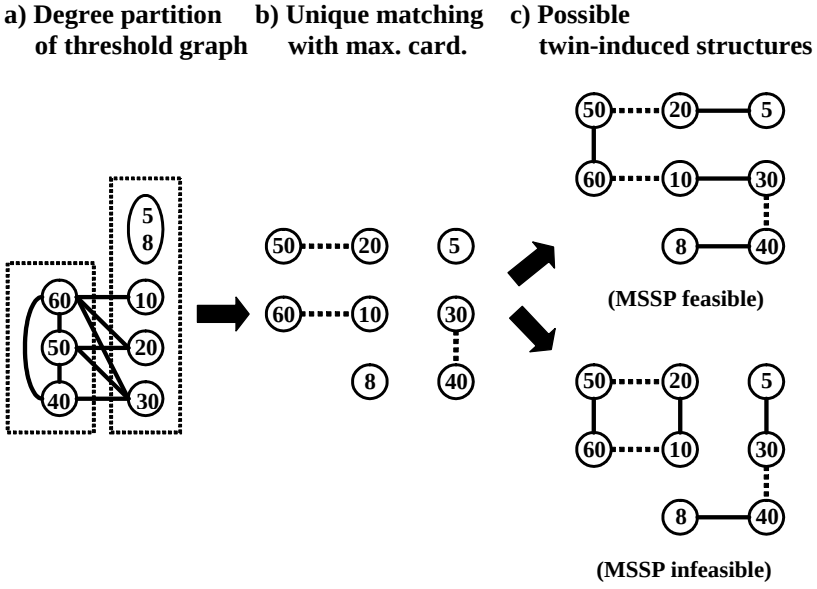


Figure 11: The case |M|=n-1

Of course, there are also instances in which there exists more than one maximum cardinality matching on the underlying threshold graph. If the MSSP under investigation is a feasible one, we can have the good luck to be immediately provided with a feasible solution by the matching algorithm (or not). Figure 12a) presents an example where there exist exactly two maximum cardinality matchings on the underlying threshold graph. In the case of the twin-node function of 12(d), we immediately arrive at a feasible solution of the MSSP by constructing the twin-induced structure with respect to the twin-node function, no matter which of the two matchings we choose. In the case of 12(c), we arrive at an feasible solution if we are so lucky as to start with the second matching ($N_S = P_1$, $d = 0$), while starting with the first matching, the twin-induced structure of which is not a feasible solution, leaves open the question of whether or

not our MSSP is feasible because we have $N_S = C_1 + P_1$. In contrast to this, Figure 12(b) still shows the same underlying threshold graph, but again with a different time node function. In this case, neither matching leads to a twin-induced structure that would fulfill condition (28), so the MSSP is infeasible.

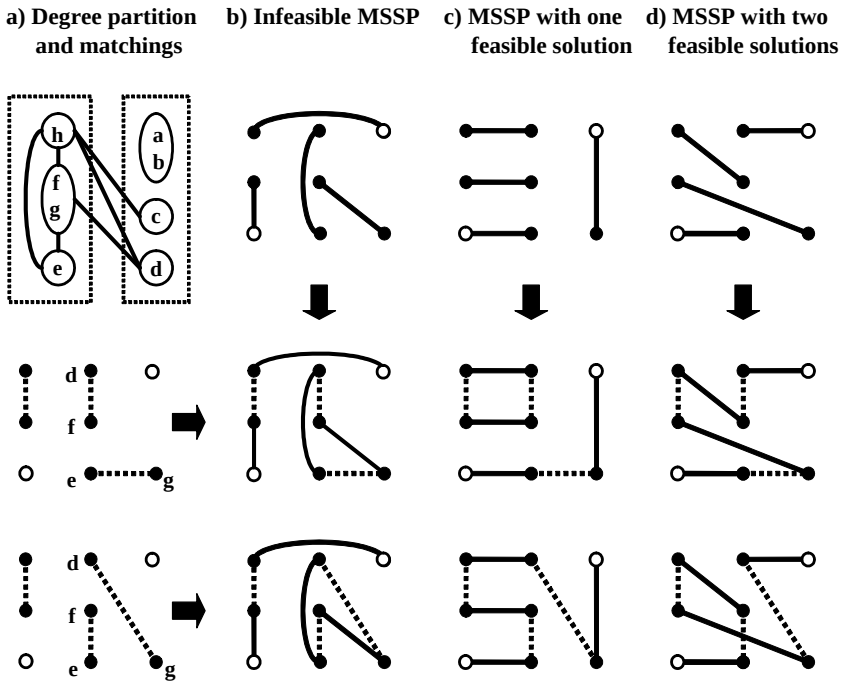


Figure 12: Matching, twin-node function and (in)feasibility

In principle, we could enumerate all possible maximum cardinality matchings on the underlying threshold graph and check if any among these has a twin-induced structure with $N_S = P_1$, $d = 0$. This would not even be a complicated task because section 5 has presented a simple matching algorithm for threshold graphs, and Remark 52(3) has addressed the question of how to obtain all possible maximum cardinality matchings on the basis of Algorithm 42. However, it is clear that this would still be a problem of non-polynomial (namely factorial) complexity. Therefore, it seems to be reasonable not to rush to other possible matchings, but instead to use the threshold property to exploit the information contained in the twin-induced structure of a given maximum cardinality matching.

The case of $|M| = n$ that we have discussed above suggest the idea that we could try to find also in the case of $|M| = n - 1$ some ways of recombining the cycles and the path of the twin-induced structure (30) in order to arrive at a matching the twin-induced structure of which

fulfils condition (28). This time, however, the situation is more difficult. While it was sufficient in figure 9 to transform a matching of cardinality $|M| = n$ into one of cardinality $|M| = n - 1$ (by substituting d edges for $d - 1$ edges), we are more restricted now and must make sure that every edge that we cancel from the given matching M is replaced by a new edge to preserve the cardinality of our matching.

Figure 13 provides two examples for this procedure. In example a), it is sufficient to remove two edges from the cycles and to add a new edge between the cycles and one edge to one of the ends of the path. In order to transform example 13(b) into a solution of the MSSP, however, we have to remove one edge from the path and three others from the cycle and add four new edges.

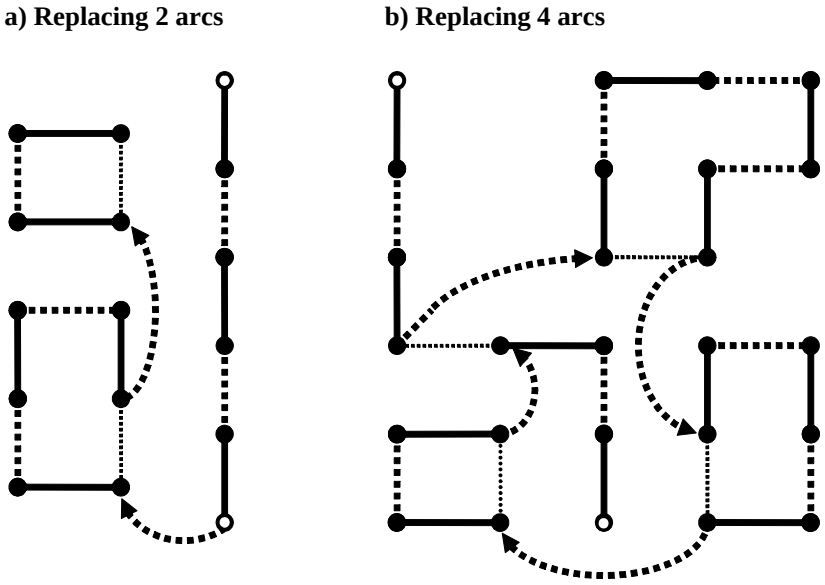


Figure 13: Constructing a feasible solution to the MSSP

Unfortunately, in the case of a more complicated twin-induced structure of our matching, this approach would again lead to a combinatorial explosion because the general approach of choosing a subset of edges to be replaced by new edges is obviously only a different perspective on the process of trying out a different matching. And yet, we can expect significant computational advantages indeed if we change only a few edges instead of calculating an entirely new maximum cardinality matching from scratch. Therefore, we will continue our approach in the following by analysing three cases: structure-preserving solutions, path-splitting solutions and cycle-splitting solutions.

7.4 Structure-preserving solutions for matchings with $|M| = n - 1$

If we restrict our attention to changing only a few edges, there is one rather general case that suggests itself quite naturally among all different options of recombining the components of the twin-induced structure of M. This case occurs when we add and remove only a *minimal* number of edges for obtaining a solution. This is the case in which all nodes remain in the same order in which they are given within the cycles and the paths of the twin-induced structure of the matching. To achieve this, we remove exactly one edge per cycle and no edge from the path, i.e. altogether d edges, and try to find exactly d new edges such that we arrive at the situation in condition (28). There are two subcases that are suitable for this procedure, which have been illustrated in figure 14 (the dotted thin lines denote the edges that have been removed, while the arrows represent those that have been added). The procedure of type 1 in Figure 14(a) consists of adding the path before (or after) all cycles, and type 2 in Figure 14(b) is the situation in which we include the path somewhere between the cycles. These two subcases are summarized in the following definition.

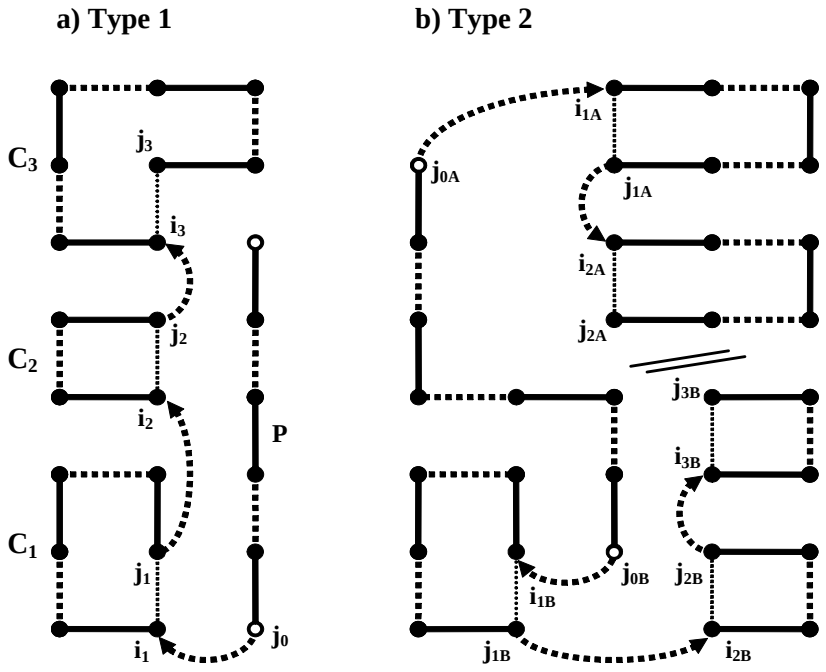


Figure 14: Types of structure-preserving solutions

Definition 81 *(Structure-preserving solution of an MSSP)*

Let $G'(N,E)$ be the graph of an MSSP, M a modified maximum cardinality matching on the underlying threshold subgraph $G(N,E)$ with cardinality $|M| = n-1$, and let the twin-induced structure of M be given by

$$N_S = C_1 + C_2 + ... + C_d + P \text{ with } d \geq 1 \ . \tag{31}$$

A solution of the MSSP is called structure-preserving with respect to M iff there exists a renumbering of the cycles and a suitable numbering of nodes such that the Hamiltonian path (28) *takes the form*

$$i_{1,1} - i_{1,2} - ... - i_{1,|Q_1|} - i_{2,1} - i_{2,2} - ... - i_{2,|Q_2|} - ...$$
$$... - i_{(d+1),1} - i_{(d+1),2} - ... - i_{(d+1),|Q_{d+1}|},$$

where

$$\{i_{l,1}, i_{l,2}, ..., i_{l,|Q_l|}\} = P \text{ for one } l \text{ with } 1 \leq l \leq d+1,$$
$$\{i_{k,1}, i_{k,2}, ..., i_{k,|Q_k|}\} = C_k \text{ for all } k \neq l \text{ with } 1 \leq k \leq d+1,$$

and the paths

$$i_{p,1} - i_{p,2} - ... - i_{p,|Q_p|} \text{ for all } 1 \leq p \leq d+1$$

are subgraphs of the twin-induced structure of the matching.

In the case of

$$\{i_{1,1}, i_{1,2}, ..., i_{1,|Q_1|}\} = P \text{ or } \{i_{(d+1),1}, i_{(d+1),2}, ..., i_{(d+1),|Q_{d+1}|}\} = P$$

we will speak of a structure-preserving solution of type 1, otherwise of a structure-preserving solution of type 2.

For a given twin-induced structure represented by (31), there are $\frac{|C_k|}{2}$ edges that can be removed per cycle (the other edges are determined by the twin node function), $\frac{(d+1)!}{2}$ different permutations of the cycles C_k and the path P (the factor 2 arises due to symmetry), and altogether 2^{d+1} ways of trying to insert both the path P and each path that remains from one of the cycles C_k into the structure-preserving solution (namely "forward" and "backward" each). Hence, there are

$$(d+1)! * \prod_{k=1}^{d} |C_k|$$

possible candidates for a structure-preserving solution of the MSSP for any maximum cardinality matching on the underlying threshold graph. Obviously, providing a general criterion for the existence of such a structure-preserving solution of an MSSP would save computational time considerably. Fortunately, exploiting the properties of the underlying threshold graph leads to such a criterion indeed.

Let us recall that our point of departure for considering structure-preserving solutions of the MSSP was the idea to use an approach for threshold graph matchings of cardinality $|M| = n-1$ that is similar to the one that was successful for matchings of cardinality $|M| = n$, where we combined the cycles of the twin-induced structure of the matching into to a solution of the

MSSP. In order to see better what we can do in the case of $|M| = n-1$, it is helpful to consider the case $|M| = n$ from a different, somewhat broader perspective.

In fact, we can interpret the case $|M| = n$, i.e. Proposition 80, in the light of Theorem 44(i). In doing so, we can observe in figure 9 that combining the cycles C_1, C_2 and C_3 actually means constructing a matching-dominated $\{(i_1, j_1), (i_2, j_2), (i_3, j_3)\}$-path relative to the given matching. In the general case, the main ingredient of Proposition 80 consists in constructing a matching-dominated path of the form

$$i_1 - j_1 - i_2 - j_2 - ... - i_d - j_d$$

from

$(i_k, j_k) \in M$ for $1 \leq k \leq d$, and
$(j_k, i_{k+1}) \in E'_G \backslash M$ for $1 \leq k \leq d-1$,

and we know from Theorem 44 that this is always possible on the basis of our matching M. It is for this very reason that a perfect matching always leads to a solution of the MSSP.

How does this perspective provide a hint for developing a criterion for structure-preserving solutions of an MSSP? We have noted above that constructing a structure preserving solution in the case $|M| = n - 1$ is equivalent to removing d edges and adding d new ones, while the recombination of cycles in the case of $|M| = n$ required only replacing d edges by $d-1$ new edges. In the perspective of alternating paths, this means adding d edges to d edges from a matching such that we arrive at an alternating path with altogether $2d$ edges (instead of altogether $2d-1$ in the case of a matching-dominated T-path), which precisely is the phenomenon of an even T-path.

Indeed, as figure 14(a) illustrates for the case of a structure-preserving solution of type 1, gluing together the cycles and the path means constructing an even path

$$j_0 - i_1 - j_1 - i_2 - j_2 - ... - i_d - j_d$$

from

$(i_k, j_k) \in M$ for $1 \leq k \leq d$, and
$(j_k, i_{k+1}) \in E'_G \backslash M$ for $0 \leq k \leq d-1$.

Therefore, we can immediately find a full characterization of structure-preserving solutions of type 1 on the basis of Theorem 44(ii). Moreover, it turns out that this insight makes it even possible to address type 2 on the basis of the result for type 1. We directly obtain the following theorem.

Theorem 82 *(Existence of structure-preserving solution of an MSSP)*

Let $G(N, E)$ be the underlying threshold subgraph of an MSSP and M a modified matching on G provided by MTGMA, with the twin-induced structure (31).

Then there exists a structure-preserving solution of the MSSP with respect to M if and only if one of the two endnodes of the path P is adjacent to some node from each of the cycles C_k, $1 \leq k \leq d$.

Proof. (1) Structure-preserving solution of type 1:

$\Leftarrow$: Assume that we have a structure-preserving solution of type 1 according to Definition 81 with the node sets $\{i_{k,1}, i_{k,2}, ..., i_{k,|Q_k|}\} = C_k$ for all k with $1 \leq k \leq d$ representing the cycles and $i_{(d+1),1}$ being the endnode of the path. P. Then $(i_{k,1}, i_{k,|Q_k|}) \in M$ and $(i_{k,|Q_k|}, i_{(k+1),1}) \in E'_G \backslash M$ for all $1 \leq k \leq d$. Hence

$$i_{1,1} - i_{1,|Q_1|} - i_{2,1} - i_{2,|Q_2|} - ... - i_{d,1} - i_{d,|Q_d|} - i_{(d+1),1}$$

is an even $\{(i_{1,1}, i_{1,|Q_1|}), (i_{2,1}, i_{2,|Q_2|}), ..., (i_{d,1}, i_{d,|Q_d|})\}$-path and Theorem 44($ii$) applies.

$\Rightarrow$: Let j_0 be the endnode of the path P that is adjacent to the nodes $j_k \in C_k$ for all $1 \leq k \leq d$. Further, let (j_k, l_k) be the corresponding pairs from the matching M such that the cycles C_k and the path P are represented by the node sets

$$C_k = \{j_k, i_{k,2}, ..., i_{k,|C_k|-1}, l_k\} \text{ for all } k \text{ with } 1 \leq k \leq d$$

and

$$P = \{j_0, i_{(d+1),2}, ..., i_{(d+1),|P|}\},$$

respectively, with the order of nodes in these sets being given according to the original order of nodes within the cycles C_k and the path.P.

Then, given an appropriate renumbering of the pairs (j_k, l_k) and, correspondingly, the cycles C_k, there exists an even alternating path

$$j_1 - l_1 - j_2 - l_2 - ... - j_d - l_d - j_0$$

according to theorem 44(ii). Hence,

$$j_1 - i_{1,2} - ... - i_{1,|C_1|-1} - l_1 - j_2 - i_{2,2} - ... - i_{2,|C_2|-1} - l_2 - ...$$
$$... - j_d - i_{d,2} - ... - i_{d,|C_d|-1} - l_d - j_0 - i_{(d+1),2} - ... - i_{(d+1),|P|}$$

is a structure-preserving solution of type 1.

(2) Structure-preserving solution of type 2:

Assume that there exists a structure-preserving solution of type 2, let the nodes i_{0A} and i_{0B} be the endnodes of the path P. Then, according to the line of argument in part (1), there exists a partition of the set of cycles $C_A + C_B = \{C_1, C_2, ..., C_d\}$ such that i_{0A} and i_{0B} are adjacent to some node from each cycle in C_A and C_B, respectively. We can assume $i_{0A} \succsim i_{0B}$ without loss of generality since the vicinal preorder on G is total, from which follows that i_{0A} is adjacent also to some node from each cycle in C_B. Therefore, whenever there exists a structure-preserving solution of type 2, there is also a structure-preserving solution of type 1. Consequently, the criterion proved in part a) is necessary and sufficient for the existence of structure-preserving solutions in general. ■

Remark 83 *Pertaining to computational effort, Theorem* 82 *can be exploited very efficiently. Because of the vicinal preorder of threshold graphs, we only have to check whether the larger one of the two endnodes of the path is adjacent to some node from each of the cycles. Moreover, also due to the vicinal preorder, we do not have to check adjacency for all nodes in all cycles, but instead it is sufficient to look for adjacency only with the largest node in each cycle. Finally, again because of the vicinal preorder, we only have to check if the higher of the two end nodes of the path is adjacent to smallest node among the largest nodes in all cycles.*

7.5 Classification of non-structure-preserving solutions for matchings with $|M| = n - 1$

In the previous subsection we have considered those solutions of the MSSP that arise when we change only a minimal number of edges (namely d edges, with d being the number of cycles of the twin-induced structure) from a given matching of cardinality $|M| = n - 1$, which led to the analysis of structure-preserving solutions. In a next step, we will consider those solutions that we obtain by changing $d + 1$ edges from the matching. Moreover, we will observe on this occasion that structure-preserving solutions are relevant for solving the MSSP also beyond the case of changing d edges. The additional $(d + 1)^{th}$ edge to be removed here can be an edge either from the cycle or from the path. We will start with the latter case.

Definition 84 *(Path-splitting solutions)*

Let $G'(N, E)$ be the graph of an MSSP, M a matching on the underlying threshold subgraph $G(N, E)$ with cardinality $|M| = n-1$, and let the twin-induced structure of M be given by (31)*.*

A solution of the MSSP is called path-splitting with respect to S iff there exists a suitable numbering of nodes such that the Hamiltonian path (28) *takes the form*

$$i_{1,1} - i_{1,2} - ... - i_{1,|Q_1|} - i_{2,1} - i_{2,2} - ... - i_{2,|Q_2|} - ...$$
$$... - i_{(d+2),1} - i_{(d+2),2} - ... - i_{(d+2),|Q_{d+2}|},$$

where

$$\{i_{l,1}, i_{l,2}, ..., i_{l,|Q_l|}\} + \{i_{m,1}, i_{m,2}, ..., i_{m,|Q_m|}\} = P$$
for some l and m with $1 \leq l, m \leq d + 2$ and $l < m - 1$,
$$\{i_{g(k),1}, i_{g(k),2}, ..., i_{g(k),|Q_{g(k)}|}\} = C_k$$
for all k with $1 \leq k \leq d$ and some bijective function
$$g : \{1, ..., d\} \to \{1, ..., d + 2\} - \{l, m\},$$

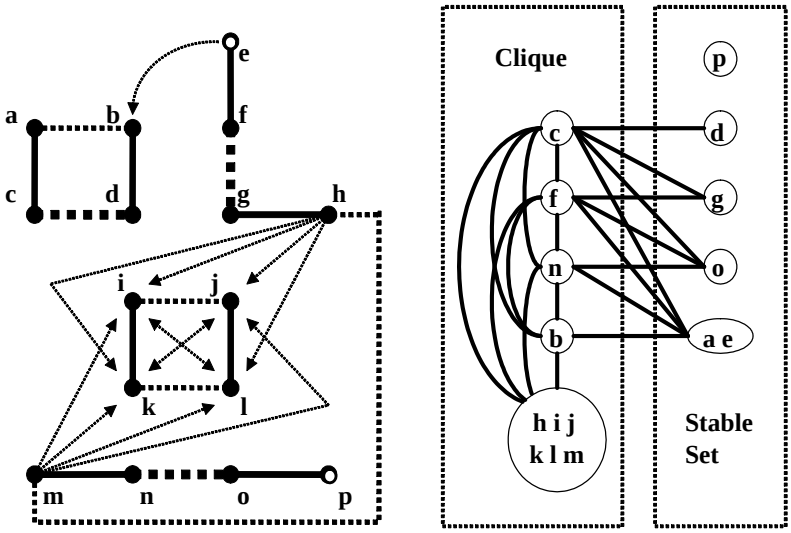
and the paths

$$i_{p,1} - i_{p,2} - ... - i_{p,|Q_p|} \text{ for all } 1 \leq p \leq d + 2$$

are subgraphs of the twin-induced structure of the matching.

Figure 15 illustrates two simple types of path-splitting solutions. In the case of type 1, the path P has been split such that the nodes of all cycles C_k can be arranged between the two segments of the path (without changing the order of the nodes within a cycle C_k). In the case of type 2, the path P has been split such that the nodes of some cycles C_k can be arranged between the two segments of the path, while the nodes of all other cycles can be attached to the "outer end" of one of the two segments of P (again without changing the order of nodes within a cycle).

a) Example of an irreducible path-splitting solution of Type 1



b) Example of an irreducible path-splitting solution of Type 2

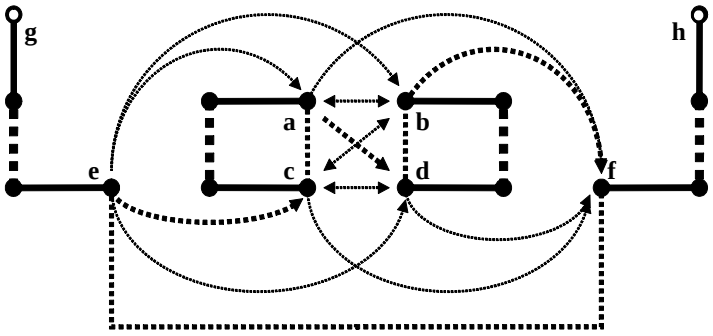


Figure 15: Irreducible path-splitting solutions

For convenience, we introduce the following conventions to refer to the structure of a path-splitting solution.

Notation 85 *(1) We will denote path-splitting solutions by*

$$X_1 - X_2 - ... - X_q,$$

with X_i, $1 \leq i \leq q$, referring to a subgraph of one of the following types:

a) the subgraph consists of the nodes of a cycle C_k, in which case we will write C for some X_i;

b) it consists of the nodes $\{i_{l,1}, i_{l,2}, ..., i_{l,|Q_l|}\}$ and we will write P_a for some X_i; or

c) it consists of the nodes $\{i_{m,1}, i_{m,2}, ..., i_{m,|Q_m|}\}$ *and we will write* P_b *for some* X_i.

(2) We write $\overrightarrow{P_a}$ *or* $\overrightarrow{P_b}$ *if* $i_{l,1}$ *or* $i_{m,1}$ *are one the (unmatched) endnodes of the path* P*, respectively, and write* $\overleftarrow{P_a}$ *or* $\overleftarrow{P_b}$ *if these endnodes are* $i_{l,|Q_l|}$ *or* $i_{m,|Q_m|}$*, respectively.*

(3) If $X_i = X_{i+1} = C$ *for some* $1 \leq i \leq q$*, we will simplify our notation by writing* C *instead of* $C - C$.

Remark 86 *In this notation, the path* P *of the twin-induced structure according to* (31) *can be represented as* $P = \overrightarrow{P_a}\overleftarrow{P_b} = \overrightarrow{P_b}\overleftarrow{P_a}$.

Using our notation, we can formally distinguish between the following 20 types of path-splitting solutions (with some types being equivalent due to symmetry; see below) :

(i) $\quad \overrightarrow{P_a} - C - \overleftarrow{P_b}$, $\overleftarrow{P_a} - C - \overleftarrow{P_b}$, $\overleftarrow{P_a} - C - \overrightarrow{P_b}$, $\overrightarrow{P_a} - C - \overrightarrow{P_b}$,

(ii) $\quad C - \overrightarrow{P_a} - \overleftarrow{P_b}$, $C - \overleftarrow{P_a} - \overleftarrow{P_b}$, $C - \overleftarrow{P_a} - \overrightarrow{P_b}$, $C - \overrightarrow{P_a} - \overrightarrow{P_b}$,

(iii) $\quad C - \overrightarrow{P_a} - C - \overleftarrow{P_b}$, $C - \overleftarrow{P_a} - C - \overleftarrow{P_b}$, $C - \overleftarrow{P_a} - C - \overrightarrow{P_b}$, $C - \overrightarrow{P_a} - C - \overrightarrow{P_b}$,

(iv) $\quad C - \overrightarrow{P_a} - \overleftarrow{P_b} - C$, $C - \overleftarrow{P_a} - \overleftarrow{P_b} - C$,
$C - \overleftarrow{P_a} - \overrightarrow{P_b} - C$, $C - \overrightarrow{P_a} - \overrightarrow{P_b} - C$, and

(v) $\quad C - \overrightarrow{P_a} - C - \overleftarrow{P_b} - C$, $C - \overleftarrow{P_a} - C - \overleftarrow{P_b} - C$,
$C - \overleftarrow{P_a} - C - \overrightarrow{P_b} - C$, $C - \overrightarrow{P_a} - C - \overrightarrow{P_b} - C$.

Each of these 20 types of path-splitting solutions represent a number of possible maximum cardinality matchings M the twin-induced structure of which takes the form of one of these types. Let us illustrate this by calculating the number of potential path-splitting solutions represented by the first 4 types of path-splitting solutions listed above. All these 4 types are characterized by the fact that the cycles are set between the two parts into which we have split the path. With d being the number of cycles of the twin-induced structure according to (31), there are $d!$ permutations of the cycles, $\prod_{k=1}^{d} |C_k|$ ways of arranging the cycles in the middle by removing altogether d edges, $\frac{|P|}{2} - 1$ ways of splitting the path by removing 1 edge, and 3 ways of gluing the two segments of the path to the left or to the right of the cycles (3 ways instead of 4 because the types $\overleftarrow{P_a} - C - \overleftarrow{P_b}$ and $\overrightarrow{P_a} - C - \overrightarrow{P_b}$ are symmetrical to each other), i.e. there are altogether

$$3d! * (\tfrac{|P|}{2} - 1) * \prod_{k=1}^{d} |C_k|$$

potential maximum cardinality matchings that could lead to a path-splitting solution of the MSSP and the twin-induced structure of which takes the form of one of the first 4 types of path-splitting solutions. Without calculating the combinatorial options for the other 16 types (some of which would lead to an even higher number of potential matchings, due to the fact that the number of matchings that would lead to the type $C - \overrightarrow{P_a} - C - \overleftarrow{P_b}$, for example, is clearly higher than the number of matchings that would lead to the type $\overrightarrow{P_a} - C - \overleftarrow{P_b}$, for example), it is obvious that a simple criterion for discovering whether the twin-induced structure of a

certain matching M gives rise to a path-splitting solution will greatly reduce the complexity of deciding whether or not a certain MSSP is feasible.

The following result is a significant step towards such a criterion.

Theorem 87 *(Classification of path-splitting solutions)*
Let $G'(N, E)$ be the graph of an MSSP, M a modified matching on the underlying threshold subgraph with cardinality $|M| = n - 1$ that has been obtained by MGTMA, and let S be the twin-induced structure of M. If there exists a path-splitting solution with respect to S, then there also exists a structure-preserving solution with respect to S, or the path-splitting solution takes one of the two forms

$$\overrightarrow{P_a} - C - \overleftarrow{P_b}, \text{ or}$$
$$C - \overrightarrow{P_a} - C - \overleftarrow{P_b}.$$

This theorem justifies the following definition.

Definition 88 *(Irreducible path-splitting solutions of type 1 and type 2)*
A path-splitting solution that takes the form $\overrightarrow{P_a} - C - \overleftarrow{P_b}$ or $C - \overrightarrow{P_a} - C - \overleftarrow{P_b}$ is called an (irreducible) path-splitting solution of type 1 or type 2, respectively.

Note that the irreducible path-splitting solutions of type 1 and type 2 are precisely the two types illustrated in Figure 14 above.

Proof. We will address the 20 cases of path-splitting solutions in the order given in the list above.

(i) Regarding $\overleftarrow{P_a} - C - \overrightarrow{P_b}$, note that we can connect the end of $\overleftarrow{P_a}$ that has been matched in the path-splitting solution with the unmatched end of $\overrightarrow{P_b}$ such that we get the structure-preserving solution $C - \overrightarrow{P_a} - \overleftarrow{P_b}$ because $P = \overrightarrow{P_a} - \overleftarrow{P_b}$.is the (un-split) path in the original matching M.

The same applies to $\overrightarrow{P_a} - C - \overrightarrow{P_b}$ and to the symmetric case $\overleftarrow{P_a} - C - \overleftarrow{P_b}$.

For $\overrightarrow{P_a} - C - \overleftarrow{P_b}$ there is nothing to show as it this one of the two types mentioned in the theorem.

(ii) The type $C - \overrightarrow{P_a} - \overleftarrow{P_b}$ is the structure-preserving solution of type 1, hence there is nothing to show.

Regarding $C - \overleftarrow{P_a} - \overrightarrow{P_b}$, there either exists also a structure-preserving solution, or the two endnodes of $\overleftarrow{P_a}$ and $\overrightarrow{P_b}$ that were not matched under M must be members of a maximal stable set of G (otherwise we would have a structure preserving solution again according to Theorem 54), in which case a solution of the type $C - \overleftarrow{P_a} - \overrightarrow{P_b}$ cannot not exist.

Regarding $C - \overleftarrow{P_a} - \overleftarrow{P_b}$, the endnode of $\overleftarrow{P_a}$ that is connected to $\overleftarrow{P_b}$ must be a member of a maximal stable set of G if there is no structure-preserving solution. Hence it must have at

least the same degree as the endnode of $\overleftarrow{P_a}$ that is connected to C (otherwise $MTGMA$ would have connected it to $\overleftarrow{P_b}$ in the original matching M), which implies the contradiction that the structure-preserving solution $C - \overrightarrow{P_a} - \overleftarrow{P_b}$ exists.

Regarding $C - \overrightarrow{P_a} - \overrightarrow{P_b}$, this type directly implies the existence of $C - \overrightarrow{P_a} - \overleftarrow{P_b}$.

(iii) For $C - \overrightarrow{P_a} - C - \overleftarrow{P_b}$ there is nothing to show as this is one of the two types mentioned in the theorem.

Regarding $C - \overleftarrow{P_a} - C - \overrightarrow{P_b}$, if there is no structure-preserving solution, the endnodes of $\overleftarrow{P_a}$ and $\overrightarrow{P_b}$ that are connected to the middle C must be members of a maximal stable set of G. Hence, both endnodes of the C in the middle must be members of a maximal clique of G. This is not possible because otherwise $MGTMA$ would have matched the endnodes of $\overleftarrow{P_a}$ and $\overrightarrow{P_b}$ with the endnodes of the middle C in the first instance.

Regarding $C - \overrightarrow{P_a} - C - \overrightarrow{P_b}$, we can connect $\overrightarrow{P_b}$ to $\overrightarrow{P_a}$ as in the original matching M, with the middle C remaining connected to $\overrightarrow{P_b}$. This yields $C - \overrightarrow{P_a} - \overleftarrow{P_b} - C$, which is structure-preserving of type 2.

Regarding $C - \overleftarrow{P_a} - C - \overleftarrow{P_b}$, we will consider the case $C_1 - C_2 - \overleftarrow{P_a} - C_3 - C_4 - \overleftarrow{P_b}$ without loss of generality. If there is no structure-preserving solution, the right endnode of $\overleftarrow{P_a}$, which we will call k in the following, must be a member of a maximal stable set of G. Hence the left endnode of C_3 is a member of a maximal clique of G, which implies that the right endnode of C_3 has at most the same degree as k (otherwise $MTGMA$ would have matched k and the left endnode of C_3 in the original matching M). Consequently, due to the vicinal preorder of G, the node k must be adjacent to the left endnode of C_4 and, again due to the way in which $MTGMA$ works, the right endnode of C_4 has at most the same degree as k. As this right endnode of C_4 is adjacent to the left endnode of $\overleftarrow{P_b}$, so is k. Therefore, and again due to $MTGMA$, the left endnode of $\overleftarrow{P_a}$, which was matched with the left endnode of $\overleftarrow{P_b}$ under the original matching M, must have at most the same degree as k. Consequently, k is adjacent to the left endnode of C_2. This and the fact that C_1 can be connected to C_2 implies that we can construct an even alternating path with the endnodes of C_1 and C_2 and the exposed node k. Hence, according to Theorem 44(ii), the node k must be adjacent to one node from each of the two endnodes of the cycles C_1 and C_2. Additionally, we have already shown in our line of argument that k is adjacent to one node from each of the two endnodes of the cycles C_3 and C_4. Due to Theorem 44(ii), there exists an even alternating path that consists of the endnodes of all four cycles C_1, C_2, C_3, C_4 and the node k. This implies that there exists a permutation π of the four cycles such that $\overrightarrow{P_b} - \overleftarrow{P_a} - C_{\pi(1)} - C_{\pi(2)} - C_{\pi(3)} - C_{\pi(4)}$ is a feasible solution of the MSSP, i.e. we could construct a structure-preserving solution of type 1.

(iv) The case $C - \overrightarrow{P_a} - \overleftarrow{P_b} - C$ is the structure-preserving solution of type 2.

Regarding $C - \overrightarrow{P_a} - \overrightarrow{P_b} - C$, we assume that there is no structure-preserving solution. Then the left endnode of $\overrightarrow{P_b}$ must be a member of a maximal stable set of G. If this left endnode had a degree that were lower than the degree of the right endnode of $\overrightarrow{P_b}$, our algorithm $MTGMA$, in the original matching M, would have connected this left endnode node to the right endnode

of $\overrightarrow{P_a}$ (instead of matching the right endnode of $\overrightarrow{P_a}$ with the right endnode of $\overrightarrow{P_b}$). Hence, the degree of the right endnode of $\overrightarrow{P_b}$ is at most the degree of the left endnode of $\overrightarrow{P_b}$ This implies that we can connect also the left endnode of $\overrightarrow{P_b}$ to the right cycle C. This, however, is a contradiction as it allows for the structure-preserving solution of type 2, i.e. $C-\overrightarrow{P_a}-\overleftarrow{P_b}-C$.

Regarding $C-\overleftarrow{P_a}-\overleftarrow{P_b}-C$, we observe that this case is symmetric to the previous case.

Regarding $C-\overleftarrow{P_a}-\overrightarrow{P_b}-C$, we must have also a structure-preserving solution according to Theorem 82 here because one of the two endnodes that connect $\overleftarrow{P_a}$ with $\overrightarrow{P_b}$ must be a member of a maximal clique of G.

(v) In the case of $C-\overleftarrow{P_a}-C-\overrightarrow{P_b}-C$, we can apply the same line of reasoning that we used for type $C-\overleftarrow{P_a}-C-\overrightarrow{P_b}$ in part (iii), which proves the existence of a structure-preserving solution.

Regarding $C-\overrightarrow{P_a}-C-\overleftarrow{P_b}-C$, we show that, if there is no structure-preserving solution, this solution implies the existence of a solution of the type $C-\overrightarrow{P_a}-C-\overleftarrow{P_b}$, which is one of the two types mentioned in the theorem. Let us assume that there is no structure-preserving solution. Then the left endnode of $\overrightarrow{P_a}$ and the right endnode of $\overleftarrow{P_b}$ must be members of a maximal stable set due to Theorem 82. By applying the same line of reasoning that we used for the case $C-\overleftarrow{P_a}-C-\overleftarrow{P_b}$ in part (iii), we can conclude that the left endnode of $\overrightarrow{P_a}$ is adjacent to at least one node among the two endnodes of each of those cycles that make up the left C. In the same fashion we can conclude that the right endnode of $\overleftarrow{P_b}$ is adjacent to at least one node among the two endnodes of each of those cycles that make up the right C. Consequently, due to the vicinal preorder, the higher node among the left endnode of $\overrightarrow{P_a}$ and the right endnode of $\overleftarrow{P_b}$ is adjacent to at least one node among the two endnodes of *all* cycles that make up the left C and the right C. Again analogously to the case $C-\overleftarrow{P_a}-C-\overleftarrow{P_b}$ in part (iii), we can conclude on the basis of Theorem 44(ii) that there exists an alternating path that consists of the higher of the two endnodes and all cycles that make up the left C and the right C. This implies that there exists a permutation of the left and the right cycles C such that all left and right cycles can be glued to the higher of the two endnodes, i.e. we arrive at either $C-\overrightarrow{P_a}-C-\overleftarrow{P_b}$ or its symmetric counterpart $\overrightarrow{P_a}-C-\overleftarrow{P_b}-C$.

Regarding $C-\overrightarrow{P_a}-C-\overrightarrow{P_b}-C$, we can apply a line of reasoning similar to (albeit slightly more complex than) the previous case. If the left endnode of $\overrightarrow{P_a}$ is of a higher degree than the left endnode of $\overrightarrow{P_b}$ (or both degrees are equal), this procedure leads to the conclusion that a permutation of the cycle(s) in the middle can be glued to a permutation of the cycle(s) on the left such that we arrive at a solution of the type $C-\overrightarrow{P_a}-\overrightarrow{P_b}-C$, which was addressed in part (iv). If, conversely, the left endnode of $\overrightarrow{P_a}$ is of a lower degree than the left endnode of $\overrightarrow{P_b}$, this procedure leads to the conclusion that some permutation of all cycles (i.e. those on the left, in the middle and on right) can be glued to the left endnode of $\overrightarrow{P_b}$, which yields $C-\overrightarrow{P_b}-\overleftarrow{P_a}$, i.e. the structure-preserving solution of type 1.

Finally, the case $C-\overleftarrow{P_a}-C-\overleftarrow{P_b}-C$ is symmetric to the previous case. ■

Note that the examples of two irreducible path-splitting solutions presented in Figure 14 above are "irreducible" indeed, namely in the sense that the structure of the underlying threshold graph does not allow for any matching based on edges other than those indicated by arrows. In particular, this implies that, in the cases of the examples given in Figure 14, it is not possible to construct a structure-preserving solution on the graph, nor to reduce one of the two types to the other. Consequently, these two examples prove the existence of genuine path-splitting solutions.

Having discussed path-splitting solutions, we now turn to the other case of changing $d+1$ edges from the original matching M. This case occurs when we remove 2 edges from 1 cycle and 1 edges from each of the other $d-1$ cycles and can be treated in a fashion similar to path-splitting solutions.

Definition 89 *(Cycle-splitting solutions)*

Let $G'(N,E)$ be the graph of an MSSP, M a modified matching on the underlying threshold subgraph $G(N,E)$ with cardinality $|M| = n-1$ that has been obtained by MTGMA, and let S, the twin-induced structure of M be given by (31)*. A solution of the MSSP is called cycle-splitting with respect to S iff there exists a renumbering of the cycles and a suitable numbering of nodes such that the Hamiltonian path* (28) *takes the form*

$$i_{1,1} - i_{1,2} - ... - i_{1,|Q_1|} - i_{2,1} - i_{2,2} - ... - i_{2,|Q_2|} - ...$$
$$... - i_{(d+2),1} - i_{(d+2),2} - ... - i_{(d+2),|Q_{d+2}|},$$

where

$$\{i_{l,1}, i_{l,2}, ..., i_{l,|Q_l|}\} + \{i_{m,1}, i_{m,2}, ..., i_{m,|Q_m|}\} = C_{k^*}$$
$$\text{for some } k^* \text{ with } 1 \leq k^* \leq d$$
$$\text{and some } l, m \text{ with } 1 \leq l, m \leq d+2 \text{ and } l < m-1,$$
$$\{i_{j,1}, i_{j,2}, ..., i_{j,|Q_j|}\} = P$$
$$\text{for some } j \neq l, m \text{ with } 1 \leq j \leq d+2,$$
$$\{i_{g(k),1}, i_{g(k),2}, ..., i_{g(k),|Q_{g(k)}|}\} = C_k$$
$$\text{for all } k \neq k^* \text{ with } 1 \leq k \leq d \text{ and some bijective function}$$
$$g : \{1, ..., d\} - \{k^*\} \rightarrow \{1, ..., d+2\} - \{j, l, m\},$$

and the paths

$$i_{p,1} - i_{p,2} - ... - i_{p,|Q_p|} \text{ for all } 1 \leq p \leq d+2$$

are subgraphs of the twin-induced structure of the matching.

Corresponding to the aforementioned conventions, we refer to the structure of a cycle-splitting solutions as follows.

Notation 90 *(1) We will denote cycle-splitting solutions by*

$$X_1 - X_2 - ... - X_q,$$

with X_i, $1 \leq i \leq q$, referring to a subgraph of one of the following types:

a) the subgraph consists of the nodes of a cycle C_k, $k \neq k^*$, in which case we will write C for some X_i;

b) it consists of the nodes $\{i_{j,1}, i_{j,2}, ..., i_{j,|Q_j|}\}$ and we will write P for some X_i;

c) it consists of the nodes $\{i_{l,1}, i_{l,2}, ..., i_{l,|Q_l|}\}$ and we will write C_a for some X_i;

d) it consists of the nodes $\{i_{m,1}, i_{m,2}, ..., i_{m,|Q_m|}\}$ and we will write C_b for some X_i.

(2) We write $\overrightarrow{C_a}$ and $\overleftarrow{C_b}$ if $(i_{l,|Q_l|}, i_{m,1}) \in M$ and $(i_{l,1}, i_{m,|Q_m|}) \in M$, while we write $\overrightarrow{C_a}$ and $\overrightarrow{C_b}$ if $(i_{l,1}, i_{m,1}) \in M$ and $(i_{l,|Q_l|}, i_{m,|Q_m|}) \in M$.

(3) If $X_i = X_{i+1} = C$ for some $1 \leq i \leq q$, we will simplify $C - C$ by C.

Using our notation, we can classify cycle-splitting solutions into the following 26 types:

(i) $P - \overrightarrow{C_a} - C - \overleftarrow{C_b}$, $P - \overrightarrow{C_a} - C - \overleftarrow{C_b} - C$,
$P - C - \overrightarrow{C_a} - C - \overleftarrow{C_b}$, $P - C - \overrightarrow{C_a} - C - \overleftarrow{C_b} - C$,

(ii) $P - \overrightarrow{C_a} - C - \overrightarrow{C_b}$, $P - \overrightarrow{C_a} - C - \overrightarrow{C_b} - C$,
$P - C - \overrightarrow{C_a} - C - \overrightarrow{C_b}$, $P - C - \overrightarrow{C_a} - C - \overrightarrow{C_b} - C$,

(iii) $\overrightarrow{C_a} - P - \overleftarrow{C_b}$, $\overrightarrow{C_a} - C - P - \overleftarrow{C_b}$, $\overrightarrow{C_a} - C - P - C - \overleftarrow{C_b}$,

(iv) $C - \overrightarrow{C_a} - P - \overleftarrow{C_b}$, $C - \overrightarrow{C_a} - C - P - \overleftarrow{C_b}$,
$C - \overrightarrow{C_a} - C - P - C - \overleftarrow{C_b}$, $C - \overrightarrow{C_a} - P - \overleftarrow{C_b} - C$,
$C - \overrightarrow{C_a} - C - P - \overleftarrow{C_b} - C$, $C - \overrightarrow{C_a} - C - P - C - \overleftarrow{C_b} - C$,

(v) $\overrightarrow{C_a} - P - \overrightarrow{C_b}$, $\overrightarrow{C_a} - C - P - \overrightarrow{C_b}$, $\overrightarrow{C_a} - C - P - C - \overrightarrow{C_b}$,

(vi) $C - \overrightarrow{C_a} - P - \overrightarrow{C_b}$, $C - \overrightarrow{C_a} - C - P - \overrightarrow{C_b}$, $C - \overrightarrow{C_a} - C - P - C - \overrightarrow{C_b}$,

(vii) $C - \overrightarrow{C_a} - P - \overrightarrow{C_b} - C$, $C - \overrightarrow{C_a} - C - P - \overrightarrow{C_b} - C$,
$C - \overrightarrow{C_a} - C - P - C - \overrightarrow{C_b} - C$,

Figure 16(a) presents a cycle-splitting solution of the type $P - \overrightarrow{C_a} - C - \overleftarrow{C_b}$.

It will not be necessary here to illustrate again the number of potential matchings the twin-induced structures of which could give rise to a cycle-splitting solution. Instead we immediately turn to the central result, which, similar to the case of path-splitting solutions, reduces significantly the number of potential matchings that we have to investigate for deciding on the feasibility of the MSSP on the basis of a potential cycle-splitting solution.

Theorem 91 *(Classification of cycle-splitting solutions)*

Let $G'(N, E)$ be the graph of an MSSP, M a greedy matching on the underlying threshold subgraph with cardinality $|M| = n - 1$, and let S be the twin-induced structure of M. If there exists a cycle-splitting solution with respect to S, then there also exists a structure-preserving solution with respect to S, or the cycle-splitting solution takes one of the four forms

$$P - \overrightarrow{C_a} - C - \overleftarrow{C_b}, \text{ or } P - \overrightarrow{C_a} - C - \overleftarrow{C_b} - C, \text{ or}$$
$$P - C - \overrightarrow{C_a} - C - \overleftarrow{C_b}, \text{ or } P - C - \overrightarrow{C_a} - C - \overleftarrow{C_b} - C.$$

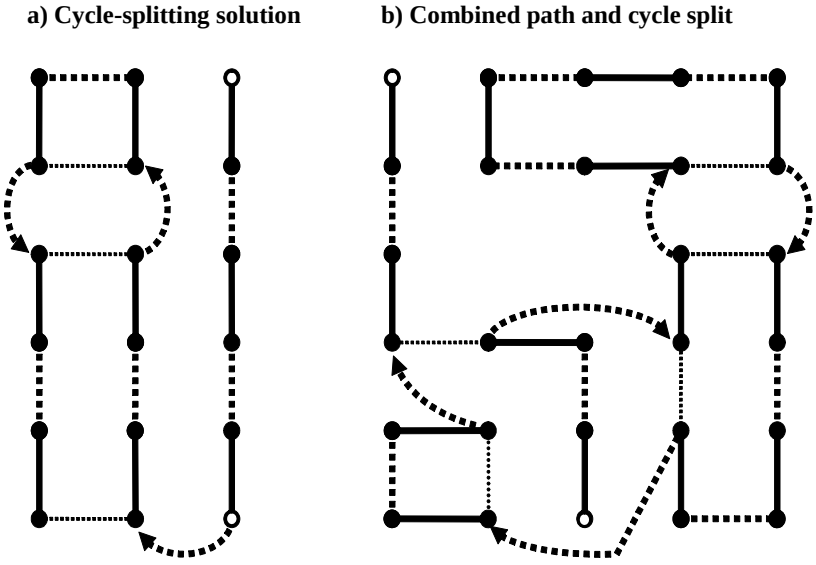


Figure 16: Solutions with a cycle-split

Proof. In a fashion similar to the proof Theorem 87, however all cycle-splitting solutions that do not take one of the forms in the present theorem can be shown to imply structure-preserving solutions of type 1 or type 2. Following the order of the list above, we sketch the phenomena that allow for a transformation of these solutions into structure-preserving ones.

(i) These are the four cycle-splitting solutions given in the theorem, so there is nothing to show.

(ii) The solutions in this group have in common that the edges that they delete from the original matching M and that they add to it form an even alternating path with the exposed node being one endnode of the path P. With Theorem 44(ii) and Theorem 82 this implies a structure-preserving solution.

(iii) The cycle segment $\overleftarrow{C_b}$ can be removed from the end of the twin-constrained alternating path and glued to $\overrightarrow{C_a}$ as in the original matching M. This directly yields a structure-preserving solution of type 1 or of type 2.

(iv) If there is no structure-preserving solution, we can conclude (from the fact that the path P is in the middle of the cycle-split solution, and in a fashion similar to the case $C-\overleftarrow{P_a}-C-\overleftarrow{P_b}$ in Theorem 87) that the two nodes of the split cycle between which the path P has been glued (under the new matching of the solution) must both be members of a maximal clique of G. As a consequence, we leave one of the endnodes of the path unmatched and we attach, to one

of the members of a maximal clique, all cycles that are not between the split cycle and the other endnode of the path (on the basis of Theorem 44(*ii*)). This yields a structure-preserving solution, which contradicts the assumption that there is no structure-preserving solution.

(*v*) Here we have the same situation as in (*iii*), but this time we glue the cycle segment $\overrightarrow{C}_b$ to $\overrightarrow{C}_a$ and attain a structure-preserving solution of type 1 or of type 2.

(*vi*) In these cases, we can cut the twin-constrained path after the cycle segment $\overrightarrow{C}_a$ and attach segment $\overrightarrow{C}_b$ to $\overrightarrow{C}_a$ (as in the original matching) such that all cycle or path segments that are currently placed between $\overrightarrow{C}_a$ and $\overrightarrow{C}_b$ will follow after $\overrightarrow{C}_b$.

(*vii*) Similar to (*iv*), but in this case we can attach some cycles to the (now) unmatched end of the path P (instead of attaching them to the second member of the maximal clique of G). ■

Analogous to the case of irreducible path-splitting solutions, it can be shown that the four types of cycle-splitting solutions in the preceding theorem are "genuine" in the sense that there exist matchings on threshold graphs such that their twin-induced structures are of one of these four types and it is not possible to construct, on the basis of a different matching, a solution of a different type. This insight and the preceding theorem justify the following definition.

Definition 92 *(Irreducible cycle-splitting solutions)*
A cycle-splitting solution that takes one of the forms $P-\overrightarrow{C}_a-C-\overleftarrow{C}_b$, $P-\overrightarrow{C}_a-C-\overleftarrow{C}_b-C$, $P-C-\overrightarrow{C}_a-C-\overleftarrow{C}_b$, *or* $P-C-\overrightarrow{C}_a-C-\overleftarrow{C}_b-C$ *is called irreducible.*

7.6 Existence of non-structure-preserving solutions for matchings with $|M| = n - 1$

Having classified path-splitting and cycle-splitting solutions, we will now develop an algorithm for finding path-splitting and cycle-splitting solutions of a given MSSP, provided there exists such a solution for an MSSP that does not have a structure-preserving solution.

According to Theorems 87 and 91 we can focus on six irreducible solutions, which all involve subpaths of the form

$$\overrightarrow{P}_a - C - \overleftarrow{P}_b \text{ or } \overrightarrow{C}_a - C - \overleftarrow{C}_b \tag{32}$$

(1) First we observe, that each of these subpaths corresponds to an alternating cycle that contains exactly one edge from each cycle represented by the symbol C and exactly one edge from the split path P $(= \overrightarrow{P}_a - \overleftarrow{P}_b)$ or the split cycle C_{k^*} $(= \overrightarrow{C}_a - \overleftarrow{C}_b)$. Conversely, whenever there exists an alternating cycle that consists of such a selection of edges, we can construct subpaths that have the structure given in (32).

(2) Second, considering the two types of irreducible path-splitting solutions

$$\overrightarrow{P}_a - C - \overleftarrow{P}_b \text{ and } C - \overrightarrow{P}_a - C - \overleftarrow{P}_b,$$

we observe that those cycles of a path-splitting solution that are not part of the subpath in (32) must be attached to one endnode of the path in the same way as this is the case with structure-preserving solutions, namely by forming an even alternating path that contains one edge from each of the cycles concerned, with the node of $\overrightarrow{P}_a$ that was unmatched under the original matching M being the exposed node of the even alternating path. This implies according to Theorem 44(ii) that a cycle is suitable for being attached to a segment of the path P in this way if and only if it contains a node that is adjacent to the larger node among the two endnodes of the path P.

(3) Third, we observe that a similar setting exists in the case of irreducible cycle-splitting solutions. In particular, all cycles that are not part of the subpath $\overrightarrow{C}_a - C - \overleftarrow{C}_b$, "contribute" to an even alternating path that has an endnode of the path P as the exposed node and contains exactly one edge from each of these cycles and one edge from the cycle C_{k^*}. Again, Theorem 44(ii) states that this is possible if and only if one node from each of these cycles (including the cycle C_{k^*}) is adjacent to the larger one of the endnodes of the path P.

In sum: each of the irreducible solutions contains one alternating cycle by virtue of one of the subpaths in (32), and one even alternating path due to the other components (the path or other cycles) of the twin-induced structure S, with one unmatched node of P being the exposed node of the even alternating path.

These considerations lead to the following proposition.

Proposition 93 *(Edge criterion for the existence of irreducible solutions)*

Let $G(N, E)$ be the underlying threshold subgraph of an MSSP, and M a maximum cardinality modified matching on G with the twin-induced structure S of M given by (31)*.*

If there exists no structure-preserving solution of the MSSP with respect to S, there exists a path-splitting or a cycle-splitting solution of the MSSP with respect to S if and only if there exists an alternating T-cycle with respect to M such that T consists of

(a) *at most one edge from the path P and and at most one from each of the cycles C_k, $1 \leq k \leq d$,*

(b) *at least one edge from each of those cycles C_k whose largest node is not adjacent to the largest endnode of P, and*

(c) *at least one edge*

(1) from the path P, or

(2) from one of those cycles whose two largest nodes are adjacent to the largest endnode of P, or

(3) from one of those cycles only one node of which is adjacent to the largest endnode of P, but not the edge that matches this one node.

Proof. $\Longleftarrow$: We show how to construct a path- or cycle-splitting solution if we have an alternating T-cycle that fulfills conditions (a) to (c). Because of (a), the segments of the twin-induced structure (i.e. path P and the cycles C_k) can be split into two groups depending on

whether or not one of their edges is an element of the alternating T-cycle. Because of (b), all cycles whose largest node is not adjacent to the largest endnode of P belong to the group of segments with an edge in the T-cycle. With respect to (c), we distinguish two cases.

case (i): the T-cycle contains an edge from P. On the basis of all edges in T we can construct a path-splitting solution of the type $\overrightarrow{P_a} - C - \overleftarrow{P_b}$ with C representing all cycles that have an edge in T (because of either (b) or (c)). Because of (b), all remaining cycles (if there are any) must have a node (which is their largest node) that is adjacent to the unmatched endnodes of $\overrightarrow{P_a}$ or $\overleftarrow{P_b}$, depending on which unmatched endnode is the larger one. Hence, we can construct an even alternating path (according to Theorem 44(ii)) that contains one edge from each of these remaining cycles and the exposed node of which is the larger one of the unmatched endnodes of $\overrightarrow{P_a}$ and $\overleftarrow{P_b}$. This implies that we can glue these remaining cycles to the end of $\overrightarrow{P_a}$ or $\overleftarrow{P_b}$ and obtain a path-splitting solution of the form $\overrightarrow{P_a} - C - \overleftarrow{P_b} - C$ or $C - \overrightarrow{P_a} - C - \overleftarrow{P_b}$.

case (ii): the T-cycle does not contain an edge from P.

STEP 1: We choose one of the cycles that has an edge in T due to condition $(c2)$ or $(c3)$. To be prepared for the the case of $(c2)$, i.e. that we choose a cycle with an edge in T has two largest nodes that are adjacent to the largest endnode of P, we show that one of these two nodes of the cycle must be incident (under the matching M) to an edge *not* in T. If this were not the case, the two nodes that are adjacent to the largest endnode of P would be mates under M. If there is no structure-preserving solution these two mates must be members of a maximal clique of G (otherwise they would not be adjacent to the larger endnode of P). This, however, is not possible due to the way in which $MTGMA$ works because $MTGMA$ would have matched one of these two mates with the larger endnode of P rather than making these two nodes of the cycle mates under M. Hence one of these two largest nodes of the chosen cycle must be incident (under the matching M) to an edge not in T. If we decide to choose a cycle that fulfills condition $(c3)$, we immediately know that we have chosen a cycle with a node incident to an edge not in T.

STEP 2: We construct an alternating even path that (1) contains one edge from each of the cycles that do not have an edge in T and (2) contains one edge from the chosen cycle such that this edge is not in T and (3) contains the larger endnode of P as its exposed node. Constructing this path is possible according to Theorem 44(ii) because we know that (1) all cycles that do not have an edge in T have a node that is adjacent to the largest endnode of P (the exposed node) because of condition (b), and that (2) the chosen cycle has a node that is adjacent to the exposed node and is incident (under the matching M) to an edge not in T (as shown in STEP 1). This alternating path allows us to glue P and the cycles that contribute an edge to the alternating path, such that we obtain a twin-constrained path of the form $P - C$.

STEP 3: We consider all cycles with an edge in T. These are those cycles that have not been included in the path $P - C$ we constructed in STEP 2 *and* the one cycle that we chose in STEP 1 and have included in the path $P - C$ in STEP 2. Because all these cycles have an edge in T, we have an alternating T-cycle and can construct a twin-constrained cycle

that contains all nodes from these cycles. Now the cycle that we chose in STEP 1 is part of both the twin-constrained path $P - C$ and the twin-constrained cycle such that the edge that the chosen cycle contributes to the even alternating path that led to $P - C$ is distinct from the edge that the chosen cycle contributes to the alternating cycle that led to the twin-constrained cycle. This yields a cycle-split solution of one of the types on Theorem 91, with the cycle that has been split into the two segments $\overrightarrow{C_a}$ and $\overleftarrow{C_b}$ being the cycle chosen in STEP 1 and the cycles arranged between the segments $\overrightarrow{C_a}$ and $\overleftarrow{C_b}$ being all other cycles with edges in T.

$\Longrightarrow$: The general idea underlying this part of the proof has been explained above as a motivation for this proposition. Because of this, we focus here on defining, for every type of irreducible path- and cycle-splitting solution, the appropriate set T that fulfills conditions (a) to (c).

$\overrightarrow{P_a} - C - \overleftarrow{P_b}$: The set T is defined to contain the edge from the path that connects the segments $\overrightarrow{P_a}$ and $\overleftarrow{P_b}$ in the original matching M (condition $(c1)$) and one edge from each of the cycles (conditions (a) and (b)). The existence of the alternating T-cycle is guaranteed by the existence of $\overrightarrow{P_a} - C - \overleftarrow{P_b}$.

$C - \overrightarrow{P_a} - C - \overleftarrow{P_b}$: We define the set T such that it contains the edge from the path that connects the segments $\overrightarrow{P_a}$ and $\overleftarrow{P_b}$ in the original matching M (condition $(c1)$) and one edge from all cycles between the segments $\overrightarrow{P_a}$ and $\overleftarrow{P_b}$. Obviously, condition (a) is fulfilled. The existence of the (sub-)path $\overrightarrow{P_a} - C - \overleftarrow{P_b}$ guarantees the existence of an alternating T-cycle. The cycles attached to $\overrightarrow{P_a}$ at the beginning of the path-splitting solution must contain, according to Theorem 44(ii), a node that is adjacent to the endnode of $\overrightarrow{P_a}$. As all other cycles have an edge in T, condition (b) is fulfilled.

$P - \overrightarrow{C_a} - C - \overleftarrow{C_b}$: The set T is defined to contain one edge from each of the cycles, which fulfills conditions (a) and (b). If there exists a cycle the two largest nodes of which are adjacent to the largest node of P, condition $(c2)$ is directly satisfied. If there exists no such cycle, we make sure that T contains an edge such that condition $(c3)$ is satisfied. This is possible because our irreducible solution contains the sub-path $P - \overrightarrow{C_a} - C$. The existence of the alternating T-cycle is guaranteed by the existence of the sub-path $\overrightarrow{C_a} - C - \overleftarrow{C_b}$.

$P - \overrightarrow{C_a} - C - \overleftarrow{C_b} - C$: We define the set T such that it contains one edge from each of the cycles of the sub-path $\overrightarrow{C_a} - C - \overleftarrow{C_b}$, which satisfies condition (a). Regarding condition (b), we observe that each of the cycles at the end of the irreducible solution (i.e. each of the cycles that do not have an edge in T) contribute to an even alternating path the exposed node of which is one endnode of P. Due to Theorem 44(ii) each of these cycles must have node that is adjacent to this endnode of P. This implies that T contains edges from all cycles required by condition (b). The edge of the split cycle $\overrightarrow{C_a} - ... - \overleftarrow{C_b}$ that is in T fulfills either condition $(c2)$ or $(c3)$. The existence of the alternating T-cycle is guaranteed by the existence of the sub-path $\overrightarrow{C_a} - C - \overleftarrow{C_b}$.

$P - C - \overrightarrow{C_a} - C - \overleftarrow{C_b}$: Analoguous to the previous case, with the cycle between the segments P and $\overrightarrow{C_a}$ taking the role of the cycle at the end of the irreducible solution in the

previous case.

$P - C - \overrightarrow{C_a} - C - \overleftarrow{C_b} - C$: Analoguous to the two previous cases, with this time the cycles in front of $\overrightarrow{C_a}$ and after $\overleftarrow{C_b}$ being those cycles that are not required, by condition (b), to contribute an edge to the alternating T-cycle. ■

In view of the previous theorem, if we would like to find out whether a certain twin-induced structure of a matching allows for a path- or cycle-splitting solution, we have to solve the combinatorial problem of finding an alternating T-cycle that fulfils the conditions (a) to (c) of our theorem. We will now attempt at solving this combinatorial problem by modelling it on the basis of a network flow problem with additional constraints.

Instead of directly looking for an alternating T-cycle we will approach this problem by looking for a flow that is a matching-dominated T-path starting from a node that dominates all other nodes in the path. This flow will be required to contain all edges from the matching that Proposition 93 calls for according to conditions (a) to (c). Once we have found this matching-dominated T-path we can connect its endnodes (because one of the endnodes is dominating) and obtain an alternating T-cycle that meets all requirements of Theorem 65. Note that looking for a matching-dominated T-path instead of an alternating T-cycle is no restriction because all alternating T-cycles trivially contain a matching-dominated T-path. Also, starting the path from a dominating node is no restriction because we know from Theorem 63 that our alternating T-path must contain a dominating node. Therefore, looking for a matching-dominated T-path that starts with a dominating node and is based on a set T that fulfills conditions (a) to (c) is both necessary and sufficient in the light of Proposition 93.

The flow problem is constructed as follows:

(1) Create a source that emits a flow of one unit, which constitutes the starting point of the flow, i.e. of the matching-dominated T-path.

(2) Connect the source to the larger nodes of all edges of all cycles C_k and the path P. (If the two nodes have the same degree, connect either of them.)

(3) Add all nodes in N_G as nodes in the flow problem, and add all edges in the matching M as a directed arc from the higher node to the lower one. (If both nodes have the same degree arbitrarily choose one direction.)

(4) Add directed arcs from every lower node incident to the edges in M to any higher node of any edge in M that this node is adjacent to and that is not part of the same cycle or path. (In case of a tie, choose arbitrarily.)

(5) Create a sink that receives a flow of one unit.

(6) Connect every lower node incident to edges in M to the sink using a directed arc. (In case of a tie, choose arbitrarily.)

(7) Require the flow to use those arcs that represent those edges of M that fulfill conditions (a) to (c) in Proposition 93.

$(8a)$ Require the flow on an arc representing an edge from M to be zero if the head of this arc is not adjacent to the node that receives the flow from the source.

(8b) Require the flow on an arc representing an edge from M to be zero if the tail of this arc has a higher degree than the node that receives the flow from the source.

(9) Require the flow to be integer.

Note that (8a) and (8b) in the description of this flow problem ensure that the flow, coming from the source, starts with a dominating node: (8b) stipulates that the flow must not pass any node that is of a higher degree than the first node passed immediately after the source, and (8a) determines that the node the flow passes immediately before going into the sink is adjacent to the first node immediately passed after the source. The following proposition provides a formal version of this flow problem and states its relation to the problem of deciding whether there exists a path- or cycle-splitting solution to the MSSP on the basis of the twin-induced structure of a matching.

Proposition 94 *(Necessary polyhedral criterion for irreducible solutions)*

Let $G(N,E)$ be the underlying threshold subgraph of an MSSP with node set $N_G = \{1,2,...,2n\}$, M a modified matching on G, the twin-induced structure of the MSSP with respect to M given by

$$N_S = C_1 + C_2 + ... + C_d + P \text{ with } d \geq 1,$$

let $i_0, i_1 \in P$ be the unmatched endnodes of the path P with i_0 being the higher one, i.e.

$$dg(i_0) \geq dg(i_1),$$

and let the function

$$c : \bigcup_{1\leq k\leq d} C_k \to \{1,2,...,d\} \text{ with}$$
$$i \mapsto c(i) = k :\Leftrightarrow i \in C_k \text{ for all } i \in \bigcup_{1\leq k\leq d} C_k$$

assign to each node the index of the twin-induced cycle it is an element of.

Morever, we define

$$A^* := \{(i,j) \in E_G : i = \arg\max_{l\in\{i,j\}} dg(l)\}$$

to be a set of arcs that correspond to the edges of G such that the arcs' tails are higher than the heads in terms of the vicinal preorder of G.

Further, let

$$I_{C0} := \{k \in \{1,...,d\} : \max_{j\in C_k} v(j) + v(i_0) < \alpha\}$$

be the set of indices of those cycles whose largest node is not adjacent to an endnode of the path P,

$$I_{C2} := \{k \in \{1,...,d\} : \exists j,l \in C_k : v(j) + v(i_0) \geq \alpha \wedge v(l) + v(i_0) \geq \alpha\}$$

the set of indices of those cycles whose largest two nodes are adjacent to an endnode of P,

$$I_{C1} := \{1,...,d\} - I_{C0} - I_{C2}$$

the set of indices of those cycles that contain exactly one node that is adjacent to an endnode of P,

$$A^*_k := \{(i,j) \in A^* \cap M : i,j \in C_k\} \text{ for all } 1 \leq k \leq d$$

for each cycle of the twin-induced structure the set of arcs that represent those edges of the cycle that are given by M,

$$A^*_{d+1} := \{(i,j) \in A^* \cap M : \ i,j \in P\}$$

the set of the arcs that represent the edges of M that are part of the twin-induced path P,

$$H_k := \bigcup_{(i,j)\in A^*_k} \{i\} \text{ for all } 1 \le k \le d+1$$

the set of the higher ones of the two nodes incident to the arcs in A^*_k,

$$H_0 := \bigcup_{1\le k\le d+1} H_k$$

the set of the higher nodes of all matched pairs, and let

$$L_0 := N_G - H_0 - \{i_0, j_0\}$$

be the set of the lower nodes of all matched pairs.

For $I_{C0} \neq \varnothing$ there exists a path-splitting or a cycle-splitting solution with respect to S only if the polyhedron P_I defined by the (in)equalities

Flow out of source:

$$\sum_{j\in H_0} x^{(A)}_j = 1 \tag{33}$$

Flow into sink:

$$\sum_{j\in L_0} x^{(B)}_j = 1 \tag{34}$$

Flow balance for higher nodes of all edges of M:

$$x^{(A)}_i - y_{i,j} = 0$$

$$\text{for all } i \in H_0 \text{ and } j \in L_0 \text{ with } (i,j) \in A^* \cap M \tag{35}$$

Flows balance for lower nodes of all edges of M:

$$y_{i,j} - \sum_{\substack{(k,j)\in A^*,\\ k\in H_0 - H_{c(j)}}} z_{j,k} - x^{(B)}_j = 1$$

$$\text{for all } j \in L_0 \text{ and } i \in H_0 \text{ with } (i,j) \in A^* \cap M \tag{36}$$

Exactly one edge used from each cycle in I_{C0}:

$$\sum_{(i,j)\in A^*_k} y_{i,j} = 1 \text{ for all } k \in I_{C0} \tag{37}$$

At most one edge used from P and from each cycle in $I_{C1} + I_{C2}$:

$$\sum_{(i,j)\in A^*_k} y_{i,j} \le 1 \text{ for all } k \in I_{C1} + I_{C2} + \{d+1\} \tag{38}$$

At least one edge from P or from a cycle in $I_{C1} + I_{C2}$
(excluding edges of cycles in I_{C1} a node of which is adjacent to i_0):

$$\sum_{(i,j)\in(\bigcup_{k\in I_{C1}+I_{C2}+\{d+1\}} A_k^*)\setminus\{(i,j)\in\bigcup_{k\in I_{C1}} A_k^*:(i,i_0)\in A_G\}} y_{i,j} \geq 1 \tag{39}$$

Arcs based on M with a head not adjacent to the node that receives the flow from the source must have a flow of zero:

$$y_{i,j} \leq 1 - x_k^{(A)} \text{ for all } k \in H_0$$

$$\text{and } (i,j) \in A^* \cap M \text{ with } (k,j) \notin E_G \tag{40}$$

Arcs based on M with a tail of a higher degree than the node that receives the flow from the source must have a flow of zero:

$$y_{i,j} \leq 1 - x_k^{(A)} \text{ for all } k \in H_0$$

$$\text{and } (i,j) \in A^* \cap M \text{ with } dg(i) > dg(k) \tag{41}$$

contains a point with coordinates

$$x_i^{(A)} \in \{0,1\} \text{ for all } i \in H_0 \ ,$$

$$x_j^{(B)} \in \{0,1\} \text{ for all } j \in L_0 \ ,$$

$$y_{i,j} \in \{0,1\} \text{ for all } (i,j) \in M \cap A^* \ ,$$

$$z_{j,i} \in \{0,1\} \text{ for all } (i,j) \in A^* \text{ with } i \in H_0 - H_{c(j)} \ . \tag{42}$$

Remark 95 *If $\arg\max_{l\in\{i,j\}} dg(l)$ in the definition of A^* above is not well-defined, we choose an arbitrary $l \in \{i,j\}$ that maximizes $dg(l)$.*

Proof. For a given MSSP without a structure-preserving solution (i.e. $I_{C0} \neq \varnothing$) the existence of a path- or cycle-splitting solution implies according to Proposition 93 the existence of a set $T \subseteq M$ that fulfills the conditions (a) to (c) such that there exists an alternating T-path. We have argued above that the existence of an alternating T-path is equivalent to the existence of a matching-dominated T-path that starts with a dominating node. What remains to be shown is that the existence of a matching-dominated T-path that starts with a dominating node implies the existence of a feasible integer point in the polyhedron P_I if T fulfills conditions (a) to (c) of Theorem 93.

Regarding the flows and the variables defining P_I, note that the variables $x_i^{(A)}$ denote flows from the source to all higher nodes of all arcs that represent edges in M, while the variables $x_j^{(B)}$ refer to flows from all corresponding lower nodes to the sink. The variables $y_{i,j}$ correspond to flows on the arcs that represent edges in M, and the variables $z_{j,i}$ denote all arcs from the heads of the arcs given by $y_{i,j}$ to the tails of all arcs to which the former head is adjacent with respect to the underlying graph G. (The fact that we use directed arcs ensures that all flows represented by the integer points of the polyhedron have the structure of an alternating path relative to M.) The four types of flows mentioned are represented by constraints (33) to (36), and any matching-dominated T-path relative to M certainly satisfies these constraints.

A matching-dominated T-path that fulfills conditions (a) to (c) of Proposition 93 also allows for a flow that satisfies constraints (37) to (39): constraints (37) restrict the flow to exactly one edge from I_{C0}, which is equivalent to what conditions (a) and (b) stipulate for the edges of those cycles whose highest node is not adjacent to the highest endnode of the path P; constraints (38) express condition (a) for the case of edges that are from the path P or one of the remaining cycles; and constraints (39) represent the restriction given by condition (c).

Finally, constraints (40) and (41) ensure that the node from G that the flow starts with is a dominating node among all nodes from G that the flow visits, which is a condition that our matching-dominated T-path satisfies. ■

The way in which we have constructed our flow problem suggests the idea that a solution of the flow problem could not only be necessary, but also sufficient for the existence of a path- or cycle-splitting solution. Unfortunately, this is not the case. The (only) reason is that the constraints defining P_I do not exclude a feasible solution within the polyhedron that contains, apart from a matching-dominated flow from the source to the sink, one or more "subcycles", i.e. cyclic flows of 1 unit that are not connected with the matching dominated flow from the source to the sink. In the presence of these subcycles, the requirements of constraints (36) to (38) (i.e. of conditions (a) to(c) of Theorem 93) are satisfied by the set of all arcs with a non-zero flow, and not just solely by the arcs that constitute the matching-dominated path. On other words: our constraints do not imply the existence of a matching-dominated T-path that starts with a dominating node, but only, for a certain subset $P \subseteq T$, the existence of a matching-dominated P-path starting with a dominating node and the existence of subcycles that contain the edges $T \backslash P$.

In principle there is a direct way of overcoming this problem, the general idea of which we will briefly sketch here as a side note.

We know from our discussion of alternating T-paths in chapter 6 (Theorem 63) that the existence of an alternating T-cycle is equivalent to the existence of a matching-dominated P-path ($P \subseteq T$) that starts with a dominating node and ends with a node in a maximal clique of the set of the nodes that are incident to edges in T. One remarkable aspect of this theorem is that it states that we do not need to have a flow that visits all edges of T as long as this flow starts from a dominating node and ends with a member of a maximal clique of nodes in T, i.e.

as long as this flow visits the "cornerstones" of the degree partitions that the nodes incident to edges in T belong to. On the basis of such a flow on the subset P, Theorem 63 guarantees that it is possible to complete the alternating T-cycle by adding all other edges in $T \backslash P$. For our setting, this implies that (provided we can make sure that the flow from the source to the sink does not only start with a dominating node, but also ends with a node in a maximal clique) we do not have to worry about possible sub-cycles because their edges $T \backslash P$ could always be integrated with the edges P from the flow to yield a matching-dominated T-path.

Moreover, it is indeed possible to restrict the polyhedron P_I to a polyhedron P_I' by adding further constraints such that a feasible point of P_I' does not only represent a matching-dominated flow starting with a dominating node, but also requires this flow to end with a member of a maximal clique. All we have to do is to add the constraints

$$\begin{aligned} y_{i,j} \leq 1 - x_k^{(B)} \text{ for } k \in L_0 \text{ and } (i,j) \in A^* \cap M \\ \text{with } dp_k < dp_j < m + 1 - dp_k \text{ for } dp_k \leq \frac{m}{2} \\ \text{and } m + 1 - dp_k < dp_j < dp_k \text{ for } dp_k > \frac{m}{2} \text{ , and} \end{aligned} \tag{43}$$

$$\begin{aligned} y_{i,j} \leq 1 - x_k^{(B)} \text{ for } k \in L_0 \text{ and } (i,j) \in A^* \cap M \\ \text{with } dp_i < m + 1 - dp_k \text{ for } dp_k \leq \frac{m}{2} \\ \text{and } dp_i < dp_k \text{ for } dp_k > \frac{m}{2} \text{ ,} \end{aligned} \tag{44}$$

with dp_i denoting the number of the degree partition that a node i is an element of, i.e.

$$dp_i = s :\Longleftrightarrow i \in D_s \text{ .}$$

These constraints (43) and (44) can be seen as the counterparts of constraints (40) and (41) in the following sense: the latter constraints ensure that the flow starts with a dominating node. They achieve this by restricting, depending on where the flow from the source goes, the degree of those nodes through which the flow may pass. Similarly, the former constraints ensure that the flow ends with a member of a maximal clique, which is also achieved by restricting the degree of those nodes through which the flow may pass. This time, however, the restriction depends on where the flow to the sink comes from.

We will end our side note here and will not go further into the details; instead we will continue with a slightly different approach. The reason for this is that the (both necessary and sufficient) criterion for the existence of path-splitting and cycle-splitting solutions that would result from following the idea as sketched in our side-note does not seem to lead to an efficient algorithm. In fact, it can be shown (but will not be shown here) that the A-matrix defining the polyhedron P_I is, due to constraints (40) and (41), not totally unimodular and that we will not

arrive at a totally unimodular A-matrix if we define a polyhedron P_I' by adding constraints (43) and (44). Finding a flow problem with a totally unimodular matrix, however, would certainly be the best possible move towards an efficient algorithm.

7.7 Existence of non-structure-preserving solutions for a greedy matching with $|M| = n - 1$

The alternative approach we will take in the following is to use a richer structure on the underlying matching M such that we do not require constraints (40) and (41) and will arrive at a more convenient polyhedron that allows for an efficient algorithm for deciding whether or not there exists a path- or a cycle-splitting solution.

The starting point for this approach is one of the main results of chapter 6. We demonstrated in chapter 6.3 that a *greedy* matching leads to a particularly tight characterization of alternating T-cycles. Theorem 65 states that the existence of an alternating T-cycle relative to a greedy matching is equivalent with the existence of a matching dominating T-path that starts with a node that dominates all nodes incident to edges in T and ends with a member of a maximal clique in the set of the nodes incident to edges in T. In other words: we have a condition for the entire set T and not merely for a subset $P \subseteq T$.

At first sight, this tighter condition does not seem to be much of help. In our side note above, on a possible approach to overcoming the subcycles that are permitted under the constraints (34) to (42), we used Theorem 63, according to which a flow on the subset P is sufficient to provide us with information about the existence of an alternating T-path. Drawing conclusions from the existence of a mere subset P allowed for integrating the (unwelcome) subcycles $P \backslash T$ into the solution. How could it now be fruitful an approach to tackle our problem on the basis of Theorem 65, which explicitly requires us to find a flow on the full set T?

The decisive aspect here is that greedy matchings do not only yield a condition for the entire set T, but also permit us to restrict our search for an alternating T-cycle to a subset of alternating T-cycles. The structure of greedy matchings is so pronounced that, according to Corollary 68, there exist an alternating T-cycle if and only if there exist a *sorted* alternating T-cycle, i.e. an alternating T-cycle

$$i_0 - j_0 - i_1 - j_1 - ... - i_{\frac{|T|}{2}-2} - j_{\frac{|T|}{2}-2} - i_{\frac{|T|}{2}-1} - j_{\frac{|T|}{2}|-1} \tag{45}$$

with

$$i_k \succeq j_k \text{ for } 0 \leq k \leq \frac{|T|}{2} - 1 \ , \tag{46}$$

$$i_0 \succeq i_1 \succeq ... \succeq i_{\frac{|T|}{2}-2} \succeq i_{\frac{|T|}{2}-1} \text{ , and} \tag{47}$$

$$j_{\frac{|T|}{2}-1} \succeq j_{\frac{|T|}{2}-2} \succeq ... \succeq j_1 \succeq j_0 \tag{48}$$

(cf. Definition 67). Apart from allowing us to restrict our search for alternating T-cycles to those with the structure (46) to (48), these statements automatically imply that the first node i_0 is a node that dominates all nodes incident to edges of T and that the last node $j_{\frac{|T|}{2}|-1}$ is a member of a maximal clique in the set of all nodes incident to edges in T.

The consequences of this setting can be summed up as follows: for deciding whether or not there exists a path- or a cycle-splitting solution it is necessary and sufficient to find a matching-dominated T-path such that the nodes incident to edges in T are sorted according to (46) to (48) and the edges in T fulfill conditions (a) to (c) of Proposition 93.

This insight leads to the following theorem.

Theorem 96 *(Polyhedral criterion for the existence of irreducible solutions)*

Let $G(N, E)$ be the underlying threshold subgraph of an MSSP with node set $N_G = \{1, 2, ..., 2n\}$, the set M be a greedy modified matching on G, the twin-induced structure of the MSSP with respect to M be given by

$$N_S = C_1 + C_2 + ... + C_d + P \text{ with } d \geq 1 \ ,$$

and the sets A^*, I_{C0}, I_{C1}, I_{C2}, A_k^* and H_k for all $1 \leq k \leq d+1$, H_0, and L_0 be defined as in Proposition 94.

Then there exists an injective "order function"

$$order[.] : A^* \cap M \rightarrow \{1, 2, ..., |A^* \cap M|\}$$

with

$$order[(i_1, j_1)] > order[(i_2, j_2)] \Longrightarrow (i_1 \succeq i_2 \ \wedge \ j_2 \succeq j_1)$$
$$\text{for all } (i_1, j_1), (i_2, j_2) \in A^* \cap M \ , \tag{49}$$

and we define, for all $j \in L_0$, the set of all nodes in H_0 for which the edge to which they are incident under M has a lower value of the order function than the edge to which j is incident under M, i.e. the set

$$H_0^{(j)} := \{i \in H_0 : order[(i, j^*)] < order[(i^*, j)]$$

$$\text{with } i^* \in H_0, j^* \in L_0 \text{ given by } (i, j^*), (i^*, j) \in M\}$$

If $I_{C0} \neq \varnothing$ there exists a path-splitting or a cycle-splitting solution with respect to S if and only if the polyhedron P_I^* defined by the (in)equalities

Flow out of source:

$$\sum_{j \in H_0} x_j^{(A)} = 1 \tag{33}$$

Flow into sink:

$$\sum_{j \in L_0} x_j^{(B)} = 1 \tag{34}$$

Flow balance for higher nodes of all edges of M:

$$x_i^{(A)} - y_{i,j} = 0$$

$$\text{for all } i \in H_0, j \in L_0 \text{ with } (i,j) \in A^* \cap M \tag{35}$$

Flow balance for lower nodes of all edges of M:

$$y_{i,j} - \sum_{k \in H_0^{(j)} \setminus H_{c(j)}} z_{j,k} - x_j^{(B)} = 1$$

$$\text{for all } j \in L_0, i \in H_0 \text{ given by } (i,j) \in A^* \cap M \tag{50}$$

Exactly one edge used from each cycle in I_{C0}:

$$\sum_{(i,j) \in A_k^*} y_{i,j} = 1 \text{ for all } k \in I_{C0} \tag{37}$$

At most one edge used from P and from each cycle in $I_{C1} + I_{C2}$:

$$\sum_{(i,j) \in A_k^*} y_{i,j} \leq 1 \text{ for all } k \in I_{C1} + I_{C2} + \{d+1\} \tag{38}$$

At least one edge from P or from a cycle in $I_{C1} + I_{C2}$
(excluding edges of cycles in I_{C1} a node of which is adjacent to i_0):

$$\sum_{(i,j) \in (\bigcup_{k \in I_{C1}+I_{C2}+\{d+1\}} A_k^*) \setminus \{(i,j) \in \bigcup_{k \in I_{C1}} A_k^* : (i,i_0) \in A_G\}} y_{i,j} \geq 1 \tag{39}$$

contains a point with coordinates

$$x_i^{(A)} \in \{0,1\} \text{ for all } i \in H_0 \ ,$$

$$x_j^{(B)} \in \{0,1\} \text{ for all } j \in L_0 \ ,$$

$$y_{i,j} \in \{0,1\} \text{ for all } (i,j) \in M \cap A^* \ ,$$

$$z_{j,k} \in \{0,1\} \text{ for all } j \in L_0 \text{ and } k \in H_0^{(j)} \setminus H_{c(j)} \tag{51}$$

Proof. We first have to show that we can construct an order function $order[.]$ with the required properties. We recall that for greedy matchings M, for all $(i_1, j_1) \in M$ there exists no edge $(i_2, j_2) \in M$ with $dg(i_1) > dg(i_2)$ and $dg(j_1) > dg(j_2)$ unless (i_1, j_1) and (i_2, j_2) belong to the same cycle of path of the twin-induced structure (Proposition 79). In the light of Proposition 93, condition (a) this implies that we can discard the edge (i_2, j_2) in such a case as it will not help us with finding an alternating cycle that meets conditions (a) to (c). Accordingly, all remaining edges $(i_k, j_k) \in M$ can be arranged such that

$$i_k \succeq j_k \text{ for } 0 \leq k \leq \frac{|M|}{2} - 1 \ , \tag{52}$$

$$i_0 \succeq i_1 \succeq ... \succeq i_{\frac{|M|}{2}-2} \succeq i_{\frac{|M|}{2}-1} \text{ , and} \tag{53}$$

$$j_{\frac{|M|}{2}-1} \succeq j_{\frac{|M|}{2}-2} \succeq ... \succeq j_1 \succeq j_0 \ , \tag{54}$$

i.e. the pre-order $(A^* \cap M, \succeq)$ defined by virtue of

$$(i_1, j_1) \succeq (i_2, j_2) :\Leftrightarrow (i_1 \succeq i_2 \ \wedge \ j_2 \succeq j_1)$$

$$\text{for all } (i_1, j_1), (i_2, j_2) \in A^* \cap M$$

is total. We construct the order function such that it is monotonically non-decreasing with respect to this pre-order, which ensures (49). If we assign different values of $order[.]$ when

$$(i_1, j_1) \succeq (i_2, j_2) \text{ and } (i_1, j_1) \preceq (i_2, j_2)$$

$$\text{for } (i_1, j_1), (i_2, j_2) \in A^* \cap M, i_1 \neq i_2, j_1 \neq j_2 \ ,$$

the function is injective.

$\Longrightarrow$: We compare the polyhedron with the one in Proposition 94. In contrast to Proposition 94, this theorem does not include constraints (40) and (41), the variables $z_{j,k}$ have been defined on a *subset* of those arcs (j, k) that we used in Proposition 94, and the sum in constraints (36), the only constraints to include the variables $z_{j,k}$, has been restricted to the new set of arcs on which we have defined the variables $z_{j,k}$ (now constraints (50)). Leaving out constraints (40) and (41) obviously does not have an impact on the existence of a suitable point in P_I^* if the existence of such a point has been established by virtue of Proposition 94.

Regarding constraints (50) and the variables $z_{j,k}$, we recall Corollary 68, according to which there exists an alternating T-cycle relative to a certain matching M if and only if there exists also an alternating T-cycle that satisfies conditions (45) to (48). Hence, if there is a path- or cycle-splitting solution, Proposition 93 automatically implies the existence of an alternating

T-cycle that fulfils (45) to (48). Therefore, the arcs

$$(j_{m-1}, i_m) \text{ for } i = 1, 2, ..., \frac{|T|}{2} - 1$$

have the property

$$i_m \preceq i_{m-1} \text{ with } i_{m-1} \in H_0 \text{ given by } (i_{m-1}, j_{m-1}) \in A^* \cap M \ ,$$

which implies with (48)

$$order[(i_{m-1}, j_{m-1})] > order[(i_m, j_m)] \tag{55}$$

(apart from possible permutations of pairs of arcs (i_1, j_1), $(i_2, j_2) \in A^* \cap M$ with

$$(i_1, j_1) \succeq (i_2, j_2) \text{ and } (i_1, j_1) \preceq (i_2, j_2) \text{ for } i_1 \neq i_2, j_1 \neq j_2,$$

which do not have an impact on the existence of a solution). The restricted set of arcs on which the variables $z_{j,k}$ have been defined still includes all arcs (j_{m-1}, i_m) that satisfy (55) for a given arc $(i_{m-1}, j_{m-1}) \in A^* \cap M$ represented by $y_{i_{m-1}, j_{m-1}}$ (provided i_m does not belong to the same cycle or path as j_{m-1}). Also, all arcs (j_{m-1}, i_m) with property (55) are still included in the sum in constraints (50), again provided i_m does not belong to the same cycle or path as j_{m-1}.Consequently, constraints (50) and the set on which we have defined the variables $z_{j,k}$ do not restrict unduly the set of points in P_I^*, and the existence of a suitable point in P_I^* follows from Proposition 94.

$\Longleftarrow$: We have a feasible point of the polyhedron P_I^* with values for the variables according to (50). The set

$$T := \{(i, j) \in M \cap A^* : y_{i,j} = 1\}$$

fulfills condition (a) of Proposition 93 due to constraints (37) and (38), condition (b) due to constraints (37), and condition (c) due to constraints (39). If we can show that there exists an alternating T-cycle relative to M, Proposition 93 implies the existence of a path- or cycle-splitting solution.

Being a subset of the matching, T fulfils properties (52) to (54) and therefore (46) to (48). The point in P_I^* contains only the flow from the source to the sink and no cycles because (49) makes sure that it is impossible to return, via the arcs represented by the variables $y_{i,j}$ and $z_{j,k}$, to an arc $(i, j) \in T$ that has already been traversed. Therefore, the edges in T and the edges in

$$Z := \{(j, k) : z_{j,k} = 1\}$$

provide us with a path as in condition (45). Because of (46) and (47), the node i_0 must be a dominating all nodes incident to edges in T, and due to (48) the node $j_{\frac{|T|}{2}|-1}$ must be a member of a maximal clique of all nodes incident to edges in T. Hence we can connect the endnodes of

the path (45), obtain an alternating T-cycle, and can apply Proposition 93 to arrive at a path- or cycle-splitting solution. ■

Solving the integer problem of Theorem 96 could be a difficult task. In contrast to the polyhedron of a minimum cost flow problem, we have constraints (37) and (38). Fortunately, the following corollary holds.

Corollary 97 *(Complexity of deciding on the existence of irreducible solutions)*
If there exists no structure-preserving solution, the existence of irreducible solutions of an MSSP can be shown in polynomial time.

Proof. We will (a) note that the number of rows and columns of the A-matrix defining the polyhedron P_I^* of the flow problem in Theorem 96 is polynomial in the number of nodes of the MSSP and (b) show that the A-matrix is totally unimodular. It is well-known (see Schrijver, 1986, chapter 19, for example) that the latter property implies that all extreme points of P_I^* are integral as the right-hand sides of the constraints in Theorem 96 are integral, too. Consequently, the existence of a point in P_I^* satisfying (51) is equivalent to the existence of a basic solution of a Linear Programming problem. If we take into account that the maximum absolute value of all numbers in the A-matrix and the b-vector of our polyhedral description is equal to 1 for all instances of the problem, we can conclude that we can decide on the existence of a basic point of P_I^* in a time depending polynomially only on the number of rows and columns of the A-matrix (see Schrijver 1986, chapter 15, for example).

(a) The number of nodes is $2n \geq 6$. (6 is the minimal number of nodes that we need for at least one cycle and one path.) The number of rows (in the order of constraints (33), (34), (35), (37), (38), (39), (50)) is

$$\begin{aligned} &1 + 1 + |H_0| + |L_0| + |I_{C0} + I_{C1} + I_{C2}| + 1 + 1 \\ &\leq 1 + 1 + (n-1) + (n-1) + \tfrac{2n-2}{4} + 1 + 1 \leq 3n. \end{aligned}$$

The number of columns (in the order of the variables given in (51)) is

$$\begin{aligned} &|H_0| + |L_0| + |M| + |\{(k,j) \in A^* : j \in L_0,\, k \in H_0^{(j)} \backslash H_{c(j)}\}| \\ &\leq \tfrac{n-1}{2} + \tfrac{n-1}{2} + n - 1 + n(n-1) = (n+2)(n-1). \end{aligned}$$

(b) For proving total unimodularity, we use a criterion that goes back to Ghouila-Houri (1962) [see also Nemhauser, Wolsey (1999), Theorem 2.7, part III]. According to this criterion, a matrix is totally unimodular if and only if for every subset $J \subseteq C = \{1, 2, ..., c\}$ of the columns of the matrix, there exists a partition $J_1 + J_2 = J$ such that the difference between the sum of the columns in J_1 (which we will denote Σ_{J_1}) and the sum of the columns in J_2 (denoted Σ_{J_2}) is a column vector with all entries being in the set $\{-1, 0, 1\}$. It is well known that the node-arc incidence matrix of a directed graph is totally unimodular (see Nemhauser/ Wolsey 1999, chapter III.3, for example).

The submatrix given by the rows representing the constraints (33), (34), (35), and (50) is a node-arc incidence matrix and, consequently, for every subset J of the columns there exists

a partition $J_1 + J_2 = J$ with the property given by the criterion by Ghouila-Houri. However, we need more precise information for being able to make a conclusion that takes into account also the rest of the constraints. For facilitating this, we introduce additional arcs in our flow problem such that we have a larger number of variables $z_{j,k}$, namely

$$z_{j,k} \in \{0,1\} \text{ for all } j \in L_0 \text{ and } k \in H_0^{(j)} \backslash H_{c(j)} \ ,$$

and replace constraint (50) by

$$y_{i,j} - \sum_{k \in H_0} z_{j,k} - x_j^{(B)} = 1$$

$$\text{for all } j \in L_0, i \in H_0 \text{ given by } (i,j) \in A^* \cap M \ . \tag{56}$$

Due to the fact that a totally unimodular matrix will, as a trivial consequence of Ghouila-Houri's theorem, remain totally unimodular when we delete a column, it suffices to prove total unimodularity for our new flow problem. The resulting A-matrix is illustrated in Figure 17, with all entries not explicitly given being zero. The symbol I represents the unit matrix, the symbol e a row vector with 1s in each entry, and the symbol m a row vector with 1s except one entry being 0. Block A contains all arcs from the source node and to the sink node. The columns of block B correspond to all arcs $(i,J) \in A^* \cap M$ (which are the arcs associated with the variables $y_{i,j}$), with the arcs having been sorted such that all arcs representing one cycle (or the path) of the twin-constrained structure appear in consecutive rows of the matrix. Block C represents all arcs (j,k) from lower nodes $j \in L_0$ to higher nodes $k \in H_0$, which have been modelled by the variables $z_{j,k}$. (Note that the block C entries for constraints (56) are the only entries of the matrix that differ from the matrix representing the polyhedron P_I^* in Theorem 96.) The variables $z_{j,k}$ have been arranged such that arcs emanating from nodes of the same cycle (or the path) appear in consecutive rows of the matrix. The number of the constraints that the columns represent are given in the first row of Figure 17.

For the remainder of the proof, we will denote, for a given set S of indices of columns of the A-matrix, by Σ_S the column that is the sum of the columns whose indices are in S. Moreover, a column (or a specified set of rows of a column) with all entries being in $\{-1,0,1\}$ and with the sum of these entries also being in $\{-1,0,1\}$ is said to have property (Q).

For a given subset J of columns, we partition the (indices of the) columns into the sets K, L, M, depending on whether the columns belong to block A, B, or C of the A-matrix, respectively, i.e. we have

$$J = K + L + M.$$

	B L O C K A								B L O C K B										Block C			
Variables	**Arcs from source**				**Arcs to sink**				**Arcs fr. l_{C0}**			**Arcs fr. l_{C2}**			**Arcs fr. l_{C1}**			**Arcs**	**Arcs cycles**			**path**
	$X_i^{(A)}$				$X_j^{(B)}$				C_1	…	$C_{\|lCo\|}$	C_1	…	$C_{\|lC2\|}$	C_1	…	$C_{\|lC1\|}$	P	C_1	…	C_d	P
Source (33)	-e …-e	…	-e … -e	-e																		
Sink (34)					e … e	…	e … e	e														
Flow balance	I … I								-I … -I										e … e			
for higher		. . .										-I … -I								. . .		
nodes (35)			I … I												-I … -I				e … e			
				I														-I				e … e
Flow balance					-I … -I				I … I										-l . . . -l	. . .	-l . . . -l	-l . . . -l
for lower						. . .						I … I										
nodes (56)							-I … -I								I … I							
								-I										I				
Exactly one arc per cycle in l_{C0} (37)									e … e													
At most one arc per cycle in l_{C2} (38)												e … e										
At most one arc per cycle in l_{C1} (38)															e … e							
… fr. path (38)																		e				
At least … (39)												e	. . .	e	m	. . .	m	e				

Figure 17: A-matrix of the modified flow problem

We start by dealing with the columns of block B. We partition the set L into sets L_1 and L_2 such that for every row of the constraints (37) and (38) an equal number of columns with entry 1 goes into both sets − provided that the number of columns with a 1 in constraints (37) and (38) is even for a given row. If this number is odd, we allow one of the sets L_1 and L_2 to be assigned one more column beyond an even distribution such that, after having assigned all columns in L to either L_1 or L_2, we have

$$| \, |L_1| - |L_2| \, | \leq 1$$

(If the columns represent arcs from I_{C1}, the number of columns with a 1 in constraints (37) and (38) is odd for a given row *and* a column with a 0 in the row of constraint (39) is among the columns, we will assign such a column with a 0 after all other columns have been assigned to either L_1 or L_2.) We observe that our assignment ensures that the column $\Sigma_{L1} - \Sigma_{L2}$ has an entry in $\{-1, 0, 1\}$ in the row of constraint (39). Moreover, the column $\Sigma_{L1} - \Sigma_{L2}$ has property (Q) for all rows of constraints (35), (56), (37) and (38).

A similar assignment can be made for block C. We partition the set M into sets M_1 and M_2 such that for each row of the constraints (35) an equal number of columns with entry 1 goes into both sets − provided that the number of columns with a 1 in constraints (35) is even for a given row. If this number is odd, we allow one of the sets M_1 and M_2 to be assigned one more column beyond an even distribution, however such that we have

$$| \, |M_1| - |M_2| \, | \leq 1$$

after having assigned to M_1 and M_2 *all* columns in M. We observe that the column $\Sigma_{M1} - \Sigma_{M2}$ has property (Q) in all rows related to constraints (35).

We now turn our attention to the lower half of the rows of block C, i.e. to those rows that are given by constraints (50). For all columns in M that have the same entries in the rows given by constraints (35) we calculate the difference between columns in M_1 and columns in M_2. The resulting column differences all have property (Q). We now exchange pairs of columns from M_1 and M_2 (a pair consists of one column from M_1 and one column from M_2) that have the additional property that the columns that form a pair have the same entries in the rows given by constraints (35). Such an exchange step allows us to change one of the column differences just calculated, such that one entry changes from $+1$ to -1 and another entry changes from -1 to $+1$. By carrying out a sufficient number of these exchange steps, it is possible to arrive at a new partition $M = M_1 + M_2$ such that $\Sigma_{M1} - \Sigma_{M2}$ has property (Q) in all entries related to constraints (56). We observe that our exchange steps are neutral with respect to property (Q) of $\Sigma_{M1} - \Sigma_{M2}$ in all rows related to constraints (35). This means that we have constructed a partition $M = M_1 + M_2$ such that property (Q) of $\Sigma_{M1} - \Sigma_{M2}$ applies to the column $\Sigma_{M1} - \Sigma_{M2}$ for both the rows related to constraints (35) and the rows related to constraints (56).

We turn to block A and partition the set K into sets K_1 and K_2 such that $\Sigma_{K1} - \Sigma_{K2}$ has property (Q) for the rows related to constraints (33), (34), (35) and (56).

We define $J_i := K_i + L_i + M_i$ for $i = 1, 2$ and have a partition $J = J_1 + J_2$. Since

$$\Sigma_{J1} - \Sigma_{J2} = \Sigma_{K1} - \Sigma_{K2} + \Sigma_{L1} - \Sigma_{L2} + \Sigma_{M1} - \Sigma_{M2},$$

the column $\Sigma_{J1} - \Sigma_{J2}$ has entries in $\{-3, -2, -1, ..., 3\}$. Due to the existence of property (Q) for the columns $\Sigma_{K1} - \Sigma_{K2}$, $\Sigma_{L1} - \Sigma_{L2}$, and $\Sigma_{M1} - \Sigma_{M2}$ for all rows but the row of constraint (39), we can, by moving columns from J_1 to J_2 or vice versa and/or by means of exchange steps as defined above, change the entries in $\Sigma_{J1} - \Sigma_{J2}$ such that the criterion by Ghoulia-Houri is satisfied. ■

In this chapter, we started from a modified matching M on the threshold graph underlying the $MSSP$. On the basis of the cardinality of such a matching, we could already make a decision on whether or not certain instances of the MSSP are feasible, with the case $|M| = n-1$ remaining open.

Proceding with our analysis, we introduced the concept of the twin-induced structure of a matching M for the case $|M| = n-1$ and had a look at how we could change the matching $|M|$ if its twin-induced structure does not directly provide us with a feasible solution of the MSSP. We addressed the most basic case, in which we change only d edges from the original matching $|M|$, with d being the number of cycles of the twin-induced structure of $|M|$. It turned out that, due to the specific structure of threshold graphs, a straight forward (polynomial-time) criterion exists that allows us to find out if changing d edges could lead to a feasible solution, namely a structure-preserving solution.

Going further, we analyzed also the case of changing $d + 1$ edges from the matching M, which results in what we call path-splitting and cycle-splitting solutions. We distinguished, in the previous section, 20 path-splitting solutions and 26 cycle-splitting solutions, and it turned out that there exist 2 genuine types of path-splitting solutions and that cycle-splitting solutions can be classified by reducing them to 4 genuine types. We provided a unifying perspective on these 6 irreducible solutions and developed, in this section, a Linear Programming problem that can tell us in polynomial time whether one of these 6 irreducible solutions exists. Apart from the practical relevance of Theorem 96 and Corollary 97 for solving the MSSP, these two statements are also of theoretical relevance because they enable us to find a polynomial-time algorithm for a more complex class of solutions.

In principle, we could proceed our analysis in the same fashion in which we have conducted it so far. If we did so, the next natural step would consist in discussing the cases of feasible solutions of the MSSP that arise from changing $d + 2$ edges from the original matching M. Figure 16(b) above illustrates a case in which $d + 2$ edges from the original matching have been changed to construct a solution that involves both a path- and a cycle-split. We can expect, however, that changing $d + 2$ edges would lead to even more subtypes of solutions than the 46 (genuine and non-genuine) subtypes we examined when changing $d + 1$ edges. As this does not look very promising under the aspects of both practical relevance and theoretical elegance, it seems to be wise to stop our discussion here. - The more so as changing edges is equivalent to calculating a new matching, and the more edges we change and the more cases we have to examine the smaller will be the computational advantage that can be gained by changing only a few edges from a given matching instead of calculating a new matching from scratch.

For these reasons, we will finish the present chapter here and will present, as a summary of the results of this chapter, a simple heuristic for deciding on the feasibility or infeasibility of a MSSP. It will turn out in chapter 9 by means of computational experiments that this heuristic is surprisingly efficient for deciding on the feasibility of a large percentage of instances.

7.8 A heuristic for the MSSP ($MSSPH$)

We start with a threshold graph G with node set $N_G = \{1, 2, ..., 2n\}$, $n \geq 2$, and an edge set E_G defined by virtue of weights given by $v : N_G \longrightarrow N_G$ and a threshold $\alpha \geq 0$, and a twin-node function $b : N_G \longrightarrow N_G$. The output of our heuristic is either a certificate of feasibility, a certificate of infeasibility, or no certificate. The general principle underlying this heuristic consists in the idea of moving from the most general and computationally least expensive procedures to more specific procedures that take more computational time. Our heuristic will proceed in 10 steps.

In step [01] of our algorithm, we use three direct tests to identify infeasible instances.

First, an instance can be recognised as infeasible if more than half of the nodes + 1 are members of a maximal stable set, i.e.

$$|\bigcup_{i=0}^{\lfloor m/2 \rfloor} D_i| > n + 1.$$

Second, we can discard an instance as infeasible if there exists a node such that both the node and its twin node have no neighbours, i.e.

$$\exists i \in N_G : \{i, b(i)\} \subseteq D_0.$$

Third, an instance is certainly infeasible if more than 2 nodes have no neighbours, i.e.

$$|D_0| > 2,$$

with D_0 being the set in the degree partition that contains all nodes of degree 0.

We have seen in this chapter that different matchings provide us with different opportunities of finding a feasible solution on the basis of the twin-induced structure of a matching. Therefore, our heuristic calculates two different matchings. As we will need at a later stage of the heuristic a greedy modified matching for finding irreducible path- and cycle-split solutions, provided that more basic approaches were not sufficient for finding a feasible solution, we calculate in step [02] the type of matching that differs the most from a greedy modified matching, namely a modest modified matching M according to $MTGMA_{\min}$.

In step [03] we evaluate this matching. If $|M| < n-1$, our instance is infeasible; if $|M| = n$, our instance is feasible (Proposition 80). If $|M| = n - 1$, we check whether one of the two unmatched nodes is an element of a maximal clique of G. If this is the case, our instance must be feasible as this directly implies a structure-preserving solution on the basis of Theorem 82.

As structure-preserving solutions can also arise without the previous condition being fulfilled, we calculate in step [04] the twin-induced structure S of M for further evaluation.

In step [05] we analyse the twin-induced structure $S(N, E)$. If it consists only of a path P, i.e. $N_S = P$, our instance must trivially be feasible. Otherwise, we check according to Theorem 82 if there exists a structure-preserving solution, i.e. we calculate the unmatched node i_0 with the largest neighbourhood and check for all cycles C_k, $1 \leq k \leq d$ whether i_0 is adjacent to one node from each cycle. Due to the vicinal preorder of our underlying threshold graph our instance is feasible with a structure preserving solution if and only if

$$\min_{1 \leq k \leq d} \max_{j \in C_k} v(i_0) + v(j) \geq \alpha,$$

see also Remark 83.

We have come now with our matching M as far as possible and calculate a greedy modified matching M' on the basis of $MTGMA_{\max}$ in step [06].

In principle, we could now compare our new matching with the old one. If $M = M^{'}$, we can conclude that the graph G allows only for one single matching and that our instance must be infeasible. (Otherwise we would have found a feasible solution in step [05].) However, as a threshold graph that has only one single perfect matching is not a very likely type of instance, we will, for saving computational time when dealing with all other types of instances, not compare our two matchings. (If, however, in a particular practical setting there is some additional information available that suggests that the number of threshold graphs with only one single perfect matching is rather large, it would be advisable to include in the heuristic a step that compares the two matchings.)

Step [07] of our algorithm calculates the twin-induced structure $S'(N, E)$ of M'.

Step [08] is identical with step 5 for the twin-induced structure $S'(N, E)$.

In step [09], if we still have not found a feasible solution, we use Theorem 96 in conjunction with Corrollary 97 to find out, by trying to calculate a basic solution of the LP problem, if the polyhedron P_I^* is empty or if our matching M' allows the construction of an irreducible path- or cycle-splitting solution.

In sum our algorithm looks as follows:

Algorithm 98 *(MSSPH - MSSP Heuristic)*

Let $G(N, E)$ be an undirected graph with the (even) set of nodes
$N_G = \{1, 2, ..., 2n\}$, neighbourhoods $N(i)$ for all $i \in N_G$,
and $b : N_G \to N_G$ a twin-node function.

[01] *If* $|\bigcup_{i=0}^{\lfloor m/2 \rfloor} D_i| > n + 1$ *then INFEASIBLE, STOP.*
If $|D_0| > 2$ *then INFEASIBLE, STOP.*
If $\exists i \in N_G : \{i, b(i)\} \subseteq D_0$ *then INFEASIBLE, STOP.*

[02] *Run $MTGMA_{\min}$, output: M.*

[03] *If* $|M| < n-1$ *then INFEASIBLE, STOP.*
If $|M| = n$ *then FEASIBLE, STOP.*
[$|M| = n-1$ *]:*
$i_0 := \underset{k \in \{i \in N_G : \nexists j \in N_G : (i,j) \in M\}}{\arg\max}\ dg(k)$
If $i_0 \in K_{\max}$ *then FEASIBLE, STOP.*

[04] *Compute* $S(N, E)$.

[05] *If* $N_S = P$ *then FEASIBLE, STOP.*
If $\min\limits_{1 \le k \le d} \max\limits_{j \in C_k} v(i_0) + v(j) \ge \alpha$ *then FEASIBLE, STOP.*

[06] *Run* $MTGMA_{\max}$*, output:* M'

[07] *Compute* $S'(N, E)$.

[08] *If* $N_{S'} = P$ *then FEASIBLE, STOP.*
If $\min\limits_{1 \le k \le d} \max\limits_{j \in C'_k} v(i_0) + v(j) \ge \alpha$ *then FEASIBLE, STOP.*

[09] *If* $P_I^* \neq \varnothing$ *then FEASIBLE.*
STOP.

Computational results of $MSSPH$ will be presented in chapter 9. It will turn out that even without implementing Step [09] our algorithm show a surprisingly efficient behaviour. For the worst-case situation, including Step [09], the following proposition holds.

Proposition 99 *(Complexity of MSSPH)*
$MSSPH$ runs in polynomial time.

Proof. Directly from Corrollary 97 as trying to find a basic solution for the polyhedron P_I is the computationally most expensive operation. ■

8 Recognising twin-constrained Hamiltonian threshold graphs

In the previous chapter we developed a heuristic that looks for specific properties of a given instance of the MSSP and, if possible, draws a conclusion on whether or not this instance is feasible. In this context, three of the final sections of that chapter examined in detail the case of non structure-preserving solutions. The present chapter builds on the insights that we gained by analysing non structure-preserving solutions and sets out to generalize them. In doing so, we will develop an efficient algorithm that can decide on the feasibility of a given instance of the MSSP in all possible cases.

We will begin with a section that motivates and explains our approach by looking at the lessons we can learn from our analysis in the previous chapter.

8.1 Motivation

We have seen in the previous chapter that the threshold graph structure of an MSSP is so strong that a small number of criteria is sufficient to decide whether there exists, among a large number of possible matchings, one matching that is a feasible solution of the MSSP. We could show, for example, that the criterion we found for structure-preserving solutions of type 1 also implies testing for possible structure-preserving solutions of type 2, that looking at possible structure-preserving and path-splitting solutions of type 1 and type 2 is sufficient for deciding whether there exists a path-splitting solution at all, and that the existence of a cycle-splitting solution can be decided solely on the basis of the existence of a structure-preserving solution or one of four specific types of cycle-splitting solutions. In other words: the threshold graph property of an MSSP seems to be so strong that, for deciding on the feasibility of an MSSP, it is not necessary to analyse all possible matchings that could lead to a solution of the MSSP, but that it seems to be sufficient to look only at some possible ways of constructing a solution. Moreover, it turned out that, for all types of solutions we discussed, a decision on the existence of such a solution can be made in polynomial time. All in all, this gives hope that the MSSP might be a problem that can be solved in polynomial time.

Additionally, we have seen that a polynomial-time algorithm for the specific types of non structure-preserving solutions we discussed was possible mainly due to the particular structure of greedy matchings. Given this structure, we could capitalise on our strong result for the existence of alternating T-cycles relative to a greedy matching by gluing together parts of the twin-induced structure to construct a solution to the MSSP. This suggests that we should seek an approach that might benefit further from the specific structure of greedy matchings and their implications for the existence of alternating T-cycles. The more so as we observed in Proposition 60 that alternating T-cycles can be used to generate all other matchings on the basis of a given matching.

For making use of this property, however, we need a perfect matching. A setting based on perfect matchings can be achieved by adding to the nodes of our MSSP 2 dominating nodes that are twin-nodes to each other. We know from chapter 4 (Lemma 38) that a threshold graph remains a threshold graph if we add a dominating node. Looking for a twin-constrained Hamiltonian *path* for solving our original MSSP will then be equivalent to searching for a twin-constrained Hamiltonian *cycle* on the new graph. In other words: instead of directly trying to solve an MSSP, we might benefit from focussing our attention on the problem of recognising twin-constrained Hamiltonian threshold graphs. (Note that, while a matching of a cardinality of at least $|M| = n - 1$ is a necessary condition for a twin-constrained Hamiltonian path (with a perfect matching being a sufficient condition, see Proposition 80), a perfect matching is only a necessary condition for a twin-constrained Hamiltonian cycle.)

Let us illustrate what happens when we transform our MSSP into the problem of recognising twin-constrained Hamiltonian threshold graphs by reconsidering the case of a path-splitting solution of type 1 (Figure 15). Introducing a pair of dominating twin-nodes u, v would lead to a situation in which the two unmatched nodes e and p (the endnodes of the path) could have been matched (with the dominating nodes, for example). As a result, the twin-induced structure of the matching would consist of 3 cycles (one of which is the former path). A twin-constrained Hamiltonian cycle (i. e the counterpart of a solution to the MSSP in this case with the new nodes) could be constructed by splitting the cycle based on the former path twice and gluing the two remaining cycles to it. Provided that the dominating nodes have been matched with the two formerly unmatched nodes (whether this is the case depends on the implementation of $MGTMA$ that we use), the cycle $i - j - k - l$ could again be glued between the nodes m and p, and the cycle $a - b - c - d$ could now be glued between e and one of the two additional dominating nodes, say u. As a consequence, this solution could be constructed by means of *two* alternating T-cycles, namely the cycles with

$$T_1 = \{(e, b), (a, u)\} \text{ and } T_2 = \{(k, l), (h, m)\}.$$

Removing the pair of dominating nodes will finally lead us to our solution of the MSSP

In the general case, the setting is more complex. The number of alternating T-cycles that we might need for constructing a solution might be higher. Also, the cycles of the twin-induced structure might have to be split into several more parts and glued together such that several former cycles might be nested into each other. Figure 18 shows a more complex setting for a twin-induced structure consisting of 3 cycles C_1, C_2 and C_3, represented by the thinner solid, dashed and dotted lines, with the solid lines being edges between twin-nodes, and the dashed and dotted lines being edges from the matching. For constructing a solution, the 3 cycles have been split into parts such that

$$C_1 = C_{1A} + C_{1B},\ C_2 = C_{2A} + C_{2B} + C_{2C}$$
$$\text{and } C_3 = C_{3A} + C_{3B}.$$

These parts of the cycle have then been glued together by means of the thick dashed edges to form the twin-constrained Hamiltonian cycle

$$C_{1A} + C_{2A} + C_{3A} + C_{1B} + C_{2B} + C_{3B} + C_{2C}$$

Proceeding from the original matching, this twin-constrained Hamiltonian cycle can be seen as having been generated by alternating cycles that consist of the thicker dashed edges and those edges from the original matching that do not take part in the twin-constrained Hamiltonian cycle (i.e. the edges e_1, e_2, ..., e_7). This means that the solution of the twin-constrained Hamiltonian cycle problem has been generated by three alternating T-cycles with

$$T_1 = \{e_1, e_2\},\ T_2 = \{e_3, e_4, e_5\},\ \text{and}\ T_3 = \{e_6, e_7\}.$$

Again, we can arrive at a twin-constrained Hamiltonian path, i.e. a solution of the MSSP, by removing our additional pair of dominating nodes (which have not been specified in this example) from the twin-constrained Hamiltonian cycle obtained.

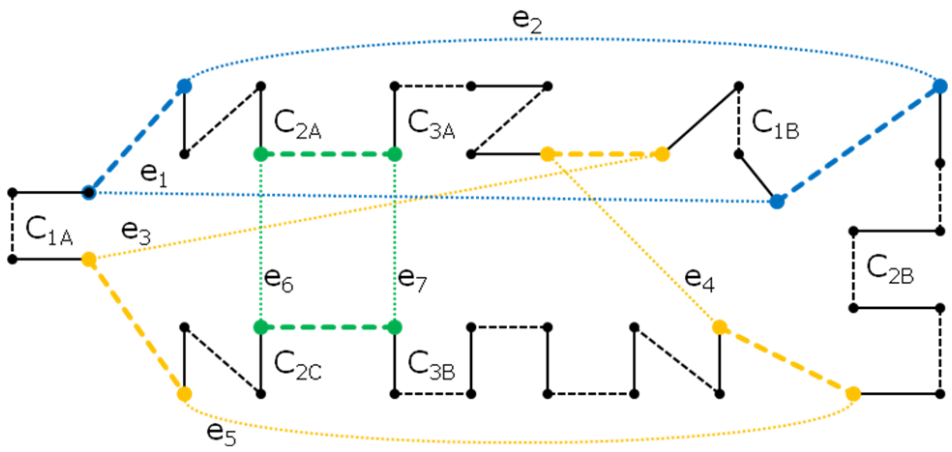


Figure 18: Constructing a solution by means of alternating T-cycles

Following the approach just outlined, the remainder of this chapter will address (solely) the problem of recognising twin-constrained Hamiltonian threshold graphs and in doing so, we will capitalise on our results on alternating T-cycles and the insights on non-structure-preserving solutions from the previous chapter.

The following section introduces the concept of "patching graphs" and provides a necessary criterion for the existence of twin-constrained Hamiltonian threshold graphs, and the subsequent third section demonstrates that we can find a similar sufficient criterion. As the criterion presented in these two sections is based on the existence of a certain family of alternating T-cycles, we can consider it as the generalized version of Proposition 93.

In the previous chapter, we proceed from Proposition 93 by looking for an algorithm to determine whether the type of alternating T-cycle required by Proposition 93 actually exists, which led to Theorem 96. In an analogous fashion, our necessary and sufficient criterion from the second and third sections of this chapter will require us to develop an algorithm that finds a certain family of alternating T-cycles if such a family exists. This task of generalizing Theorem

96 will be achieved in the fourth section. Finally, the fifth section presents, as a summary of the results of this chapter, a complete polynomial-time algorithm for deciding on the question of whether a given instance of an MSSP is feasible.

8.2 Patching graph and a necessary criterion for twin-constrained Hamiltonicity

For formulating our necessary criterion, we introduce the concept of the "patching graph". A patching graph can be seen to contain information that indicates whether there is room for gluing together certain alternating T-cycles to form a twin-constrained Hamiltonian cycle.

Definition 100 *(Patching graph of a twin-induced structure)*

Let $G(N,E)$ be a threshold graph with a twin-node function b, the family $(C_d)_{d\in D}$ the twin-induced structure of a perfect modified matching M on G, and $(T_q \subseteq M)_{q\in Q}$ a disjunct family such that there exist alternating T_q-cycles relative to M for all $q \in Q$. The graph $P(N,E)$ defined by $N_P := D$ and

$$E_P := \{(i,j) \in D \times D : (k_1,l_1),(k_2,l_2) \in T_q \text{ for some } (k_1,l_1) \in C_i, (k_2,l_2) \in C_j \text{ and } q \in Q\}$$

is called the patching graph of (C_d) relative to (T_q).

Example 101 *In the case of Figure 18, the patching graph is a complete graph defined on the node set $D = \{1,2,3\}$.*

Remark 102 *(1) There exists a concept called "subtour patching" in the literature on the Travelling Salesman Problem that has some similarity with our concept of a patching graph. This concept, however, is based on a hypergraph over the node set of the graph under investigation, while we have defined a patching graph with respect to the cycles of the twin-constrained structure (Gilmore, Lawler, Shmoys 1986). Accordingly, these two concepts should not be confused.*

(2) Note that the patching graph of (C_d) relative to (T_q) is certainly invariant under sorting the edges of an alternating T-cycle (cf. Definition 67). We will use this property in the following sections when trying to construct certain patching graphs.

On the basis of the concept of the patching graph, establishing a necessary criterion for the existence of a twin-constrained Hamiltonian cycle is straight-forward.

Theorem 103 *(Patching graph for twin-constrained Hamiltonian threshold graphs)*

Let $G(N, E)$ be a threshold graph with twin function b and M_0 a perfect modified matching such that its twin-induced structure is a twin-constrained Hamiltonian cycle C_0 on G. Furthermore, let $(C_d)_{d\in D}$ be the twin-induced structure of an (arbitrary) perfect modified matching $M \neq M_0$ on G. Then there exists a (well defined) family $(K_q)_{q\in Q}$ of alternating cycles by virtue of

$$\bigcup_{q\in Q} K_q = M_0 \oplus M \tag{57}$$

and a disjunct family $(T_q \subseteq M)_{q\in Q}$ defined by

$$T_q := K_q \cap M \text{ for all } q \in Q \tag{58}$$

such that K_q is an alternating T_q-cycle relative to M for all $q \in Q$ and the patching graph of (C_d) relative to (T_q) is connected.

Proof. Because of $M \neq M_0$, we have $M_0 \oplus M \neq \varnothing$. Due to the fact that M_0 and M are perfect matchings, every node that is adjacent to any edge in $M_0 \oplus M$ has exactly two different mates, namely one with respect to M_0 and one with respect to M. Hence $M_0 \oplus M$ consists only of (disjunct) cycles, and $(K_q)_{q\in Q}$ is well defined. As every consecutive pair of edges consists of one edge from M_0 and one edge from M, definition (57) indeed ensures that K_q is an alternating T_q-cycle relative to M for all $q \in Q$

It remains to be shown that the patching graph is connected. We proceed from assuming the contrary. If the patching graph P of (C_d) relative to (T_q) is not connected, there exist disjunct sets $D_1, D_2 \subsetneqq D$ such that for all $d_1 \in D_1$ and $d_2 \in D_2$, the graph P does not contain a path from d_1 to d_2, i.e. there exists no set T_q that contains edges from both C_{d_1} and C_{d_2}. Moreover, as $T_q \subseteq M$, none of the sets T_q has an element that connects two nodes one of which is incident to an edge in C_{d_1}, and the other one of which is incident to an edge in C_{d_2}. Consequently, for all $d_1 \in D_1$ and $d_2 \in D_2$ there is no alternating cycle K_q that contains an edge between a node incident to an edge in C_{d_1} and one incident to an edge in C_{d_2}.

We widen our perspective to all edges of the twin-constrained Hamiltonian cycle C_0. As M_0 is a modified matching, these are given by the partition

$$\begin{aligned} C_0 &= M_0 + \{(i, b(i)) : i \in N_G\} \\ &= M_0 \backslash M + M_0 \cap M + \{(i, b(i)) : i \in N_G\}. \end{aligned}$$

By the definition of the twin-induced structure $(C_d)_{d\in D}$, the set

$$\{(i, b(i)) : i \in N_G\}$$

does not contain any edge that connects a node incident to an edge in C_{d_1} with a node incident to an edge in C_{d_2}. As M is the matching that $(C_d)_{d\in D}$ is based on, the set $M_0 \cap M$ does not contain any such edge either. Finally, the same statement can be made for the set $M_0 \backslash M$ because it contains only edges from the cycles $(K_q)_{q\in Q}$, which we have already analyzed above. Therefore, the graph induced by C_0 has at least two components, namely the subgraphs

induced by the nodes incident to edges in

$$\sum_{d_1 \in D_1} C_{d_1} \text{ and } \sum_{d_2 \in D_2} C_{d_2},$$

respectively, i.e. C_0 cannot be a twin-constrained Hamiltonian cycle. Accordingly, our assumption that the patching graph of (C_d) relative to (T_q) is not connected does not hold. ■

We slightly reformulate this theorem with the following corollary.

Corollary 104 *(Twin-constrained Hamiltonian threshold graph - necessary condition)*
Let $G(N, E)$ be a threshold graph with twin-node function b, and $(C_d)_{d \in D}$ the twin-induced structure of a perfect modified matching M on G. There exists a twin-constrained Hamiltonian cycle on G only if either $(C_d)_{d \in D}$ is a twin-constrained Hamiltonian cycle or there exists a disjunct family $(T_q \subseteq M)_{q \in Q}$ with alternating T_q-cycles relative to M for all $q \in Q$ such that the patching graph of (C_d) relative to (T_q) is connected.

Proof. Assume there exists a twin-constrained Hamiltoninan cycle on G that is not $(C_d)_{d \in D}$. Hence there exists a perfect modified matching $M_0 \neq M$ such that its twin-induced structure is a twin-constrained Hamiltonian cycle. Then, according to the preceding theorem, the family $(T_q \subseteq M)_{q \in Q}$ defined by (57) and (58) has the properties that the corollary calls for. ■

Note that we could obtain our necessary criterion in a rather simple, straight-forward way. We did not even have to refer to any specific properties of threshold graphs or the underlying matching M to establish our theorem. In fact, Theorem 103 and Corollary 104 hold for twin-constrained Hamiltonian cycles on all types of graphs. (The only reason why we explicitly mentioned threshold graphs in these statements is that we defined, in earlier chapters, some of the technical terms we made use of − such as "modified matching" and "twin-induced structure" − only for the case of threshold graphs.)

The intuitive, straight-forward character of the statements above, however, should not mislead us into thinking that twin-constrained Hamiltonicity can be recognised easily. The following simple example demonstrates that our theorem, in its present formulation, is certainly not suitable as a sufficient criterion.

Example 105 *Consider a graph $G(N, E)$ with*

$$N_G = \{a, b, c, d, e, f, g, h\} \text{ and}$$
$$E_G = \{(a, b), (c, d), (e, f), (g, h), (a, c), (b, e), (d, f)\},$$

further a twin-node function $b : N_G \to N_G$ given by

$$b(a) = c,\ b(b) = d,\ b(e) = g, \text{ and } b(f) = h.$$

The set

$$M := \{(a, b), (c, d), (e, f), (g, h)\} \subset E_G$$

is a (modified) matching on G.
Consequently, for

$$T := \{(a,b),(c,d),(e,f)\} \subset M$$

the path

$$a - b - e - f - d - c - a$$

is an alternating T-cycle relative to M. As the twin-induced structure of M consists of two cycles, namely

$$C_1 = \{a,b,c,d\} \text{ and } C_2 = \{e,f,g,h\},$$

our alternating T-cycle gives rise to the patching graph $P(D,E)$ with

$$D = \{1,2\} \text{ and } E = \{(1,2)\},$$

which is connected. The graph G, however, is obviously not twin-constrained Hamiltonian with respect to b.

8.3 Sufficient criterion for twin-constrained Hamiltonicity of threshold graphs

We know from the previous section that, for finding out whether, for a given twin-node function b, a threshold graph is twin-constrained Hamiltonian with respect to b, we only have to look for disjunct families of alternating T-cycles that give rise to a connected patching graph. We will now set out to show that the existence of an appropriate disjunct family of alternating T-cycles with a connected patching graph is also sufficient for the twin-constrained Hamiltonicity of a threshold graph. However, as Example 105 demonstrated, arriving at a sufficient condition for twin-constrained Hamiltonicity needs stronger assumptions. It will turn out in this section that it is sufficient to start from a greedy matching on a threshold graph.

Our proof will be constructive. The general principle underlying our proof is to "glue" together, successively, all cycles (C_d) of the twin-induced structure of M such that we eventually arrive at the desired twin-constrained Hamiltonian cycle. In order to achieve this, we will use the edges from the alternating T_q-cycles that we have at our disposal as linking elements between the cycles of the twin-induced structure. The proof proceeds by successively taking into account all members of the family (T_q).

We begin with a Lemma that allows us to construct, on the basis of one alternating T_q-cycle, a twin-constrained alternating cycle on our graph that connects all nodes that are part of all those cycles of the twin-induced structure of which the alternating T_q-cycle contains an edge.

Lemma 106 *(Twin-constrained cycles from alternating T_q-cycles)*

Let $G(N,E)$ be a threshold graph with twin-node function b, the family $(C_d)_{d\in D}$ the twin-induced structure of a perfect greedy modified matching M on G and $(T_q \subseteq M)_{q\in Q}$ a disjunct family such that there exist alternating T_q-cycles relative to M for all $q \in Q$. Then for every set $T_q \subseteq M$ there exists a perfect modified matching M^* on G such that its twin-induced structure $(C^*_d)_{d\in D^*}$ contains a twin-constrained cycle that connects all nodes incident to edges in those sets C_d that T_q contains an element of.

Proof. Let $T_q \subseteq M$ be an arbitrary member of the family $(T_q)_{q \in Q}$. As the matching M is greedy, there exists a sorted alternating T_q-cycle relative to M according to Corollary 68 and we can assume in the following that the nodes and edges of T_q are given according to (45), (46), (47) and (48). (In the case of ties, choose an arbitrary order that remains fixed in the following. Also note that we can disregard in the following, due to Proposition 79, those edges that do not fulfil the degree property of Proposition 51 that is necessary for Corollary 68.)

STEP 1: Examine the edges $(i_k, j_k) \in T_q$ in the order given by (45), starting from $k = 0$. If the list of edges in T_q begins with several edges from the same cycle C_d, jump to the last one of these, which we denote by $(i_{k_1}, j_{k_1}) \in T_q$. We proceed in the list of edges until one of the following two cases occurs:

Case (1.1): We come to an edge (i_{k_2}, j_{k_2}) that is an element of one of the cycles in the family (C_d) that any other edge in

$$\{(i_{k_1}, j_{k_1}), (i_{k_1+1}, j_{k_1+1}), ..., (i_{k_2-1}, j_{k_2-1})\}$$

is also an element of. Then we define

$$\widehat{T}_{q,1} := \{(i_{k_1}, j_{k_1}), (i_{k_1+1}, j_{k_1+1}), ..., (i_{k_2-1}, j_{k_2-1})\}.$$

Case (1.2): We arrive at the end of the list, i.e. at the edge $(i_{|T|-1}, j_{|T|-1})$, without having found an edge (i_{k_2}, j_{k_2}) of *Case* (1.1). Then we define

$$\widehat{T}_{q,1} := \{(i_{k_1}, j_{k_1}), (i_{k_1+1}, j_{k_1+1}), ..., (i_{|T|-1}, j_{|T|-1})\}.$$

For both case (1.1) and case (1.2), let

$$\widehat{D}_{q,1} := \{d \in D : (i,j) \in C_d \text{ for some } (i,j) \in \widehat{T}_{q,1}\}$$

be the set of the indices of all cycles from the twin-induced structure (C_d) of M that the edges in $\widehat{T}_{q,1}$ are an element of, and

$$\widehat{C}_{q,1} := \sum_{d \in \widehat{D}_{q,1}} C_d$$

the set of all edges from all cycles in (C_d) which are represented (via their indices) in $\widehat{D}_{q,1}$.

We know because of Corollary 70 that the canonical alternating $\widehat{T}_{q,1}$-cycle exists, which we designate as $\widehat{K}_{q,1}$. On this basis, we define

$$\overline{C}_{q,1} := (\widehat{C}_{q,1} - \widehat{T}_{q,1}) + (\widehat{K}_{q,1} - \widehat{T}_{q,1}).$$

Let us examine this set. The nodes that are incident to edges in the set $\overline{C}_{q,1}$ are exactly those nodes that are incident to edges in $\widehat{C}_{q,1}$ due to the fact that the nodes incident to edges in $\widehat{T}_{q,1}$ are the same nodes that are incident to edges in $\widehat{K}_{q,1} - \widehat{T}_{q,1}$. Moreover, every node incident to an edge in $\overline{C}_{q,1}$ has exactly two neighbours in the subgraph induced by $\overline{C}_{q,1}$, namely first its twin node (by virtue of $\widehat{C}_{q,1}$) and second either an edge from

$$\overline{C}_{q,1} \cap M = \overline{C}_{q,1} \cap \widehat{C}_{q,1} - \{(i,j) \in N_G \times N_G : i = b(j)\}$$

or an edge from $\widehat{K}_{q,1} - \widehat{T}_{q,1}$. Consequently, $\overline{C}_{q,1}$ consists only of disjunct cycles. Finally, $\overline{C}_{q,1}$ is connected because of the very construction of $\widehat{K}_{q,1}$ as an alternating $\widehat{T}_{q,1}$-cycle that contains one edge from each cycle in $\widehat{C}_{q,1}$. *In sum:* $\overline{C}_{q,1}$ *is a twin-constrained cycle that connects all nodes incident to edges in* $\widehat{C}_{q,1}$.

If *Case* (1.2) has arisen above, we continue directly with *STEP* 3. Otherwise we proceed with further examining the edges in the list T_q, starting from $k_2 + 1$. If the remainder of T_q,

namely the set

$$\{(i_{k_2+1}, j_{k_2+1}), (i_{k_2+1}, j_{k_2+1}), ..., (i_{|T|-1}, j_{|T|-1})\},$$

does not contain any edge $(i_{k_3}, j_{k_3}) \notin \overline{C}_{q,1}$ with $k_3 \geq k_2 + 1$, we continue also with $STEP$ 3.

$STEP$ 2: Let $k_3 \geq k_2 + 1$ be the smallest number such that $(i_{k_3}, j_{k_3}) \notin \overline{C}_{q,1}$. We continue to examine the edges in T_q in the order given in (21) and distinguish two cases.

$Case$ (2.1): $k_3 = |T| - 1$. We define the set

$$\widehat{T}_{q,2} := \{(i_{k_3-1}, j_{k_3-1}), (i_{k_3}, j_{k_3})\}.$$

$Case$ (2.2): $k_3 < |T| - 1$. We further examine the list until, analogously to $STEP$ 1, one of the following two cases occurs:

$Case$ (2.2a): We come to an edge (i_{k_4}, j_{k_4}) that is an element of one of the cycles in the family (C_d) that any other edge in

$$\{(i_{k_3}, j_{k_3}), (i_{k_3+1}, j_{k_3+1}), ..., (i_{k_4-1}, j_{k_4-1})\}$$

is also an element of, or we arrive at an edge $(i_{k_4}, j_{k_4}) \in \overline{C}_{q,1}$. Then we define

$$\widehat{T}_{q,2} := \{(i_{k_3-1}, j_{k_3-1}), (i_{k_3}, j_{k_3}), ..., (i_{k_4-1}, j_{k_4-1})\}$$

$Case$ (2.2b): We arrive at the end of the list, i.e. at the edge $(i_{|T|-1}, j_{|T|-1})$, without having found an edge (i_{k_4}, j_{k_4}) that has one of the two properties in case (2.2a). Then we define

$$\widehat{T}_{q,2} := \{(i_{k_3-1}, j_{k_3-1}), (i_{k_3}, j_{k_3}), ..., (i_{|T|-1}, j_{|T|-1})\}.$$

For both case (2.1) and case (2.2), again analogously to $STEP$ 1, but without including the edge that $\widehat{T}_{q,2}$ has in common with $\widehat{T}_{q,1}$, we define

$$\widehat{D}_{q,2} := \{d \in D : (i,j) \in C_d \text{ for some } (i,j) \in [\widehat{T}_{q,2} - \{(i_{k_3-1}, j_{k_3-1})]\}$$

as the set of all cycles of the twin-induced structure of which $\widehat{T}_{q,2} \backslash \widehat{T}_{q,1}$ contains an edge, and

$$\widehat{C}_{q,2} := \sum_{d \in \widehat{D}_{q,2}} C_d$$

as the set of all edges of all cycles of the twin-induced structure of which $\widehat{T}_{q,2} \backslash \widehat{T}_{q,1}$ contains an edge.

Designating the canonical alternating $\widehat{T}_{q,2}$-cycle as $\widehat{K}_{q,2}$, which exists because of Corollary 70, we define

$$\overline{C}_{q,2} := \overline{C}_{q,1} - \{(i_{k_3-1}, j_{k_3-1})\} + (\widehat{C}_{q,2} - \widehat{T}_{q,2}) + (\widehat{K}_{q,2} - \widehat{T}_{q,2}).$$

Let us examine this set.

(1) The nodes incident to $\overline{C}_{q,2}$ are exactly those nodes that are incident to edges in $\widehat{C}_{q,2}$ and in $\overline{C}_{q,1}$ due to the facts that (a) the set $\overline{C}_{q,1}$ is a cycle and removing one edge (here: $\{(i_{k_3-1}, j_{k_3-1})\}$) will not remove any nodes incident to edges in this set and (b) the set $\widehat{C}_{q,2}$ consists of cycles and, by definition, the set $\widehat{T}_{q,2}$ contains exactly one edge from each of these cycles, and (c) the set $(\widehat{K}_{q,2} - \widehat{T}_{q,2})$ does not contain edges between nodes that are not incident to edges in either $\overline{C}_{q,1}$ or $\widehat{C}_{q,2}$.

(2) Every node incident to an edge in $\overline{C}_{q,2}$ has exactly two neighbours in the subgraph induced by $\overline{C}_{q,2}$, namely first its twin node (by virtue of $\widehat{C}_{q,2}$ and $\overline{C}_{q,1}$) and second either an edge from

$$\overline{C}_{q,2} \cap M = [\overline{C}_{q,1} \backslash \widehat{T}_{q,2} + \overline{C}_{q,2} \cap \widehat{C}_{q,2}] - \{(i,j) \in N_G \times N_G : i = b(j)\}$$

or an edge from $\widehat{K}_{q,2} - \widehat{T}_{q,2}$. Consequently, $\overline{C}_{q,2}$ consists only of disjunct cycles.

(3) The set $\overline{C}_{q,2}$ contains a connected subgraph because of the very construction of $\widehat{K}_{q,2}$ as an alternating $\widehat{T}_{q,2}$-cycle that contains one edge from each cycle in $\widehat{C}_{q,2}$ and from the cycle $\overline{C}_{q,1}$. *In sum:* $\overline{C}_{q,2}$ *is a twin-constrained Hamiltonian cycle that connects all nodes incident to edges in* $\widehat{C}_{q,2}$ and in $\overline{C}_{q,1}$.

If either case (2.1) or case (2.2*b*) has arisen above, we have reached the end of the list T_q and proceed directly with $STEP$ 3. Otherwise, i.e. iff case (2.2*a*) had occurred, we proceed by further examining the remaining edges in the list T_q, starting from $k_2 + 1$ with k_2 being "redefined" by $k_2 := k_4$. If the remainder of T_q, namely the set

$$\{(i_{k_2+1}, j_{k_2+1}), (i_{k_2+1}, j_{k_2+1}), ..., (i_{|T|-1}, j_{|T|-1})\},$$

does not contain any edge $(i_{k_3}, j_{k_3}) \notin \overline{C}_{q,1}$ with $k_3 \geq k_2 + 1$, we continue also with $STEP$ 3. Otherwise, we go to the beginning of $STEP$ 2, "redefine" the set $\overline{C}_{q,1}$ as $\overline{C}_{q,1} := \overline{C}_{q,2}$, and repeat the process of $STEP$ 2 as often as it is necessary to reach the end of the list T_q, i.e. to continue with $STEP$ 3.

$STEP$ 3: As a result of the preceding steps, we have successively built a twin-constrained Hamiltonian cycle that connects all nodes that are incident to any of the cycles C_d that T_d contains an edge from, i.e. the twin-constrained Hamiltonian cycle $\overline{C}_{q,2}$ contains all nodes from

$$\{i \in N_G : (i,j) \in C_d \text{ and } T_q \cap C_d \neq \varnothing \text{ for some } j \in N_G,\ d \in D\}.$$

Then the set

$$M^* := \overline{C}_{q,2} + \bigcup_{\substack{d \in D \text{ with} \\ C_d \cap T_q = \varnothing}} C_d - \{(i,j) \in N_G \times N_G : i = b(j)\},$$

i.e. the set of all edges from the twin-constrained cycle $\overline{C}_{q,2}$ and from those cycles of the twin-induced structure of the matching M that did not take part in constructing $\overline{C}_{q,2}$, but without the edges between pairs of twins, is the matching we were looking for. ■

The next Lemma shows that we can enlarge a twin-constrained cycle we have constructed on the basis of one (or more) alternating T_q-cycles such that it includes also all nodes incident to edges in those cycles of the twin-induced structure of which an additional alternating T_q-cycle contains an edge − provided that the additional alternating T_q-cycle has at least one edge in common with one of those cycles of the twin-induced structure of which one of the other T_q-cycles contains an edge.

Lemma 107 *(Enlarging twin-constrained cycles)*

Let $G(N,E)$ be a threshold graph with twin-node function b, $(C_d)_{d\in D}$ the twin-induced structure of a perfect greedy modified matching M on G and $(T_q \subseteq M)_{q\in Q}$ a disjunct family such that there exist alternating T_q-cycles relative to M for all $q \in Q$. If there exists, for some subset $\overline{Q} \subsetneqq Q$, a perfect modified matching M^* on G such that its twin-induced structure $(C^*_d)_{d\in D^*}$ contains an alternating twin-constrained cycle $\overline{C}_{\overline{Q}}$ with

$$\{i \in N_G : (i,j) \in \overline{C}_{\overline{Q}} \text{ for some } j \in N_G\}$$
$$= \{i \in N_G : (i,j) \in \bigcup_{\substack{d\in D \text{ with} \\ \exists q\in\overline{Q}: C_d\cap T_q\neq\varnothing}} C_d\}$$

i.e. a twin-constrained cycle that connects all nodes incident to edges in those sets C_d that some T_q with $q \in \overline{Q}$ contains an element of, then there exists, for all $q^* \in Q - \overline{Q}$ with $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$, also a perfect modified matching M^{**} on G such that its twin-induced structure $(C^{**}_d)_{d\in D^{**}}$ contains a twin-constrained alternating cycle $\overline{C}_{\overline{Q}+\{q^*\}}$.

Proof. If we have $\overline{C}_{\overline{Q}} \supseteq T_{q^*}$ for a given $q^* \in Q - \overline{Q}$ with $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$, the matching $M^{**} := M^*$ trivially has the desired properties due to $\overline{C}_{\overline{Q}} = \overline{C}_{\overline{Q}+\{q^*\}}$ in this case. Hence we can assume $T_{q^*}\backslash\overline{C}_{\overline{Q}} \neq \varnothing$ in the following. Furthermore, we can assume without loss of generality that $M^*\backslash\overline{C}_{\overline{Q}} \subsetneqq M$. (If this is not the case, we can generate such a matching $M^*\backslash\overline{C}_{\overline{Q}}$ by constructing a greedy modified matching on the subgraph of G induced by $M^*\backslash\overline{C}_{\overline{Q}}$.) Hence there exists a sorted alternating T_{q^*}-cycle relative to M^* according to Corollary 68 and we can assume in the following that the nodes and edges of T are given according to (45), (46), (47), and (48). (In the case of ties, choose an arbitrary order that remains fixed in the following. Again, as in the previous lemma, note that we can disregard in the following, due to Proposition 79, those edges that do not fulfil the degree property of Proposition 51 that is necessary for Corollary 68.)

STEP 1: We choose a maximal cardinality subset of edges

$$\widehat{T}_{q^*,1} := \{(i_{k_1}, j_{k_1}), (i_{k_1+1}, j_{k_1+1}), ..., (i_{k_2}, j_{k_2})\} \subseteq T_{q^*}$$

such that $\widehat{T}_{q^*,1}$ contains exactly one edge from $\overline{C}_{\overline{Q}}$ (which is possible because of $\overline{C}_{\overline{Q}}\cap T_{q^*} \neq \varnothing$, but $T_{q^*}\backslash\overline{C}_{\overline{Q}} \neq \varnothing$) and no two edges from the same member of the family $(C_d)_{d\in D^*}$. By defining

$$\widehat{D}_{q^*,1} := \{d \in D : (i,j) \in C_d \text{ for some } (i,j) \in \widehat{T}_{q^*,1}\}$$

as be the set of the indices of all cycles from the twin-induced structure (C_d) of M that the edges in $\widehat{T}_{q^*,1}$ are an element of,

$$\widehat{C}_{q^*,1} := \sum_{d\in\widehat{D}_{q^*,1}} C_d + \overline{C}_{\overline{Q}}$$

as the set of all edges from the cycle $\overline{C}_{\overline{Q}}$ and all cycles in (C_d) which are represented (via their indices) in $\widehat{D}_{q^*,1}$, and the set $\widehat{K}_{q^*,1}$ as the canonical alternating $\widehat{T}_{q^*,1}$-cycle, which exists because of Corollary 70, we arrive, by "gluing" together the cycles in (C_d) and the cycle $\overline{C}_{\overline{Q}}$ in the same fashion as in *STEP* 1 in the proof of the previous lemma, at a *twin-constrained alternating cycle*

$$\overline{C}_{q^*,1} := (\widehat{C}_{q^*,1} - \widehat{T}_{q^*,1}) + (\widehat{K}_{q^*,1} - \widehat{T}_{q*,1}).$$

that connects all nodes incident to edges in $\widehat{C}_{q^*,1}$ and $\overline{C}_{\overline{Q}}$.

STEP 2: We proceed by examining the list of nodes analogously to *STEP* 2 in the proof of the previous lemma, starting from node $k_2 + 1$. However, when we have reached the end of the list, we do not proceed to *STEP* 3 of the proof of the previous lemma, but examine (unless $k_1 = 0$) the first part of the list of the edges in T_q, starting from (i_{k_1-1}, j_{k_1-1}) in a backwards(!) order. Once we have reached the top of the list of nodes, we have constructed, successively, a twin-constrained Hamiltonian cycle that connects all nodes that are incident to any of those cycles C_d from which some T_d with $d \in \overline{Q}$ contains an edge, and those nodes that are incident to edges in all cycles C_d from which T_{q^*} contains an edge, i.e. our twin-constrained alternating cycle $\overline{C}_{\overline{Q}+\{q^*\}}$ contains all nodes from

$$\{i \in N_G : (i,j) \in (\bigcup_{\substack{d \in D \text{ with} \\ \exists q \in \overline{Q}: C_d \cap T_q \neq \varnothing}} C_d) \cup (\bigcup_{\substack{d \in D \text{ with} \\ C_d \cap T_{q^*} \neq \varnothing}} C_d) \text{ for some } j \in N_G\}.$$

Then the set

$$M^{**} := \overline{C}_{\overline{Q}+\{q^*\}} + \bigcup_{\substack{d \in D \text{ with} \\ q \in Q - \overline{Q} - \{q^*\} \text{ and } C_d \cap T_q = \varnothing}} C_d - \{(i,j) \in N_G \times N_G : i = b(j)\}$$

is the matching that the lemma claims to exist. ■

We are now prepared to give a proof of a sufficient condition for the twin-constrained Hamiltonicity of a threshold graph. The proof proceeds by induction. It starts with constructing a twin-constrained alternating cycle on the basis of one alternating T_q-cycle according to Lemma 106 and successively adds, by means of Lemma 107, edges from those cycles of the twin-induced structure of which other alternating T_q-cycles contain an edge.

Theorem 108 *(Recognizing twin-constrained Hamiltonian threshold graphs - sufficient criterion)*

Let $G(N, E)$ be a threshold graph with twin-node function b, the family $(C_d)_{d \in D}$ the twin-induced structure of a perfect greedy modified matching M on G and $(T_q \subseteq M)_{q \in Q}$ a disjunct family such that there exist alternating T_q-cycles relative to M for all $q \in Q$. If the patching graph of (C_d) relative to (T_q) is connected then there exists a twin-constrained Hamiltonian cycle on G.

Proof. We choose some set $T_{q_0} \subseteq M$, for which, according to Lemma 106, there exists a perfect modified matching M^* on G such that its twin-induced structure $(C^*_d)_{d \in D^*}$ contains a twin-constrained alternating cycle that connects all nodes incident to edges in those sets C_d that T_{q_0} contains an element of. Assume that, for $\overline{Q} := \{q_0\}$, we can find a $q^* \in Q - \overline{Q}$ that fulfils the condition of Lemma 107. Then Lemma 107 guarantees that there exists a twin-constrained cycle $\overline{C}_{\{q_0,q^*\}}$ that connects all nodes incident to edges in those sets C_d that T_{q_0}

or T_{q^*} contains an element of, and we might be able to apply Lemma 107 again, this time to the set $\overline{Q} := \{q_0, q^*\}$. If it is possible to successively apply Lemma 107, i.e. if there always exists a $q^* \in Q - \overline{Q}$ with $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$ for any given subset $\overline{Q} \subsetneqq Q$, we will finally arrive at a matching M^{**} the twin-induced structure of which contains a twin-constrained cycle $\overline{C}_Q$, i.e. a cycle that connects all nodes that are incident to edges in those sets C_d that some member of the family $(T_q)_{q\in Q}$ contains an element of.

As the patching graph P of $(C_d)_{d\in D}$ relative to $(T_q)_{q\in Q}$ is connected, there are no isolated nodes in $N_P = D$. Therefore, by definition of the patching graph, there exists, for every $d \in D$, a member of $(T_q)_{q\in Q}$ that contains an element of C_d. Consequently, the cycle $\overline{C}_Q$ connects all nodes that are incident to some edge in some member of the family $(C_d)_{d\in D}$, i.e., by definition of the twin-induced structure of a matching, the twin-constrained cycle $\overline{C}_Q$ is a twin-constrained Hamiltonian cycle on G.

It remains to show that there always exists a $q^* \in Q - \overline{Q}$ with $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$ for any given subset $\overline{Q} \subsetneqq Q$. For a certain $\overline{Q} \subsetneqq Q$, let $D_{\overline{Q}} \subseteq D$ be a subset of nodes of the patching graph such that $(C_d)_{d\in D_{\overline{Q}}}$ is the family of those cycles of which some T_q with $q \in \overline{Q}$ contains an edge, i.e.

$$D_{\overline{Q}} = \{d \in D : C_d \cap T_q \neq \varnothing \text{ for some } q \in \overline{Q}\}.$$

We distinguish between two cases.

Case (1): If $D_{\overline{Q}} = D$, we already have "glued together" all cycles of the twin-induced structure of the matching, i.e. the cycle $\overline{C}_{\overline{Q}}$ is twin-constrained Hamiltonian. Hence we have $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$ for all $q^* \in Q - \overline{Q}$. (Alternatively, we could stop the process of successively applying Lemma 107 here.)

Case (2): If $D_{\overline{Q}} \subsetneqq D$, the connectedness of the patching graph ensures that there exists a node $d_1 \in D - D_{\overline{Q}}$ that is a neighbour of some $d_2 \in D_{\overline{Q}}$. Then, according to the definition of the patching graph, there exists a set T_{q^*} that contains edges from both C_{d_1} and C_{d_2}. Because of $d_1 \in D - D_{\overline{Q}}$, the cycle C_{d_1} has not been used yet when constructing $\overline{C}_{\overline{Q}}$. Due to the way in which Lemma 106 and Lemma 107 construct a twin-constrained cycle $\overline{C}_{\overline{Q}}$, this implies $q^* \in Q - \overline{Q}$. Moreover, $d_2 \in D_{\overline{Q}}$ implies that the cycle C_{d_2} has already been used in constructing $\overline{C}_{\overline{Q}}$. Because the family $(T_q \subseteq M)_{q\in Q}$ is disjunct, T_{q^*} contains an edge from C_{d_2} that has not been discarded in the process of constructing $\overline{C}_{\overline{Q}}$. Consequently, we have $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$. ■

8.4 Constructing suitable families of alternating T_q-cycles

We know from Theorem 108 in the previous section that starting from a perfect modified greedy matching we can find a twin-constrained Hamiltonian cycle if we can find a disjunct family (T_q) with alternating T_q-cycles that gives rise to a connected patching graph. We also know, from Corollary 104, section 8.2, that, for concluding that there does not exists a twin-constrained Hamiltonian cycle, we can start from any perfect modified matching, which could well be a greedy matching, and "only" have to find out whether there exists no disjunct family (T_q) with

alternating T_q-cycles that would give rise to a connected patching graph. The question that remains for solving the recognition problem for twin-constrained Hamiltonian threshold graphs is: How can we either find a suitable family of alternating T_q-cycles or know that such a family does not exist?

Regarding the second part of the question, let us recall Remark 102(2), which notes that the patching graph does not change if we analyse *canonical sorted* alternating T_q-cycles. This implies that we can restrict our effort to look for a suitable family (T_q) to all families of canonical sorted alternating T_q-cycles relative to a perfect greedy modified matching. In other words: we can use the set of all families of canonical sorted alternating T_q-cycles relative to a perfect greedy modified matching as the domain of a matching generator in the sense of Proposition 60. In the light of Corollary 104 and Theorem 108 this means that the set of canonical sorted alternative cycles relative to a perfect greedy modified matching already contains all information necessary for determining whether or not a certain threshold graph with a twin-node function is twin-constrained Hamiltonian. We will now use this insight for our search for a suitable family of alternating T_q-cycles.

Algorithm 109 *(Family construction algorithm - FCA)*

INPUT: a sorted list of edges $(i_k, j_k) \in M$ with M being a perfect modified matching on a threshold graph G with nodes $N_G = \{1, 2, ..., 2n\}$ and the edges from M being sorted in the sense of (46) *and* (47)*, (provided they fulfill the degree property of Proposition* 51*), neighbourhoods $N(i)$ for $i \in N_G$, and a list $C[(i_k, j_k)]$ that indicates for each edge $(i, j) \in M$ the number $d \in D$ of the cycle of the twin-induced structure of which it is an element.*

[01] *(number of alternating T_q-cycle to be constructed)*
$q^* := 0$
[02] *(set of edges for alternating T_q-cycle)*
$T_q := \{\}$ *for all* $1 \leq q \leq \frac{n}{2}$
[03] *(set of indices of those cycles of the twin-induced structure that have an edge in T_q)*
$S_q := \{\}$ *for all* $1 \leq q \leq \frac{n}{2}$
[04] *(edge from matching that is being under consideration)*
$k := 1$
[05] *REPEAT*
[06] *(find the first edge of next set T_q to be constructed)*
WHILE $(k < n-1) \wedge [j_k \notin N(i_{k+1}) \vee C[(i_k, j_k)] = C[(i_{k+1}, j_{k+1})]]$
DO $k := k + 1$
[07] *IF* $[j_k \in N(i_{k+1}) \wedge C[(i_k, j_k)] \neq C[(i_{k+1}, j_{k+1})]]$ *THEN*

[08] *(include first edge into a new set T_q)*
$q^* := q^* + 1; T_{q^*} := \{(i_k, j_k)\}; S_{q^*} := \{C[(i_k, j_k)]\}$

[09] *(complete the set T_q)*
WHILE $(k < n) \wedge (j_k \in N(i_{k+1})) \wedge (C[(i_{k+1}, j_{k+1})] \notin S_q)$
DO $k := k + 1; T_{q^*} := T_{q^*} + \{(i_k, j_k)\}; S_{q^*} := S_{q^*} + \{C[(i_k, j_k)]\}$

[10] *END IF*

[11] *(go to next edge)*
$k := k + 1$

[12] *UNTIL* $(k \geq n)$, *STOP.*

OUTPUT: $q^* = 0$, *or sets* T_q *and* S_q *for* $1 \leq q \leq q^*$

We will give a brief overview of how the algorithm proceeds. After preliminarily setting some variables, the algorithm starts searching from the highest edge until it reaches the second last edge or until it finds an edge the lower node of which is adjacent to the higher node of the succeeding edge in the sorted list, with both edges not being from the same cycle of the twin-induced structure of the matching (line [06]). Once FCA has found a pair of succeeding edges from different cycles (line [07]), it starts a new set T_q and adds to it the first edge of the pair (line [08]). It continues by adding all succeeding edges to the set T_q as long as we have not reached the last edge of our sorted list, the lower node of the current edge is adjacent to the higher node of the succeeding edge and the succeeding edge does not belong to a cycle of the twin-induced structure that is already represented in T_q (line [09]). If composing the set T_q has been completed, the algorithm proceeds to the next edge in our sorted list in order to check whether this edge is suitable for beginning a new set T_{q+1} (line [11]) – unless we have already examined the second last or last edge of our sorted list (line [12]).

The following theorem states that FCA generates sets that provide us with a disjunct family of alternating T_q-cycles, i.e. the algorithm FCA provides a link between the result of our matching algorithm $MTGMA_{\max}$ and the structure that we require to draw conclusions about twin-constrained Hamiltonicity on the basis of Corollary 104 and Theorem 108.

Theorem 110 *(Sufficiency of a solution by FCA)*
Let $G(N, E)$ be a threshold graph with twin-node function b, and $(C_d)_{d \in D}$ the twin-induced structure of a perfect modified matching M on G. Unless FCA terminates with $q^* = 0$, it terminates with a disjunct family $(T_q \subseteq M)_{q \in Q}$ such that there exist alternating T_q-cycles relative to M for all $q \in Q$.

Proof. If and only if $q^* \neq 0$ when the algorithm terminates, FCA has constructed sets $(T_q)_{q \in Q}$ with $Q := \{1, 2, ..., q^*\}$. According to the way in which the edges of the matching M are sorted and because of the way in which FCA operates, the nodes incident to edges in $T_q \subseteq M$ can be represented by the set

$$N_{T_q} := \{i_{k_q}, j_{k_q}, i_{k_q+1}, j_{k_q+1}, ...,$$
$$i_{k_q+|T_q|-2}, j_{k_q+|T_q|-2}, i_{k_q+|T_q|-1}, j_{k_q+|T_q|-1}\}.$$

Due to lines [07] and [09] the algorithm makes sure that, for each $q \in Q$, we have $j_k \in N(i_{k+1})$ for all nodes $j_k \in N_{T_q}$ with $k_q \leq k \leq k_q + |T_q| - 2$. Hence there exists a path

$$i_{k_q} - j_{k_q} - i_{k_q+1} -, j_{k_q+1} - ...$$
$$-i_{k_q+|T_q|-2} - j_{k_q+|T_q|-2} - i_{k_q+|T_q|-1} - j_{k_q+|T_q|-1} \quad \text{for all } q \in Q.$$

Because the edges in $T_q \subseteq M$ are sorted in the sense of (46) and (47), the node i_{k_q} must be dominating all nodes incident to edges in T_q. We choose the node $j_{k_q^*}$ with the largest neighbourhood among all nodes $j_k \in N_{T_q}$ with $k_q \leq k \leq k_q + |T_q| - 2$, which must be a member of a maximal clique of all nodes incident to edges in T_q. Hence there exists a matching dominated P-path

$$i_{k_q} - j_{k_q} - i_{k_q+1} - j_{k_q+1} - ... - i_{k_q^*-1} - j_{k_q^*-1} - i_{k_q^*} - j_{k_q^*}$$

with

$$P := \{(i_{k_q}, j_{k_q}), (i_{k_q+1}, j_{k_q+1}), ..., (i_{k_q^*-1}, j_{k_q^*-1}), (i_{k_q^*}, j_{k_q^*})\}$$

to which we can apply Theorem 63, i. e. there exists, for all $q \in Q$, an alternating T_q-cycle relative to M. The family $(T_q)_{q\in Q}$ is disjunct because the algorithm uses every edge $(i_k, j_k) \in M$, $1 \leq k \leq |M|$, only once. ■

On the basis of the preceding theorem we can build, unless FCA terminates with $q^* = 0$, the patching graph of the twin-induced structure of M relative to the family (T_q). If the patching graph is connected, we know there exists and can construct a twin-constrained Hamiltonian cycle according to Theorem 108. The next theorem states that, if TCA terminates with $q^* = 0$ or if the family (T_q) found by FCA does not provide us with a patching graph that is connected, we do not have to look any further and can conclude that there exists no twin-constrained Hamiltonian cycle on G .

Theorem 111 *(Necessity of a solution by FCA)*

Let $G(N, E)$ be a threshold graph with twin-node function b, and $(C_d)_{d\in D}$ the twin-induced structure of a perfect greedy modified matching M on G with $|D| > 1$. If FCA terminates with $q^* = 0$ or if the patching graph of $(C_d)_{d\in D}$ relative to the family $(T_q \subseteq M)_{q\in Q}$ with which FCA terminates is not connected, there exists no disjunct family $(\widehat{T}_q \subseteq M)_{q\in Q}$ relative to which the patching graph of the twin-induced structure of M is connected.

Proof. We first recall that, due to the degree property of a greedy matching (Proposition 51), an ordering of the edges of the matching in the sense of (22) and (23) also complies with (26). (Note that we can disregard here, due to Proposition 79 those edges that do not fulfill the degree property of Proposition 51.) As a consequence, given the order in the sense of (22), (23) and (26) and with $k_0 := 0$ and $k_p := |M| - 1$, the (relevant) edges of a greedy matching show the following structure:

$$i_{k_0} - j_{k_0} - i_{k_0+1} - j_{k_0+1} - ... - i_{k_1} - j_{k_1} \ ,$$

$$i_{k_1+1} - j_{k_1+1} - ... - i_{k_2} - j_{k_2} \ ,$$

$$i_{k_2+1} - j_{k_2+1} - ... - i_{k_3} - j_{k_3} \ , \quad ... \quad ,$$

$$i_{k_{(p-2)}+1} - j_{k_{(p-2)}+1} - ... - i_{k_{(p-1)}} - j_{k_{(p-1)}} \ ,$$

$$i_{k_{(p-1)}+1} - j_{k_{(p-1)}+1} - ... - i_{k_p} - j_{k_p} \tag{59}$$

with

$$j_{k_s} \notin N(i_{k_s+1}) \text{ for all } 1 \leq s \leq p-1$$

and, due to the vicinal preorder of the underlying threshold graph, even with

$$j_k \notin N(i_m) \text{ for all } k, m \in \{0, 1, ..., |M|-1\}$$

$$\text{if } k_{s_1} + 1 \leq k \leq k_{s_1+1} \ , \ \ k_{s_2} + 1 \leq m \leq k_{s_2+1} \text{ with } 1 \leq s_1 < s_2 \leq p-1 \ . \tag{60}$$

We know that, for greedy matchings, the existence of an alternating T-cycle for a set $T \subseteq M$ is equivalent to the existence of a sorted alternating T-cycle (Corollary 68), and that sorting the edges of a set T does not have any impact on the connectedness of the patching graph (Remark 102(2)).

In conjunction with (59) and (60), this implies that we have to look only for subsets

$$T_q \subseteq \{(i_{k_{(s-1)}+1}, j_{k_{(s-1)}+1}), (i_{k_{(s-1)}+2}, j_{k_{(s-1)}+2}), ..., (i_{k_s}, j_{k_s})\}$$

when searching for sets T_q that could lead to a family $(T_q)_{q \in Q}$ of alternating T_q-cycles relative to which the patching graph is connected.

Additionally, while searching within each of the sets

$$\{(i_{k_{(s-1)}+1}, j_{k_{(s-1)}+1}), (i_{k_{(s-1)}+2}, j_{k_{(s-1)}+2}), ..., (i_{k_s}, j_{k_s})\} \tag{61}$$

given by s with $1 \leq s \leq p$, we can restrict ourselves to constructing subsets T_q that do not contain more than one edge from the same cycle of the twin-induced structure (because doing so will have no impact on the connectedness of the patching graph).

Moreover, due to the vicinal preorder of the underlying threshold graph, if we have

$$j_k \in N(i_m) \text{ for some } k, m \in \{0, 1, ..., |M|-1\} \text{ with } k < m \ ,$$

we also have

$$k_s + 1 \leq k < m \leq k_{s+1} \text{ for some } s \text{ with } 1 \leq s \leq p-1 \ ,$$

$$j_l \in N(i_m) \text{ for all } k \leq l \leq m-1 \text{ , and}$$

$$j_k \in N(i_l) \text{ for all } k+1 \leq l \leq m \ .$$

Consequently, we can construct the subsets T_q while only considering edges along the order of (59).

We observe that these are the principles that FCA follows in lines [06], [07] and [09].

Case (1): FCA terminates with $q^* = 0$. Due to line [07] of the algorithm, this implies that FCA has not found any pair of edges (i_k, j_k) and (i_{k+1}, j_{k+1}) with $j_k \in N(i_{k+1})$ and $C[(i_k, j_k)] \neq C[(i_{k+1}, j_{k+1})]$. Hence, each of the sets (61) given by $1 \leq s \leq p$ consists only of edges that are elements of the same cycle of the twin-induced structure of the matching. From what has been said above (in particular (60)), we can conclude that it is not possible to construct a set $\widehat{T}$ with two edges from different cycles of the twin-induced structure as its elements such that there exists an alternating $\widehat{T}$-cycle relative to M. This implies, by definition of the patching graph, that there exists no set $\widehat{Q}$ and no family $(\widehat{T}_q)_{q \in \widehat{Q}}$ with alternating $\widehat{T}_q$-cycles relative to which the patching graph would be connected. (This means that relative to all possible families of alternating T_q-cycles the patching graph consists only of isolated nodes.)

Case (2): FCA terminates with $q^* > 0$ and the patching graph $P(D, E)$ relative to the family (T_q) with which FCA terminates is not connected. We distinguish two sub-cases.

Case (2.1): The patching graph has an isolated node $i \in D$. Then, by definition of the patching graph and due to the fact that FCA does not construct sets T_q that contain edges from the same cycle of the twin-induced structure of the matching (lines [06] and [07]), there must be sets according to (61) such that all edges $(i_m, j_m) \in C_i$ are within these sets and none of these sets contains any edge $(i_k, j_k) \notin C_i$. From what has been said above (in particular (60)), we can conclude that for all $(i_m, j_m) \in C_i$ and $(i_k, j_k) \notin C_i$ it is not possible to construct a set $\widehat{T}$ with (i_m, j_m), $(i_k, j_k) \in \widehat{T}$ such that there exists an alternating $\widehat{T}$-cycle relative to M. Hence, by definition of the patching graph, there exists no set $\widehat{Q}$ and no family $(\widehat{T}_q)_{q \in \widehat{Q}}$ with alternating $\widehat{T}_q$-cycles relative to which the patching graph would be connected.

Case (2.2): The patching graph is not connected and has no isolated nodes. Then there exists, by definition of the patching graph $P(D, E)$, a partition $D = D_1 + D_2$ such that among the family (T_q) constructed by FCA there is no set T_q with (i_m, j_m), $(i_k, j_k) \in T_q$ for some edges (i_m, j_m), $(i_k, j_k) \in M$ with $C[(i_m, j_m)] \in D_1$ and $C[(i_k, j_k)] \in D_2$. Due to lines [06] and [07] of the algorithm, we know that all edges from cycles $(C_d)_{d \in D_1}$ are elements of some sets (61), while all edges from cycles $(C_d)_{d \in D_2}$ must be elements of some other sets (61). From what has been said above (in particular (60)), we can conclude that for all (i_m, j_m), $(i_k, j_k) \in M$ with $C[(i_m, j_m)] \in D_1$ and $C[(i_k, j_k)] \in D_2$ it is not possible to construct a set $\widehat{T}$ with (i_m, j_m), $(i_k, j_k) \in \widehat{T}$ such that there exists an alternating $\widehat{T}$-cycle relative to M. Hence, by definition of the patching graph, there exists no set $\widehat{Q}$ and no family $(\widehat{T}_q)_{q \in \widehat{Q}}$ with alternating $\widehat{T}_q$-cycles relative to which the patching graph would be connected. ■

Remark 112 *Note that we needed the degree property of greedy matchings (Proposition* 79*) for the proof of Theorem* 111*, while this was not necessary for Theorem* 110*. This setting exhibits an interesting asymmetry. According to Theorem* 110*, our algorithm FCA constructs a family*

of alternating T_q-cycles relative to which the patching graph might be connected, but only on the basis of the degree property of greedy matchings will we be able to construct a twin-constrained Hamiltonian cycle from it (Theorem 108). Conversely, every twin-constrained Hamiltonian cycle (unless we have a twin-induced structure with only one cycle) implies a connected patching graph (Corollary 104), but only on the basis of the degree property are we guaranteed to find a suitable family of alternating T_q-cycles with FCA (Theorem 111).

8.5 An algorithm for recognising twin-constrained Hamiltonian threshold graphs ($TGHRA$)

In the previous three chapters we have developed, on the basis of the concepts of sorted alternating T-cycles and the patching graph, both a necessary and a sufficient condition for deciding on the twin-constrained Hamiltonicity of threshold graphs. In the following, as a summary of the results of this chapter, we will present a polynomial-time algorithm that recognizes twin-constrained Hamiltonian threshold graphs and constructs a twin-constrained Hamiltonian cycle, if such a cycle exists.

Algorithm 113 *(Twin-constrained Hamiltonicity Recognition Algorithm - TCHRA)*

INPUT: a threshold graph $G(N,E)$ with the (even) set of nodes
$N_G = \{1,2,...,2n\}$, neighbourhoods $N(i)$ for all $i \in N_G$,
and $b : N_G \to N_G$ a twin-node function.

[01] *Run $MGTMA_{\max}$, output: sorted matching M.*
[02] *If $|M| < n$ then INFEASIBLE, STOP.*
[03] *Calculate twin-induced structure $(C_d)_{d\in D}$*
[04] *If $d = 1$ then FEASIBLE, STOP.*
[05] *Calculate list with $C[(i_k,j_k)] = d :\Longleftrightarrow (i_k,j_k) \in C_d$ for all $1 \le k \le n$.*
[06] *Run FCA, output: q^*, lists T_q and S_q for $1 \le q \le q^*$.*
[07] *If $q = 0$ then INFEASIBLE, STOP.*
[08] *(check if patching graph connected)*
(first alternating T_q-cycle to be considered)
$q := 1$, $Q := \{1\}$, $S := S_1$;
[09] *(consider all other alternating T_q-cycles)*
WHILE $(q \le q^* \wedge S \ne D)$ DO
[10] *REPEAT $q := q + 1$*
[11] *UNTIL $((q = q^* + 1) \vee (q \notin Q \wedge S_q \cap S \ne \varnothing))$*
[12] *If $q \le q^*$ then $S := S + S_q$, $Q := Q + \{q\}$, $q := 1$;*
[13] *END DO;*
[14] *If $S \ne D$ then INFEASIBLE else FEASIBLE; STOP.*

OUTPUT: statement of FEASIBILITY or INFEASIBILITY

Theorem 114 *(Complete recognition of twin-constrained Hamiltonicity by TCHRA)*
Let $G(N,E)$ be a threshold graph $G(N,E)$ with the (even) set of nodes $N_G = \{1,2,...,2n\}$, neighbourhoods $N(i)$ for all $i \in N_G$, and $b : N_G \to N_G$ a twin-node function. Then $TCHRA$ recognises whether or not G is twin-constrained Hamiltonian with respect to b.

Proof. The algorithm calculates a maximum cardinality modified matching M according to Proposition 75 (line [01]). Trivially, if M is not perfect, there cannot be any twin-constrained Hamiltonian cycle on G (line [02]). Having calculated the twin-induced structure of M with respect to b (line [03]), there obviously exists a twin-constrained Hamiltonian cycle on b if the twin-induced structure consists of only one cycle (line [04]). If this is not the case, $TCHRA$ generates a list that indicates for each edge from M the cycle of which the edge is an element (line [05]) and tries to construct a family $(T_q \subseteq M)_{q \in Q}$ of alternating T_q- cycles such that the patching graph of the twin-induced structure relative to (T_q) is connected (line [06]). Because of Theorem 111 in conjunction with Theorem 103 there exists no twin-constrained Hamiltonian cycle on G if FCA terminates with $q = 0$ (line [07]).

If FCA has not terminated with $q = 0$, the algorithm has found a family of alternating T_q-cycles (Theorem 110) and checks if the patching graph of the twin-induced structure relative to (T_q) is connected. This is achieved in lines [08] to [14]. The algorithm checks the connectedness by proceeding in the fashion of Theorem 108: it starts with one set T_q, namely (arbitrarily chosen) the set T_1, and considers the set S_q of the indices of all cycles of the twin-induced structure of which there are elements in T_q (line [08]). If the set S_q (here: S_1) contains the indices of all cycles of the twin-induced structure ($S_1 = D$), the patching graph is connected by definition and the algorithm jumps to line [14] to state the feasibility of the twin-constrained Hamiltonian cycle problem on the basis of Theorem 108. If this is not the case, $TCHRA$ looks for another set S_q that shares the index of one cycle with S_1 (lines [10] and [11], which corresponds to the criterion $\overline{C}_{\overline{Q}} \cap T_{q^*} \neq \varnothing$ in Theorem 108). If such a set does not exist, the patching graph is not connected and the algorithm terminates in line [14] with a statement of infeasibility on the basis of Theorem 111 in conjunction with Theorem 103.

Otherwise, if such a set does exist, $TCHRA$ adds this new set S_q to the set S_1 (line [11]) and checks again if the set S of the indices of all cycles of the twin-induced structure considered so far is equal to the node set D of the patching graph (line [09]). If yes, the patching graph is connected and the algorithm jumps to line [14] and states feasibility, again on the basis of Theorem 108. Otherwise, $TCHRA$ continues by searching for another set S_q in lines [10] and [11]. This procedure is repeated until $TCHRA$ has found a sufficient number of sets S_q and can declare feasibility on the basis of Theorem 108, or $TCHRA$ has considered all sets S_q (line [10], condition $q = q^* + 1$) and has still not been able to link all nodes of the patching graph ($S \neq D$, line [14]), in which case we conclude infeasibility on the basis of Theorems 103 and 111.

As $TCHRA$ always ends with a statement about (in)feasibility and this statement, as we have shown, is correct, $TCHRA$ can decide whether or not a given threshold graph is twin-constrained Hamiltonian with respect to a certain twin-node function. ■

Remark 115 (1) *In the form given above, $TCHRA$ solely decides whether the twin-constrained Hamiltonian cycle problem is feasible. If we would like to construct a twin-constrained Hamiltonian cycle if such a cycle exists, we can add the procedure of constructing a cycle as used in Lemmas* 106 *and* 107 *and Theorem* 108*. (This procedure can even be simplified as FCA constructs only sets T_q that have at most one edge from each of the cycles of the twin-induced structure − in our proofs of Lemmas* 106 *and* 107 *we could not take this for granted.*

(2) *If we would like to use $TCHRA$ for solving an $MSSP$, we have to add, before running $TCHRA$, a pair of dominating twin-nodes to our threshold graph. A statement of (in)feasibility by $TCHRA$ is then equivalent to a statement of (in)feasibility regarding the MSSP. If we modify $TCHRA$ to construct a twin-constrained Hamiltonian cycle, provided such a cycle exists, we remove our pair of dominating twin-nodes and obtain a solution to the MSSP (cf. section* 8.1*).*

The performance of $TCHRA$ with respect to computational time will be illustrated in chapter 9. We conclude this chapter by addressing the complexity of the twin-constrained Hamiltonian cycle problem on threshold graphs.

Proposition 116 *(Complexity of recognizing twin-constrained Hamiltonian cycles on threshold graphs)*

Let $G(N,E)$ be a threshold graph $G(N,E)$ with the (even) set of nodes $N_G = \{1,2,...,2n\}$, neighbourhoods $N(i)$ for all $i \in N_G$, and $b : N_G \to N_G$ a twin-node function. Then it is possible to decide in at most $O(n^2)$ time whether or not G is twin-constrained Hamiltonian with respect to b. If yes, we can construct a solution in further $O(n)$ time.

Proof. We will address only those parts of $TCHRA$ that might need more than $O(1)$ time. The matching algorithm (line [01]), also in its modified form, requires $O(n \log_2 n)$ time because of the sorting algorithm that is a part of it (Corollary 50). The twin-induced structure of the matching (line [03]) can be calculated in at most $O(\frac{n^2}{2})$ time as we have n edges from the matching and at most $\frac{n}{2}$ cycles of the twin-induced structure. The list of the indices of the cycles of the twin-induced structure of which an edge is an element (line [05]) can be generated while calculating the twin-induced structure (which adds only $O(1)$ time to each step of calculating the twin-induced structure). FCA examines all edges in the matching exactly once (line [06]) and therefore terminates in at most $O(n)$ time. The time for arriving at a decision of whether the patching graph is connected (lines [08] to [13]) is bound by $O(\frac{n}{4}(\frac{n}{2}+1))$ time as the number of alternating T_q-cycles is bound by $\frac{n}{2}$. Hence the complexity of all subroutines of $TCHRA$

leads to $O(n^2)$ time as an upper bound on the performance of $TCHRA$. Constructing a twin-constrained Hamiltonian cycle can be done on the basis of the subroutine that checks whether the patching graph is connected as long as we keep track on the order in which we add the sets S_q to the set S. With this information we have to deal with every edge of the matching only once, which leads to a complexity of $O(n)$. ■

At the end of our discussion of twin-constrained paths and cycles in Chapters 7 and 8 we can finally make a statement on the complexity of the MSSP, which we set out to solve in Chapter 1. As mentioned in Chapter 5.3, the complexity of the MSSP is between the complexity of deciding on the Hamiltonicity of a threshold graph (which is in problem class **P**) and the complexity of the problem of deciding whether there exists, on a given graph, a twin-constrained Hamiltonian path (which is an **NP**-complete problem). We are now prepared to give a definite answer on the question of whether or not the MSSP can be solved in polynomial time.

Corollary 117 *(Complexity of the MSSP)*

Let $G(N, E)$ be a threshold graph $G(N, E)$ with the (even) set of nodes $N_G = \{1, 2, ..., 2n\}$, neighbourhoods $N(i)$ for all $i \in N_G$, and $b : N_G \to N_G$ a twin-node function. Then it is possible to decide in $O(n^2)$ time whether or not the MSSP on G with respect to b is feasible. If yes, we can construct a solution in further $O(n)$ time.

Proof. In view of the fact that adding a pair of dominating twin-nodes to G can be done in $O(1)$ time within $MGTMA$, right after having sorted the nodes, the statement follows directly from the previous proposition in conjunction with Remark 115(2). ■

9 Computational results

In this section, we will provide computational results for the two algorithms we developed: the heuristic $MSSPH$, which looks for specific feasible and infeasible cases of the MSSP (chapter 7.8), and the polynomial-time algorithm for recognizing all twin-constrained Hamiltonian threshold graphs ($TCHRA$), which was given in chapter 8.5.

9.1 General remarks about the implementation

Both algorithms were implemented in C++ code using the integrated development environment Microsoft Visual C++ and executed on a computer equipped with an Intel Core 2 Duo processor (U7500) with a frequency of 1.06GHz.

While the implementation of $TCHRA$ follows exactly the description in chapter 8.5, our implementation of $MSSPH$ does not include the final Step [10] of the algorithm presented in chapter 7.8 (i.e. the step of trying to find a feasible solution of the $MSSP$ by looking for an extreme point of the polyhedron P_I^*). As $MSSPH$ is intended to be a fast heuristic to quickly decide on the feasibility and infeasibility of a large percentage of given instances and our computational experiments (as we will soon see) have led to convincing results beyond expectation even without including Step [10], this implementation decision seems to be justified. Additionally, in view of the computational results, it is questionable whether the time-consuming process of calling an LP solver in Step [10] would lead to computational results that could compete with $TCHRA$, which can decide on the (in)feasibility of 100% of the instances given. The source codes of our implementations of $MSSPH$ and $TCHRA$ are given in Appendices A and B, respectively.

For the sake of comprehensiveness, Appendix C provides the source code of a much longer and more complex heuristic called "$MSSP$ 3.4", which, being an extended version of $MSSPH$, also looks for some types of non-structure-preserving (path- and cycle-splitting) solutions and can hence decide of the (in)feasibility of larger percentage of instances than $MSSPH$. This heuristic was used for computational experiments during the process of research on the $MSSP$ and turned out to be helpful for gaining some of the insights into the structure of the $MSSP$ that led to chapters 7 and 8 of the present thesis. Moreover, the results of this longer algorithm gave strong hints that an implementation of Step [10] of $MSSPH$ would not lead to computational advantages that go beyond what has been achieved with the two algorithms developed in chapters 7 and 8. As the computational experiments with $MSSPH$ have led, as we will see, to results that show that $MSSPH$ in its implemented version clearly fulfills the purpose for which it was developed, we will not discuss more in detail the heuristic $MSSP$ 3.4 as presented in Appendix C.

The computational tests of $MSSPH$ and $TCHRA$ were carried out on the basis of more than 100 different types of randomly generated data sets, each with 10^6 to 10^9 instances of the

$MSSP$. Due to the large number of instances tested (and the choice of the data sets used, see below), we can assume that the results provide us with a representative picture of the behaviour of $MSSPH$ and $TCHRA$ in general. The reminder of this chapter addresses the most relevant aspects of these tests by presenting and discussing the computational results for 31 randomly generated data sets with 10^6 instances each.

One instance of the $MSSP$ is characterized by the minimal distance α of the scoring knives, the number n of boxes, and the widths of the $2n$ scores. Without loss of generality all instances generated used the value $\alpha = 70mm$, which is the value mentioned in the original problem description by Goulimis (2004). The same article mentioned that the number of boxes is typically up to $n = 10$. To give a more reliable upper bound on the computational time, the majority of the computational results presented here is based on data sets consisting of 20 boxes, but we will also have a look at the behaviour of $MSSPH$ and $TCHRA$ for other data sets, consisting of 10 to 100 boxes.

The widths of the scores, i.e. the numbers assigned to the nodes of the underlying threshold graph, were assumed to be independent and identically distributed discrete random variables that assume integer values. Computational tests were carried out for discrete uniform distributions and discrete versions of symmetric triangular distributions. The choice of the latter type of distribution is motivated by the idea that a discrete distribution based on triangular distribution provides us with a simple way of studying the behaviour of our algorithms in cases in which the width of the boxes is distributed around a peak value (similar to the situation of a normal distribution). The decision for these two distributions implies that, with respect to the width of the scores, all data sets are fully defined by the type of distribution (uniform / triangular) and a single interval $[a, b]$, which indicates the range in which the (integer) widths of the scores can be found with non-zero probability. For the case of a uniformly distributed random variable X over $S := \{a, a+1, ..., b\}$, this implies a probability mass function

$$f_X : S \to [0, 1]$$

$$\text{with } f_X(x) = \frac{1}{b-a+1} \text{ for } x \in S \ .$$

The probability mass function of the discrete version of a symmetric triangular distribution on the same interval (assuming that $b-a+1$ is even, which does not seem to be too restrictive an assumption here) is given by

$$f_X(x) = \int_{x-0.5}^{x+0.5} \frac{4(x-a')}{(b'-a')^2} dx \text{ for } x \in S, x < \frac{b+a}{2}$$

and

$$f_X(x) = \int_{x-0.5}^{x+0.5} \frac{4(b'-x)}{(b'-a')^2} dx \text{ for } x \in S, x > \frac{b+a}{2}$$

with the parameters a' and b' of the symmetric triangular distribution being defined by

$$a' := a - 0.5 \text{ and } b' := b + 0.5 \ .$$

This yields

$$f_X(x) = \frac{2 + 4(x-a)}{(b-a+1)^2} \text{ for } x \in S, x < \frac{b+a}{2}$$

and

$$f_X(x) = \frac{2 + 4(b-x)}{(b-a+1)^2} \text{ for } x \in S, x > \frac{b+a}{2} \ . \tag{62}$$

The generation of the uniformly distributed random numbers in C++ was straight forward as C++ is equipped with a function that returns uniformly distributed discrete pseudo-random numbers in the interval $[0, 32767]$. For generating triangularly distributed random numbers, our implementation derives the cumulative distribution function of the required triangular distribution from (62) in a first step, transforms the random numbers provided by C++ into uniformly distributed random numbers in the interval $[0, 1]$ in a second step, and finally calculates the triangularly distributed random numbers by means of the inverse of the cumulative distribution function (inverse transform sampling, see Devroye (1986), for example). While being not optimal from the perspective of computational time, this method has been chosen because of its comparably easy implementation and in view of the fact that the intervals that define our distributions are rather small such that the loss of computational time due to this method can be disregarded for our purpose.

9.2 Evaluation of $MSSPH$

We will proceed by discussing the computational results for our heuristic $MSSPH$, which can be found in Tables 1 to 5. The columns of Table 1 contain the results of 6 instances on the interval $[1, 70]$ with a uniform distribution that differ only with respect to the number of boxes (i.e. we have a look at the behaviour of the heuristic for the case that the scores of the boxes are distributed between 1 and the minimum distance of the knives). Apart from the computational time needed to analyse 10^6 instances and the number of feasible and infeasible ones found among these, the table contains also the number of instances that could be identified as one of the 9 cases of (in)feasible instances that $MSSPH$ looks for. For the reader's convenience, we will give a brief list of these cases here (see chapter 7.8 for details):

- Case (1): too many nodes that are elements of a maximal stable set $\Rightarrow INFEASIBLE$

- Case (2): a pair of twin-nodes consists of two isolated nodes $\Rightarrow INFEASIBILE$,
- Case (3): the threshold graph has more than two isolated nodes $\Rightarrow INFEASIBLE$,
- Case (4): there is no matching with at least cardinality $|M| < n-1 \Rightarrow INFEASIBLE$,
- Case (5): the threshold graph allows for a perfect matching $\Rightarrow FEASIBLE$,
- Case (6): $MTGMA_{\min}$ has generated a matching the twin-induced structure of which is a direct solution to the $MSSP \Rightarrow FEASIBLE$,
- Case (7): there exists a structure-preserving solution w.r.t. the matching generated by $MTGMA_{\min} \Rightarrow FEASIBLE$,
- Case (8): $MTGMA_{\max}$ has generated a matching the twin-induced structure of which is a direct solution to the $MSSP \Rightarrow FEASIBELE$,
- Case (9): there exists a structure-preserving solution w.r.t. the matching generated by $MTGMA_{\max} \Rightarrow FEASIBLE$.

We observe that in all types of instances given in Table 1 our heuristic is able to decide on the (in)feasibility of more than 98.7% of the random instances in quite an efficient way (less than 10 minutes for 10^6 instances even in the case of 100 boxes, i.e. in the case of an underlying threshold graph with 200 nodes), which should be sufficient for all practical purposes. The higher the number of nodes, the higher the percentage of instances that the algorithm can solve, which mainly seems to be due to the fact that the probability that a matching with a cardinality of at least $n-1$ exists apparently decreases (case (4)). This compensates for the decline in the number of instances that turn out to be one of the cases (6) to (9), which can be expected to become less likely when the number of nodes increases. All in all, the percentage of instances that $MSSPH$ can prove to be (in)feasible in such a short amount of time is remarkable If we take into account that, for each instance with 10 boxes, there are about 1.858×10^9 possible permutations of the boxes (see chapter 1), this result can only be explained by the fact that our heuristic must be looking at exactly those characteristics of an instance that are truly crucial for the $MSSP$. Therefore, if comparable results can be achieved also under different distributions, this would ultimately justify the approach to the $MSSP$ that we chose in chapter 2: we could assume to have found the very "mathematical essence" of what the MSSP is structurally about.

Boxes (n)	10	20	40	60	80	100
Interval	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]
Time (/sec.)	12	33	111	218	376	575
# feas. inst.	536603	438780	381870	365379	359936	361210
# infeas. inst.	450408	547053	606593	624833	632033	631739
% solved inst.	98.7011	98.5833	98.8463	99.0212	99.1969	99.2949
(1) $\|S_{\max}\| > n+1$	212827	256190	277402	279735	276735	270052
(2) $\{i, b(i)\} \subseteq D_0$	13715	4823	1151	335	123	37
(3) $\|D_0\| > 2$	30936	23968	11698	5408	2351	975
(4) $\|M\| < n-1$	192930	262072	316342	339355	352824	360675
(5) $\|M\| = n$	244255	226902	230637	242158	253495	266008
(6) $TGMA_{\min}$: P	61328	34334	17911	12052	8826	7134
(7) $TGMA_{\min}$: S	213213	161706	121436	101149	88952	80278
(8) $TGMA_{\max}$: P	14013	9844	5390	3695	2683	2198
(9) $TGMA_{\max}$: S	3794	5994	6496	6325	5980	5592

Table 1: MSSPH, uniform distribution I

In order to see if our results can be confirmed for different types of data sets, computational tests have been carried out for uniformly distributed score widths on different intervals. Table 2 shows results for intervals with a diameter of about half the size of the minimal distance of the scoring knives in the case of 20 boxes. We can observe that the percentage of solved instances is higher the less the interval covers the area around half of the minimal distance of the knives, i.e. the area around $35mm$. This is due to the fact that, if the interval contains only small numbers, too many nodes have a rather low degree and the heuristic recognises this as cases (1) to (3), while our heuristic can easily prove feasibility on the basis of a perfect matching if the interval contains only larger numbers. We notice that for all these distributions, the algorithm can decide on the (in)feasibility of a higher percentage of instances than for the distribution in Table 1. Also, we can conclude that the results for intervals with numbers smaller or larger than in the intervals given in Table 2 would be the same as in the cases of $[6, 35]$ and $[36, 65]$, respectively, i.e. it is not necessary to test $MSSPH$ for score widths above $70mm$ and negative numbers. This implies also that we do not have to carry out computational tests for intervals with a diameter larger than 70.

Boxes (n)	20	20	20	20	20	20	20
Interval	[6,35]	[11,40]	[16,45]	[21,50]	[26,55]	[31,60]	[36,65]
Time (/sec.)	4	4	5	36	39	36	34
# feas. inst.	0	0	12	551825	997180	10^6	10^6
# infeas. inst.	10^6	10^6	999980	440718	2816	0	0
% solved inst.	100	100	99.9992	99.2570	99.9996	100	100
(1) $\lvert S_{\max}\rvert > n+1$	10^6	999918	894507	185047	872	0	0
(2) $i, b(i) \in D_0$	0	81	74531	1650	0	0	0
(3) $\lvert D_0\rvert > 2$	0	1	30904	8406	0	0	0
(4) $\lvert M\rvert < n-1$	0	0	38	245615	1944	0	0
(5) $\lvert M\rvert = n$	0	0	0	353452	992475	10^6	10^6
(6) $TGMA_{\min}$: P	0	0	6	30315	458	0	0
(7) $TGMA_{\min}$: S	0	0	2	156668	4236	0	0
(8) $TGMA_{\max}$: P	0	0	4	6043	3	0	0
(9) $TGMA_{\max}$: S	0	0	0	5374	8	0	0

Table 2: MSSPH, uniform distribution II

Apparently, the worst case for $MSSPH$ is a situation when the widths of the boxes are centered around half of the minimum knife distance because in such a situation only comparably few instances are characterized by the existence of a perfect matching or by many nodes with a low degree. To further evaluate our heuristic in these critical worst case situations, $MSSPH$ was confronted with distributions with a support around half of the minimum knife distance that differ only with the respect to the diameter of the interval (Table 3). It turns out that, with respect to the percentage of instances solved, the distribution over the interval $[1, 70]$ is the worst case, which suggests that the results given in Table 1 do provide an appropriate overview of the performance of $MSSPH$. In contrast to this, the most unfavourable case with respect to computational time occurs when the interval has a small diameter because less instances can be recognized as infeasible on the basis of the fast criteria (1) to (3). We observe, however, that, all in all, the amount of time that our heuristic requires remains in the same order of magnitude, i.e. with respect to computational time, $MSSPH$ is rather stable.

Boxes (n)	20	20	20	20	20	20	20
Interval	[1,70]	[6,65]	[11,60]	[16,55]	[21,50]	[26,45]	[31,40]
Time (/sec.)	33	33	34	35	36	37	41
# feas. inst.	438780	453963	474338	504835	551852	634258	814170
# infeas. inst.	547053	532925	513897	485143	440718	361080	184440
% solved inst.	98.5833	98.6888	98.8235	98.9978	99.2570	99.5338	99.8610
(1) $\lvert S_{\max}\rvert > n+1$	256190	246833	233413	214724	185047	133915	39091
(2) $i, b(i) \in D_0$	4823	4188	3440	2577	1650	672	19
(3) $\lvert D_0\rvert > 2$	23968	20788	17380	13451	8406	3142	61
(4) $\lvert M\rvert < n-1$	262072	261116	259664	254391	245615	223351	145269
(5) $\lvert M\rvert = n$	226902	243273	266281	299807	353452	451700	686745
(6) $TGMA_{\min}$: P	34334	33832	32939	31724	30315	26542	15799
(7) $TGMA_{\min}$: S	161706	161870	161023	160429	156668	147305	107664
(8) $TGMA_{\max}$: P	9844	9108	8259	7258	6043	4151	1497
(9) $TGMA_{\max}$: S	5994	5880	5836	5617	5374	4560	2465

Table 3: MSSPH, uniform distribution III

In order to further test the stability of the performance of our heuristic, we consider cases of symmetric triangular distributions centered around half of the minimal knife distance (Tables 4 and 5). In view of the results above, these cases must be considered rather critical. The distributions used for the computations the results of which are provided in Table 4 combine the worst cases we have found (i.e. the cases of the first and the last column of the previous table) into one distribution: a broad interval ($[1, 70]$) with a probability mass function that has a peak at half of the minimum knife distance. Indeed, as Table 4 shows, the computational time increases by approximately factor 1.5, while the percentage of instances solved decreases to 97.0681% in the case of 100 boxes, i.e. 200 nodes. But again, our results remain in the same order of magnitude.

Boxes (n)	10	20	40	60	80	100
Interval	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]
Time (/sec.)	17	45	135	274	472	740
# feas. inst.	556274	454867	387429	364187	352965	348845
# infeas. inst.	427224	522679	586227	607991	618383	621836
% solved inst.	98.3498	97.7546	97.3656	97.2178	97.1348	97.0681
(1) $\|S_{\max}\| > n+1$	177983	202842	200249	186220	170705	155967
(2) $\{i, b(i)\} \subseteq D_0$	16960	8117	3587	2243	1542	1125
(3) $\|D_0\| > 2$	38675	41620	39416	35895	33015	30276
(4) $\|M\| < n-1$	193606	270100	342975	383633	413121	434468
(5) $\|M\| = n$	235957	203659	188379	188433	191230	196533
(6) $TGMA_{\min}$: P	66344	40028	23533	16978	13348	10954
(7) $TGMA_{\min}$: S	233135	191083	157947	142910	133533	126983
(8) $TGMA_{\max}$: P	17054	13832	10131	8122	7039	6398
(9) $TGMA_{\max}$: S	3784	6265	7439	7744	7815	7977

Table 4: MSSPH, triangular distribution I

For the sake of comprehensiveness, Table 5 shows the results for triangular distributions on sets that contain only numbers within a range of about half the minimum knife distance. We can conclude from the first two columns (in comparison with Table 2) that 4 sec. of the computational time needed in the cases investigated in Tables 4 and 5 is due to the extra time needed for calculating the random instances under the triangular distribution by means of inverse transform sampling, which implies that in all non-trivial cases, the computational times of instances with a triangular distribution are indeed comparable with those of a uniform distribution. We observe that, apart from the time needed for generating the instances, the results under a triangular distribution are similar to those for a uniform distribution.

In sum we can state that our heuristic is remarkably efficient and stable with respect to both the percentage of instances that it can solve and the computational time required. While in principle there can be "pathological" distributions that might lead to a much longer computational time and/or a significantly lower percentage of instances solved, our analysis on the basis of different types of distributions did not provide any hint that these distributions are likely to occur. (Note that we can reasonably assume, for example, that a distribution with two peaks would not have a significant impact on the performance of our heuristic. If these two peaks were close to each other, we could expect a result similar to those of the triangular distributions tested, and if the peaks were rather distant from each other, we could expect a situation that is even more favourable than that of a uniform distribution.) Therefore, we will end our evaluation of $MSSPH$ here.

Boxes (n)	20	20	20	20	20	20	20
Interval	[6,35]	[11,40]	[16,45]	[21,50]	[26,55]	[31,60]	[36,65]
Time (/sec.)	8	8	9	44	42	39	38
# feas. inst.	0	0	1	603705	10^6	10^6	10^6
# infeas. inst.	10^6	10^6	999997	377089	0	0	0
% solved inst.	100	100	99.9998	98.0794	100	100	100
(1) $\lvert S_{\max}\rvert > n+1$	10^6	10^6	995781	97303	0	0	0
(2) $i, b(i) \in D_0$	0	0	2000	4920	0	0	0
(3) $\lvert D_0\rvert > 2$	0	0	2102	24959	0	0	0
(4) $\lvert M\rvert < n-1$	0	0	114	249907	0	0	0
(5) $\lvert M\rvert = n$	0	0	0	340037	999996	10^6	10^6
(6) $TGMA_{\min}$: P	0	0	0	37694	0	0	0
(7) $TGMA_{\min}$: S	0	0	1	207067	4	0	0
(8) $TGMA_{\max}$: P	0	0	0	12377	0	0	0
(9) $TGMA_{\max}$: S	0	0	0	6530	0	0	0

Table 5: MSSPH, triangular distribution II

As in practical cases, the number of instances is (only) "in the many hundreds (if not thousands)" (Goulimis, 2004, pp. 1368) we can conclude on the basis of both the computational time needed for 10^6 instances under various distributions and the percentage of instances solved that $MSSPH$ is an efficient, stable and reliable method for quickly solving a large percentage of instances in a typical practical situation.

9.3 Evaluation of $TGHRA$

We now turn to the computational test carried out with $TGHRA$. Tables 6 to 10 contain the results for the same type of randomly generated data sets that were used for evaluating the behaviour of $MSSPH$ as presented in Tables 1 to 5. Again, the computational tests were executed on the basis of data sets with 10^6 instances each and a scoring knife distance $\alpha = 70mm$. As $TGHRA$ looks for a twin-constrained Hamiltonian cycle and we are interested in finding a twin-constrained Hamiltonian path, our implementation adds a pair of twin-nodes nodes (i.e. one box) with score widths $v(i_0) = v(i_1) = 70$ before starting $TGHRA$ (see chapter 8.1 and Remark 115(2)). The following tables list for each data set the time the algorithm took to consider all instances and the number instances that turned out to be feasible and infeasible. Moreover, the tables include the number of instances that were found to be among the 5 possible cases that lead to a decision about the question of (in)feasibility. For the reader's convenience we will give a brief list of these cases here (see chapter 8.5 for details):

- Case (1): the underlying threshold graph has no perfect matching $\Rightarrow INFEASIBLE$,
- Case (2): $MTGMA_{\max}$ has generated a matching the twin-induced structure of which is a direct solution to the $MSSP \Rightarrow FEASIBELE$,

- Case (3): all nodes of the patching graph are isolated $\Rightarrow INFEASIBLE$,
- Case (4): the patching graph is unconnected $\Rightarrow INFEASIBLE$,
- Case (5): the patching graph is connected $\Rightarrow FEASIBLE$.

Analogous to tables 1 to 5, Tables 6 to 10 show the results for a uniform distribution on the interval $[1, 70]$ and different numbers of boxes; for a uniform distribution on intervals that have a diameter of about half of the minimum knife distance; for a uniform distribution on intervals centered around half of the minimum knife distance; for a triangular distribution on the interval $[1, 70]$ and different numbers of boxes; and for a triangular distribution on intervals that have a diameter of about half of the minimum knife distance, respectively.

Boxes (n)	10	20	40	60	80	100
Interval	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]
Time (/sec.)	20	57	180	376	665	971
# feas. inst.	533664	446774	391555	374177	369457	368791
# infeas. inst.	466336	553226	608445	625823	630543	631209
(1) $\|M\| < n$	464202	552802	608395	625810	630539	631207
(2) full P	232854	141637	89311	69354	59990	53319
(3) $q^* = 0$	1003	171	12	3	0	0
(4) PG uncon.	1127	253	38	10	4	2
(5) PG con.	300810	305137	302244	304823	309467	315472

Table 6: TGHRA, uniform distribution I

Boxes (n)	20	20	20	20	20	20	20
Interval	[6,35]	[11,40]	[16,45]	[21,50]	[26,55]	[31,60]	[36,65]
Time (/sec.)	58	57	55	57	59	57	58
# feas. inst.	0	0	11	556625	997684	10^6	10^6
# infeas. inst.	10^6	10^6	999989	443375	2316	0	0
(1) $\|M\| < n$	10^6	10^6	999989	443234	2316	0	0
(2) full P	0	0	4	175694	314333	316055	315776
(3) $q^* = 0$	0	0	0	50	0	0	0
(4) PG uncon.	0	0	0	91	0	0	0
(5) PG con.	0	0	7	380931	683351	683945	684224

Table 7: TGHRA, uniform distribution II

Boxes (n)	20	20	20	20	20	20	20
Interval	[1,70]	[6,65]	[11,60]	[16,55]	[21,50]	[26,45]	[31,40]
Time (/sec.)	57	57	57	57	57	58	59
# feas. inst.	446774	462638	481983	510161	556625	638300	817855
# infeas. inst.	553226	537362	518017	489839	443375	361700	182145
(1) $\|M\| < n$	552802	537034	517699	489623	443234	361653	182121
(2) full P	141637	146619	152328	161392	175694	200205	244121
(3) $q^* = 0$	171	90	101	72	50	17	6
(4) PG uncon.	253	238	217	144	91	30	18
(5) PG con.	305137	316019	329655	348769	380931	438095	573734

Table 8: TGHRA, uniform distribution III

Boxes (n)	10	20	40	60	80	100
Interval	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]	[1,70]
Time (/sec.)	26	68	203	411	699	1070
# feas. inst.	557217	471557	412036	390448	381160	379362
# infeas. inst.	442783	528443	587964	609552	618840	620638
(1) $\|M\| < n$	440082	527614	587668	609397	618724	620579
(2) full P	243679	149626	92866	73103	61494	54759
(3) $q^* = 0$	1344	276	63	25	25	9
(4) PG uncon.	1357	553	233	130	91	50
(5) PG con.	313538	321931	318170	317345	319666	324603

Table 9: TGHRA, triangular distribution I

Boxes (n)	20	20	20	20	20	20	20
Interval	[6,35]	[11,40]	[16,45]	[21,50]	[26,55]	[31,60]	[36,65]
Time (/sec.)	63	63	62	63	64	62	62
# feas. inst.	0	0	0	620480	10^6	10^6	10^6
# infeas. inst.	10^6	10^6	10^6	379520	0	0	0
(1) $\|M\| < n$	10^6	10^6	10^6	378913	0	0	0
(2) full P	0	0	0	196235	315411	315477	315890
(3) $q^* = 0$	0	0	0	188	0	0	0
(4) PG uncon.	0	0	0	419	0	0	0
(5) PG con.	0	0	0	424245	684589	684523	684110

Table 10: TGHRA, triangular distribution II

We observe that the computational time required for solving all instances of the problem is, except in those cases that are trivial for $MSSPH$, about twice the time that our heuristic $MSSPH$ needed, i.e. our algorithm $TGHRA$ takes an amount of time that can be considered reasonable for practical purposes. Again, it is the case that instances with a triangular distribution require a longer computational time. However, if we take into account that generating the triangularly distributed data sets takes, as concluded above, about 4 sec. for all 10^6 instances, the difference in computational time is only marginal. We note that in general the computational time is, for the case of 20 boxes, remarkably independent of the distributions chosen. In fact, our computational experiments show no indication that our algorithm could behave entirely differently if other distributions were chosen. This suggests that we have developed an algorithm that should be suitable and reliable for all cases that might arise in a practical setting.

If we compare the results gained for $TGHRA$ with those for $MSSPH$, we can observe that most infeasible cases can be recognised by $TGHRA$ on the basis of the fact that there is no perfect matching on the underlying threshold graph (case (1) for $TGHRA$). This case corresponds to cases (1) to (4) for $MSSPH$, which implies that the simple criteria that $MSSPH$ applies to prove infeasibility are obviously sufficient to discover almost all infeasible instances of the MSSP. This explains in part the high percentage of instances that $MSSPH$ can solve.

On a side-note we remark that the percentage of infeasible cases that cannot be recognised by the fact that there is no perfect matching on the underlying threshold graph (cases (3) and (4)) decreases as the number of nodes increases (see Tables 6 and 9). This is probably due to the fact that a higher number of nodes on the same interval leads to degree partitions of a higher cardinality, which increases the probability that a degree partition contains nodes that belong to different cycles of the twin-induced structure of the matching. As a consequence, the probability increases that there exist alternating T-cycles that allow for a connected patching graph and hence for a solution of the $MSSP$. Proving this, however, would be a question for further research.

Finally we can conclude that a combination of $MSSPH$ and $TGHRA$ would, in view of our computational experiments, lead to faster algorithm for solving all instances. Such an algorithm could start with $MSSPH$ and look for instances that fall into the categories of cases (1) to (7) of $MSSPH$ and then continue with $TGHRA$ to solve the small number of instances that remain unsolved. Due to the small percentage of instances that we would need $TGHRA$ for (2% to 3% of all instances according to our computational results), we can expect the computational time required by such a combined algorithm to be close to the time that $MSSPH$ takes. In a very time-sensitive practical setting, this combined algorithm should be the algorithm of choice.

10 Conclusion

The Minimum Score Separation Problem (MSSP) is a combinatorial problem that was introduced as an open problem in the OR literature in JORS 55. The present thesis set out to solve it.

In chapter 1, we introduced the MSSP and set the task to develop an algorithm that, at least as a heuristic for a large percentage of instances, can quickly determine whether or not a certain instance of the MSSP is feasible. We proceeded by presenting two ways of modelling this problem in chapter 2, namely the originally proposed way of modelling the MSSP and a new approach. It was argued that modelling the MSSP as a twin-constrained Hamiltonian path problem (instead of a Travelling Politician Problem) is more elegant (also in view of the principle of Occam's razor) as it does not require us to double certain mathematical entities and that it gives us the opportunity to capitalise on the adjacency conditions for each score of a box separately, which might allow for a more direct way of exploiting the structure of the problem.

The succeeding two chapters laid the foundations for this approach. In chapter 3, we related our problem to the literature on Hamiltonian paths, alternating Hamiltonian paths, the TSP, the CTSP and the GTSP and discussed the complexity of these problems and of our twin-constrained Hamiltonian path problem, and in chapter 4 we introduced the concept of the threshold graph and had a look at the basic characteristics of this type of graph.

Building on these preliminaries, most of which had already been addressed in the relevant literature, the following two chapters 5 and 6 studied the existence and structure of paths and cycles on threshold graphs more in detail. As every twin-constrained Hamiltonian path contains a matching on a threshold graph, we decided to examine first the existence of paths and cycles and their properties from the perspective of maximum cardinality matchings on threshold graphs and analyzed the circumstances under which a subset T of a matching can be extended to an (even, matching-dominated, or augmented) T-path. In chapter 5, this led to a maximum cardinality proof of a type of matching algorithms on threshold graphs, to a new criterion for the existence of Hamiltonian paths on threshold graphs (and, as a corollary, to a new proof of a criterion in the literature), and to the insight that the twin-constrained Hamiltonian path problem on threshold graphs is located at the border of NP-complete and polynomial-time problems. Because of this it seemed to be advisable to pursue a two-track policy and be open to developing both a polynomial time algorithm for the MSSP and a fast heuristic. In chapter 6, we introduced the concept of alternating T-cycles relative to a given matching and, building on our criterion for the existence of Hamiltonian paths from chapter 5, we derived several criteria for the existence of alternating T-cycles on threshold graphs in general and for the existence of alternating T-cycles relative to a matching gained from a specific algorithm ($TGMA_{\max}$) in particular.

Having laid the theoretical foundations about constructing paths and cycles on threshold graphs, we turned to the MSSP, following our two-track policy in chapters 7 and 8. Chapter 7 focussed on developing a heuristic for quickly solving a large percentage of instances of the MSSP, while chapter 8, generalizing the insights from chapter 7, led to a polynomial-time algorithm that solves all instances.

In chapter 7, we introduced the concept of a modified matching and of the twin-induced structure of a matching. Analysing the structural setting given by these concepts and making use of the results of chapters 5 and 6, we addressed the question of the feasibility of the MSSP with respect to matchings with cardinalities of $|M| < n - 1$, $|M| = n$, and $|M| = n - 1$. In the latter case, we developed criteria for the existence of structure-preserving solutions and two types of non-structure-preserving, namely path- and cycle-splitting solutions. Finally, we could show that a graph that allows for any of the types of solutions examined can be recognized in polynomial time.

In chapter 8, we generalized these results to all possible cases of instances of the MSSP. For studying twin-constrained Hamiltonian cycles, we introduced the concept of the patching graph of the twin-induced structure of a matching relative to a family of alternating T-cycles. On this basis, heavily drawing on the results of chapter 6, we could show that in the particular case of greedy matchings, the twin-constrained Hamiltonicity of a threshold graph is equivalent with the existence of a family of alternating T-cycles relative to which the patching graph of the twin-induced structure of a greedy matching is connected. By means of an algorithm that, as we could prove on the basis of chapter 6 again, allows us to decide on the question of whether or not there exists such a family of alternating T-cycles relative to a given greedy matching, we eventually arrived at a polynomial-time algorithm for the MSSP.

Finally, in the previous chapter 9, we carried out computational tests for the two algorithms we developed. It was demonstrated that the two algorithms provide excellent results with respect to computational time and, in the case of the heuristic, also with respect to the percentage of the instances solved. Moreover, the results of our computational tests suggested that the algorithms show a stable behaviour under various distributions of the input data and how our two algorithms can be combined to create an even more efficient algorithm. In sum, our results demonstrate that we have indeed solved the MSSP and developed an algorithm that can be expected to work efficiently for practical purposes. This ultimately justifies the approach to the MSSP that we chose in chapter 2, which now can arguably be said to capture the very essence of what the mathematical structure of the MSSP is about.

Concluding this thesis, we will address six questions for further research, the first three of which are directly related to the results of this thesis, while the latter three can be considered generalisations of the work undertaken here.

1. We saw in chapter 3 that the distinct structure of threshold graphs has led to several different characterisations of this class of graphs. Even more characterisations can be found in the literature (see Mahadev and Peled, 1995). In chapter 4, we proved a very strong statement

about alternating T-paths on threshold graphs (Theorem 44), and in chapter 5, based on the alternating path theorem, presented a new, surprisingly simple criterion for the existence of Hamiltonian paths on threshold graphs (Theorem 53). One would expect such strong statements as Theorems 44 and 53 to hold only for a very small class of graphs. Therefore, it would be an interesting topic of research pertaining to the structure of threshold graphs to see whether these theorems can be exploited as a starting point for deriving a new, alternative characterization of this class of graphs.

2. We concluded at the end of chapter 5 that in the case of threshold graphs the twin-constrained Hamiltonian path problem could either be solvable in polynomial time (as the "ordinary" Hamiltonian path problem on threshold graphs) or be **NP**-complete (as the twin-constrained Hamiltonian path problem in the general case). In chapter 8, drawing on the results of the preceding chapters, we could eventually show that for threshold graphs also the twin-constrained Hamiltonian path problem is in **P** and shifted the frontier of what is known to be in **P**. For doing so, some theoretical effort was necessary and we had to rely heavily on the specific structure of threshold graphs. In view of the fact that the twin-constrained Hamiltonian path problem generalizes the "ordinary" Hamiltonian path problem, this might not be too surprising, but it raises the question of whether there exists a class of graphs, for which the "ordinary" Hamiltonian path problem is in **P**, but for which the twin-constrained Hamiltonian path problem is **NP**-complete. There might be a limit such that the specific structure of a graph is not rich enough to allow for a polynomial-time algorithm for the twin-constrained case. We know, for example, that the "ordinary" Hamiltonian cycle problem on 2-regular graphs is (trivially) in **P**, while the "ordinary" Hamiltonian cycle problem on a 3-regular graph (i.e. a 2-regular graph plus the edges of a "twin-node function") is **NP**-complete (Garey, Johnson and Tarjan, 1976). Therefore, as Frits Spieksma (University of Leuven) remarked (oral communication), recognising twin-constrained Hamiltonian 2-regular graphs might well be an **NP**-complete problem. So far, despite some efforts by Frits Spieksma and the author of the present thesis in addressing this topic, this question has remained open. If, however, such a class of graphs could eventually be found, this might well lead to a better understanding of the boundary that separates problems in **P** from those that are **NP**-complete (provided that **P**$\neq$**NP**, of course).

3. Now that the twin-constrained Hamiltonian cycle problem on threshold graphs turned out to be solvable in polynomial time, it might be helpful for practical applications to have a polyhedral description of it, in particular because the problem occurs in conjunction with the cutting-stock problem, which is typically addressed by means of Integer Programming. In chapter 7, we presented a polyhedral description of path- and cycle-splitting solutions, i.e. for a case in which one alternating T-cycle is sufficient to construct a connected patching graph. It would be an interesting to see if this approach can be extended to the case of an arbitrary number of alternating T-cycles that we have addressed without polyhedral means in chapter 8.

4. A natural generalization of the twin-constrained Hamiltonian path problem on threshold graphs would be the "twin-constrained TSP" on threshold graphs. One way of addressing this question would be to examine how well-known general-purpose heuristics for the TSP behave in the twin-constrained case. However, it might be more fruitful to draw on our results of the previous chapters to develop an approach tailored to the case of threshold graphs. After all, chapters 5, 6, 7 and 8 provide many structural insights into constructing, combining and extending ("ordinary" and twin-constrained) Hamiltonian paths on threshold graphs that a heuristic looking for an optimal twin-constrained Hamiltonian cycle could exploit.

5. Another natural generalization of the topic of this thesis is the class of directed graphs that can be seen as the counterpart of threshold graphs: Ferrers digraphs. Introduced by Riguet (1951), Ferrers digraphs can be characterized as those directed graphs that lead to threshold graphs when every arc is replaced by an edge (Mahadev and Peled, 1995). While the directedness of the arcs implies that not all characterisations of threshold graphs can be translated into characterisations of Ferrers digraphs without appropriate amendments, it would be interesting to explore if and how the various structural aspects of Hamiltonian threshold graphs and (twin-constrained) Hamiltonian threshold graphs that we studied in chapters 3 to 8 can be reconstructed for the directed case.

6. Finally, it would be interesting to see if it is possible to generalize our results to a class of graphs that includes the class of threshold graphs and has a more general structure. A promising candidate for such a task would be the class of interval graphs (cf. Remark 52(2)). Interval graphs, which have many applications in scheduling, are graphs in which each node is assigned an interval on the real line (instead of a single number as in the case of threshold graphs), with two nodes being adjacent if and only if their assigned intervals overlap. They are a particularly interesting generalization of threshold graphs in our case because (a) our matching algorithm $TGMA_{\min}$ leads to a maximum cardinality matching also in the case of interval graphs (Moitra and Johnson, 1989) and (b) the Hamiltonian cycle problem on interval graphs can also be solved in polynomial time (Keil, 1985). This implies that some of our results can be expected to be (more or less directly) transferable to the case of interval graphs, and that it is also possible that the twin-constrained Hamiltonian path problem on interval graphs is in **P**. As the present thesis did not only develop a polynomial-time algorithm for the twin-constrained Hamiltonian path problem on threshold graphs, but also examined in chapters 5 to 8 in detail the structural circumstances that allow for such an algorithm in the first instance, we are prepared to understand clearly when and, if so, why particular aspects of the structure of threshold graphs can or cannot be transferred to the case of interval graphs. These insights might well be of help for undertaking such a generalization of our results to the case of interval graphs − a generalization that would move further the frontier of what is known to be solvable in polynomial time.

References

[1] Abouelaoualim, A., K.Ch. Das, W. Fernandez de la Wega, M. Karpinski, Y. Manoussakis, C.A. Martinhon, and R. Saad (2009): Cycles and paths in edge-coloured graphs with given degrees, *Journal of Graph Theory*, pre-published online.

[2] Abueida, A. and R. Sritharan (2006): Cycle extendability and Hamiltonian cycles in chrodal graph classes, *SIAM Journal on Discrete Mathematics* 20(3), pp. 669 - 681.

[3] Angluin, D. and L.G. Valiant (1979): Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences* 18(2), pp. 155-193.

[4] Anily, S., J. Bramel, A. Hertz (1999): A $\frac{5}{3}-$approximation algorithm for the clustered traveling salesman tour and path problems, *Operations Research Letters* 24, pp. 29-35.

[5] Arkin, E.M., R. Hassin, L. Klein (1994): Restricted delivery problems on a network, *Networks* ,29 pp. 205-216.

[6] Balas, E. (2002): The Price Collecting Traveling Salesman Problem and Its Applications. In: G. Gutin and A.P. Punnen (eds.): *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publisher: Dordrecht, pp. 663-696.

[7] Balas, E., M. Fischetti and W.R. Pulleyblank (1995): The precedence-constrained asymmetric traveling salesman polytope, *Mathematical Programming* 68, pp. 241-265.

[8] Bang-Jensen, J. and G. Gutin (1997): *Discrete Mathematics* 165/166, pp. 39-60.

[9] Bankfalvi, M and Z. Bankfalvi (1968): Alternating Hamiltonian Circuits in two-coloured complete graphs. In: *Theory of Graphs (Proc. Colloq. Tihany)*, Academic Press: New York, pp. 11-18.

[10] Barvinok, A., E.K. Gimadi and A.I. Serdyukov (2002): The Maximum TSP. In: G. Gutin and A.P. Punnen (eds.): *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publisher: Dordrecht, pp. 585-608.

[11] Bellmore, M. and G.L. Nemhauser (1968): The Traveling Salesman Problem: A Survey, *Operations Research* 16(3), pp. 538-558.

[12] Bermond, J.C. (1978): Hamiltonian Graphs. In: Beineke and Wilson (eds): *Selected Topics in Graph Theory*. Academic Press: London, 1978.

[13] Bertossi, A.A. (1983): Finding Hamiltonian circuits in proper interval graphs, *Information Processing Letters* 17, pp. 97-101.

[14] Biggs, N.L., E.K. Lloyd and R.J. Wilson (1976): *Graph Theory 1736-1936*. Clarendon Press: Oxford.

[15] Blazewicz, J., M. Kasprzaka, B. Leroy-Beaulieuc and D. de Werra (2008): Finding Hamiltonian circuits in quasi-adjoint graphs, *Discrete Applied Mathematics* 156(13), pp. 2573-2580

[16] Bollobás, B. and P. Erdös (1976): Alternating Hamiltonian cycles, *Israel Journal of Mathematics* 23, pp. 126-131.

[17] Bollobás, B., T.I. Fenner and A.M. Frieze (1987): An algorithm for finding Hamiltonian paths and cycles in random graphs. *Combinatorica* 7(4), pp. 327-341.

[18] Brandstädt, A., F.F. Dragan and E. Köhler (2000): Linear time algorithms for hamiltonian problems on (claw, net)-free graphs. *SIAM Journal on Computing* 30(5), 1662-1677.

[19] Brandstädt, A., V.B. Le and J.B. Spinrad (1999): *Graph Classes - A Survey.* SIAM monographs on discrete mathematics and applications. SIAM: Philadelphia.

[20] Broder, A.Z., A.M. Frieze and E. Shamir (1994): Finding hidden Hamiltonian cycles. *Random Structures and Algorithms* 5(3), pp.395-410.

[21] Brunacci, F. (1988): Two useful tools for constructing hamiltonian circuits, *European Journal of Operational Research* 34, pp. 231-236.

[22] Burkhard, R.E., V.G. Deineko, R. van Dal, J.A.A. van der Veen and G.J. Woeginger (1998): Well-Solvable Cases of the Travelling Salesman Problem: A Survey, *SIAM Review* 40(3), pp. 496-546.

[23] Chen, C.C. and D.E. Daykin (1976): Graphs with Hamiltonian Cycles Having Adjacent Lines Different Colors, *Journal of Combinatorial Theory B* 21, pp. 135-139.

[24] Chisman, J.A. (1975): The clustered traveling salesman problem, *Computers & Operations Research* 2(2), pp. 115-119

[25] Chvátal, V (1985): Hamiltonian Cycles. In: E.L. Lawler, J,K, Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys (eds.): *The Traveling Salesman Problem.* John Wiley & Sons: New York, pp. 403-430.

[26] Chvátal, V. and P.L. Hammer (1973): *Set packing problems and threshold graphs.* CORR 73-21. University of Waterloo, Canada, August 1973.

[27] Chvátal, V. and P.L. Hammer (1977): Aggregation of inequalities in integer programming. In: P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser (eds.): *Studies in Integer Programming. Annals of Discrete Mathematics 1.* North-Holland Publishing Company: Amsterdam, pp. 145-162.

[28] Cobham, A. (1965): The Intrinsic Computational Difficulty of Functions. In: Y. Bar-Hille (ed.): *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*. North-Holland Publishing Company: Amsterdam, pp. 24-30.

[29] Cook, S.A. (1971): The Complexity of Theorem Proving Precedures, *Proceedings of the 3rd ACM Symposium on the Theory of Computing*. ACM, pp. 151-158.

[30] Daykin, D.E. (1976): Graphs with Cycles Having Adjacent Lines Different Colors, *Journal of Combinatorial Theory B* 20, pp. 149-152.

[31] Devroye, L. (1986): *Non-Uniform Random Variate Generation*. Springer: Berlin.

[32] Dinga, C., Y. Cheng and M. He (2007): Two-Level Genetic Algorithm for Clustered Traveling Salesman Problem with Application in Large-Scale TSPs, *Tsinghua Science & Technology* 12(4), pp. 459-465.

[33] Dumas, Y., J. Desrosier, E. Gelinas and M.M. Solomon (1995): An Optimal Algorithm for the Traveling Salesman Problem with Time Windows, *Operations Research* 43 (2), pp. 367-371.

[34] Ecker, K. and S. Zaks (1977): *On a graph labelling problem*. Bericht 99, Gesellschaft für Mathematik und Datenverarbeitung mbH: Bonn.

[35] Edmonds, J. (1965a): Paths, trees and flowers, *Canadian Journal of Mathematics* 17, pp. 449-467.

[36] Edmonds, J. (1965b): Minimum partition of a matroid into independent subsets, *Journal of Research of the National Bureau of Standards* 69B, pp. 67-72.

[37] Feremans, C., M. Labbé and G. Laporte (2002): A Comparative Analysis of Several Formulations for the Generalized Minimum Spanning Tree Problem, *Networks* 39(1), pp. 29-34.

[38] Feremans, C., M. Labbé and G. Laporte (2003): Generalized network design problems, *European Journal of Operational Research* 148(1), pp. 1-13

[39] Feremans, C., M. Labbé, A.N. Letchford and J.J. Salazar (2009): On generalized network design polyhedra. Submitted to *Networks* January 2009. (Retrieved from *http://www.lancs.ac.uk/staff/letchfoa/publications.htm*, 10/08/2009.)

[40] Fischetti, M., J.J. Salazar-Gonzáles and P. Toth (2002): The Generalized Traveling Salesman Problem and Orienteering Problems. In: G. Gutin and A.P. Punnen (eds.): *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publisher: Dordrecht, pp.609-662.

[41] Frieze, A.M. (1988): Finding Hamilton cycles in sparse random graphs. *Journal of Combinatorial Theory B* 44, pp. 230-250.

[42] Garey, M.R., D.S. Johnson and E. Tarjan: The planar hamiltonian circuit problem is NP-complete, *SIAM Journal on Computing* 5, pp. 704-714.

[43] Gavish, B. and S.C. Graves (1978): *The travelling salesman problem and related problems.* Working Paper OR-078-78, Operations Research Center, MIT: Cambridge, MA.

[44] Gendreau, M., G. Laporte and D. Vigo (1999): Heuristics for the traveling salesman problem with pickup and delivery. *Computers and Operations Research* 26(7), pp. 699-714.

[45] Ghouila-Houri, A. (1962): Characterisation des matrices totalement unimodulaires. In: *Comptes Rendus Hebdomadaires des Seances de L'Academie des Sciences* 254, pp. 1192-1194.

[46] Gilmore, P.C. and R.E. Gomory (1961): A linear programming approach to the Cutting Stock Problem, *Operations Research* 9, pp. 849-859.

[47] Gilmore, P.C. and R.E. Gomory (1963): A linear programming approach to the Cutting Stock Problem: Part II, *Operations Research* 11, pp. 863-888.

[48] Gilmore, P.C., Lawler,E.L., and D.B. Shmoys(1986): Well-solved special cases. In: E.L. Lawler, J,K, Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys (eds.): *The Traveling Salesman Problem.* John Wiley & Sons: New York, pp. 87-144.

[49] Golumbic, M.C. (1980): *Algorithmic Graph Theory and Perfect Graphs.* Academic Press: New York.

[50] Gould, R.J. (1991): Updating the Hamiltonian problem - a survey, *Journal of Graph Theory* 15(2), pp. 121-157.

[51] Gould, R.J. (2003): Advances on the Hamiltonian Problem - A Survey. *Graphs and Combinatorics* 19(1), pp. 7-52.

[52] Goulimis, C. (2004): Minimum score separation - an open combinatorial problem associated with the cutting stock problem. *Journal of the Operational Research Society*, 55, pp. 1367-1368.

[53] Gutin, G. (2009): Traveling Salesman Problem. In: C.A. Floudas, P.M. Pardalos (eds.): *Encyclopedia of Optimization*, 2nd ed., Springer: Berlin, pp. 3935-3944.

[54] Gutin, G. and A.P. Punnen (eds.) (2002): *The Traveling Salesman Problem and Its Variations.* Kluwer Academic Publisher: Dordrecht.

[55] Guttmann-Beck, N., R. Hassin, S. Khuller and B. Raghavachari (2000): Approximation Algorithms with Bounded Performance Guarantees for the Clustered Traveling Salesman Problem, *Algorithmica* 28(4), pp. 422-437.

[56] Häggkvist, R. (1979): On F-Hamiltonian Graphs. In: J. A. Bondy and U. S. R. Murty (eds): *Graph Theory and Related Topics*, Academic Press: New York, pp. 219-231.

[57] Hamilton, W. R. (1858): Account of the Icosian Calculus. *Proceedings of the Royal Irish Academy* 6, 1858.

[58] Hammer, P.L., T. Iberaki and B. Simeone (1981): Threshold Sequences, *SIAM Journal of Algebraic Discrete Methods* 2, pp. 39-39.

[59] Harary, F. and U.N. Peled (1987): Hamiltonian Threshold Graphs. *Discrete Applied Mathematics* 16, pp. 11-15.

[60] Henderson, P.B. and Y. Zalcstein (1977): A graph-theoretic characterization of the PV_{chunk} class of synchronizing primitives. *SIAM Journal of Computing* 6, pp. 88-108.

[61] Hoffman, A.J. and P. Wolfe (1985): History. In: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.): *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*. Wiley: Chichester, pp. 1-16.

[62] Hung, R.W. and M.S.Chang (2005): Linear-time algorithms for the Hamiltonian problems on distance-hereditary graphs, *Theoretical Computer Science* 341(1), pp. 411-440.

[63] Johnson, D.S, and C.H. Papadimitrou (1985): Computational Complexity. In: Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.): *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*. Wiley: Chichester, pp. 37-86.

[64] Jongen, H.T., K. Meer and E. Triesch (2004): *Optimization Theory*. Kluwer Academic Publishers: Dordrecht.

[65] Jongens, K. and T. Volgenant (1985): The symmetric clustered traveling salesman problem, *European Journal of Operational Research* 19(1), pp. 68-75.

[66] Jünger, M., G. Reinelt and G. Rinaldi (1995): The Traveling Salesman Problem. In: M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (eds.): *Network Models. Handbook of Operations Research and Management Science*. Vol. 7, pp. 225-330.

[67] Kaiser, T., Z. Ryjáček, D. Král, M. Rosenfeld, and H.-J. Voss (2007): Hamilton Cycles in Prisms, *Journal of Graph Theory* 56(4), pp. 249-269.

[68] Kabadi, S.N. and A.P. Punnen (2002): The Bottleneck TSP. In: G. Gutin and A.P. Punnen (eds.): *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publisher: Dordrecht, pp. 697-736.

[69] Kano, M. and X. Li (2008): Monochromatic and Heterochromatic Subgraphs in Edge-Colored Graphs - A Survey, *Graphs and Combinatorics* 24(4), pp. 237-263.

[70] Karp, R.M. (1972): Reducibility among Combinatorial Problems. In: R.E. Miller and J.W. Thatcher (eds.): *Complexity of Computer Computations*. Plenum Press: New York, pp. 85-103.

[71] Keil, J.M. (1985): Finding Hamiltonian circuits in interval graphs, *Information Processing Letters* 20, pp. 201-206.

[72] Kirkman, T.P (1856): On the representation of polyhedra. *Philosophical Transactions of the Royal Society London A* 146, 413-418.

[73] Kocay, W. (1992): An extension of the multi-path algorithm for finding Hamilton cycles, *Discrete Mathematics* 101, pp. 171-188.

[74] Koren,M. (1973): Extreme degree sequences of simple graphs. *Journal of Combinatorial Theory B*, 15, pp. 213-224.

[75] Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.) (1985): *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*. Wiley: Chichester.

[76] Laporte, G., A. Asef-Vaziri, C. Sriskandarajah (1996): Some Applications of the Generalized Travelling Salesman Problem. *Journal of the Operational Research Society* 47, 1461-1467.

[77] Laporte, G. and U. Palekar (2002): Some applications of the clustered travelling salesman problem, *Journal of the Operational Research Society* 53, pp. 972-976.

[78] Laporte, G., J.-Y. Potvin and F. Quilleret (1997): A tabu search heuristic using genetic diversification for the clustered traveling salesman problem, *Journal of Heuristics* 2(3), pp. 187-200.

[79] Laporte, G. and F. Semet (1999): Computational evaluation of a transformation procedure for the symmetric travelling salesman problem. *INFOR* 37, pp. 114-120.

[80] Mahadev, N.V.R. and U.N. Peled (1994): Longest cycles in threshold graphs. *Discrete Mathematics* 135, pp.169-176.

[81] Mahadev, N.V.R. and U.N. Peled (1995): *Threshold Graphs and Related Topics. Annals of Discrete Mathematics 56*. Elsevier: Amsterdam.

[82] Micali, S., and V. V. Vazirani (1980): An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. *Proceedings of the 21st Annual Symposium on the Foundations of Computer Science*, pp. 17-27.

[83] Moitra, A, and R.C. Johnson (1989): A parallel algorithm for maximum matching on interval graphs. *Proceedings of the 1989 International Conference on Parallel Processing.* Pennsylvania State University Press: University Park, PA, pp. III/114 - III/120.

[84] Mulder, H.M. (1992): Julius Petersen's theory of regular graphs, *Discrete Mathematics* 100, pp. 157-175

[85] Müller-Merbach, H. (1983): Zweimal Travelling Salesman. *DGOR-Bulletin* 25, pp. 12-13.

[86] Nemhauser, G. and L. Wolsey (1999): *Integer and Combinatorial Optimization.* John Wiley: New York.

[87] Orlin, J. (1977): The minimal integer separator of a threshold graph. In: P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser (eds.): *Studies in Integer Programming. Annals of Discrete Mathematics* 1. North-Holland Publishing Company: Amsterdam, pp. 415-419.

[88] Orman, A.J. and H.P. Williams (2004): *A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem.* LSEOR 04.67: London.

[89] Papadimitrou, C.H. and K. Steiglitz (1998): *Combinatorial Optimization. Algorithms and Complexity.* Dover Publications: Mineola, New York.

[90] Petersen, J. (1891): Die Theorie der regulären Graphs. *Acta Mathematica* 15, pp. 193-220.

[91] Picouleau, C. (1994): Complexity of the hamiltonian cycle in regular graph problem. *Theoretical Computer Science* 131, pp. 463-473.

[92] Posa, L. (1976): Hamiltonian circuits in random graphs, *Discrete Mathematics* 14, pp. 359-364.

[93] Ramalingam, G. and C.P. Rangan (1988): A unified approach to domination problems on interval graphs. *Information Processing Letters* 27, pp. 271-274.

[94] Renaud, J. and F.F. Boctor (1998): An efficient composite heuristic for the symmetric generalized traveling salesman problem, *European Journal of Operational Research* 108(3), pp. 571-584

[95] Riguet, J. (1951): Les relations de ferres. *Comptes Rendus de l'Academie de Sciences, Paris, Serie I, Mathematique* 232, pp. 1729-1730.

[96] Schrijver, A. (1986): *Theory of Linear and Integer Programming.* Wiley: Chichester.

[97] Schrijver, A. (2003): *Combinatorial Optimization. Polyhedra and Efficiency.* 3 volumes. Springer: Berlin.

[98] Shields, I.B. (2004): *Hamilton Cycle Heuristics in Hard Graphs*, PhD thesis, North Carolina State University.

[99] Shih, W.K., T.C. Chern and W.L. Hsu (1992): An O(n2 log n) time algorithm for the Hamiltonian cycle problem on circular-arc graphs, *SIAM Journal on Computing* 21, pp. 1026–1046.

[100] Shufelt, J.A. and H.J. Berliner (1994): Generating Hamiltonian Circuits without backtracking from errors. *Theoretical Computer Science* 132, pp. 347-375.

[101] Truemper, K (1977): Unimodular Matrices of Flow Problems with Additional Constraints, *Networks* 7: pp. 343-358.

[102] Vandegriend, B. and J. Culberson (1998): The $G_{n,m}$ Phase Transition is Not Hard for the Hamiltonian Cycle Problem, *Journal of Artificial Intelligence Research* 9, pp. 219-245.

[103] Voigt, B.F. (1831): *Der Handlungsreisende, wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein.* Von einem alten Commis-Voyageur, Ilmenau. (Republished 1981, Verlag Bernd Schramm: Kiel.)

[104] Wagner, I.A. and A.M. Bruckstein (1999): Hamiltonian(t) - An Ant-Inspired Heuristic for Recognizing Hamiltonian Graphs, *Proceedings of the 1999 Congress on Evolutionary Computation*. IEEE Press.

[105] Yao, A. (1996): A Note on Alternating Cycles in Edge-Coloured Graphs, *Journal of Combinatorial Theory B* 68, pp. 222-225.

A MSSPH 3.6: C++ source code

```
// ****************************************************************************
// ----------------------------------------------------------------------------
// ************************   MSSP-Heuristic 3.6   ****************************
// ----------------------------------------------------------------------------
// ****************************************************************************

// 04 April 2010, Kai Helge Becker

// ****************************************************************************
// Header files, constants, global variables
// ****************************************************************************

#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

// Global variables and constants
// Constants
   int const numinst = 1000000;      // number of instances
   int const numbox = 20;            // number of boxes
   int const numsc  = 2 * numbox;    // number of scores
   int const minwidth = 41;           // minimal width of each score
   int const maxwidth = 70;          // maximal width of each score
   int const thradj = 70;            // threshold for adjacency
   int const thrstrong = thradj / 2; // threshold for strong node (= thradj/2)
   int const empty = 999;            // flag for empty variable entry
                                     // (used w/ matlist, unconnode, lastmatch)
   int const nocomp
           = (numbox+(numbox%2))/2;  // no of components (chain + no of cycles)
   int const b
            = maxwidth - minwidth + 1; // no of different score sizes
   double const b2 = b;

// Variables

   double probability[b];            // pdf of triangular distribution
   double endofpartition[b];         // cdf of triangular distribution
   int sto0;                         // random integer number
   double stochastic;                // random number in [0,1[ derived from sto0
   int wbox[numsc];                  // width of all scores
   int i, j, k;                      // loops
   int stack;                        // stack
   int stackinv;
   int ordsc[numsc];                 // array index of ith smallest score
   int invordsc[numsc];              // inverse function of the above
   int instance;                     // counters for instances and cases
   int feacounter;
   int infcounter;
   int toomanyweak;
   int noncontwin;
   int uncon;
   int perfmat;
   int poormat;
   int suffmat;
   int yeahcounter;
   int completechain;
   int wscyclesandchain;
   int completechaininv;
   int wscyclesandchaininv;

   int twinno1b;                  // place of an unconnectable box (case 1b)
   double runtime1;               // running time
   //double runtime2;             // running time without generating instances
   //long time4rand;              // time for generating one instance
   int adjlist[numsc] [numsc];    // adjacency list (based on sorted indices)
```

```
   int matlist[numsc];           // matching list (based on sorted indices)
   int matlistinv[numsc];        // matching list for inverse matching
   int matcard;                  // cardinality of matching
   int matcardinv;               // cardinality of inverse matching
   int unconpointer;             // number of unconnected nodes
   int unconpointerinv;          // same for inverse matching
   int twinnomat;                // place of twinnode for matching (sorted ind)
   int twinnomatinv;             // same 4 inverse matching
   int lastmatch;                // place of last matched node (sorted ind)
   int lastmatchinv;             // same 4 inverse matching
   int weakeststrong;            // place of weakest strong node (sorted ind)
   int unconnode[numsc];         // place of unconected nodes (sorted ind)
   int unconnodeinv[numsc];      // place of unconectd node in inverse matching
   int smallestuncon;            // smallest unconnected node (sorted ind)
   int smallestunconinv;          // for INV case
   int twin[numsc];              // place of twin node (sorted ind)
   int analysed[numsc];          // flag if node already inclded in chain/cycle
   int analysedinv[numsc];        // for INV case
   int component[nocomp] [numsc];// nodes in chain [0] and cycles [1..nocomp-1]
   int componentinv[nocomp] [numsc]; // for INV case
   int lengthofcomponent[nocomp];// length of component (chain is component 0)
   int lengthofcomponentinv[nocomp]; // for INV case
   int lengthchain[(numsc+1)];   // distribution of chain length
   int lengthchaininv[(numsc+1)];

   int currentcomponent;         // no of cycle being analysed
   int smallestconnotana;        // smallest connected node not analysed yet
   int smallestconnotanainv;         // for INV case
   int nocycles;                 // number of cycles
   int nocyclesinv;              // number of cycles in INV case
   int strongestnode[nocomp];    // characteristic of each cycle, used from
1..nocycles
   int strongestnodeinv[nocomp];
   int weakeststrongest;         // characteristics of all cycles
   int weakeststrongestinv;
   int pss[nocomp];
   int pwss;
   int pssinv[nocomp];
   int pwssinv;

   int ordcyc[nocomp];            // order of cycles according to weakness of
                                  // strongest node, for checkresult(8)
   int ordcycinv[nocomp];         // chckresult 11
   int placeofstrongestnode[nocomp]; // pl of s node in original order of cycles
                                     // input for component[nocomp] [xxx]
                                     // used for checkresult(8)
   int placeofstrongestnodeinv[nocomp]; // chckresult (11)
   int currentplace;              // next place in array "result" to be filled
   int currentplaceinv;
   int numberofelsecases;         // more than one cycle
   int numberofelsecasesinv;

   int distofcyclesstart[nocomp];// distribution of cycles b4 cycle analysis
   int distofcyclesleft[nocomp]; // distribution of cycles after cycle analysis
   int distofcyclesstartinv[nocomp]; // same for INV case
   int distofcyclesleftinv[nocomp];
   int prob;                     // counter for some problematic cases
   int probinv;                      // smallest for INV

   int result[numsc];                               // RESULT
   int resultcounter;
   int checkcasecounter[130];
   int problemcounter;
   int status;                                      // result of result check

// ****************************************************************************
```

```
// Function for checking results
// ****************************************************************************
  int checkresult (int subcase)
  {
    // Test
    // result[17] = 0;

    // Count check
    ++resultcounter;
    ++checkcasecounter[subcase];

    // Local variable
    int problem;                                        // flag for problem
    problem = 0;

    // Checking if there is a "999" node
    for(i=0; i<numsc; ++i)
    {
      if (result[i] == 999)
      {
        problem = 1;
        cout << endl << "999 case" << endl;
      }
    }

    // Checking twin node and matching characteristic except for last pair
    for(i=0; i<=(numsc-4); i=i+2)
    {
       // Checking twin node chacteristic
       if (  (((ordsc[result[i]]) % 2) == 0)                    // ordsc[i] even
          && (ordsc[result[i]] != ((ordsc[result[(i+1)]])-1)))
       {
         problem = 1;
         break;
       }
       if (  (((ordsc[result[i]]) % 2) == 1)                    // ordsc[i] odd
          && (ordsc[result[i]] != ((ordsc[result[(i+1)]])+1)))
       {
         problem = 1;
         break;
       }

       // Checking matching characeristic
       if (wbox[ordsc[result[(i+1)]]] + wbox[ordsc[result[(i+2)]]] < thradj)
       {
         problem = 1;
         break;
       }
    }

    // Checking twin node characteristic for last pair
    if (  ((ordsc[result[(numsc-2)]] % 2) == 0)              // ordsc[i] even
       && (ordsc[result[(numsc-2)]] != ((ordsc[result[(numsc-1)]])-1)))
       problem = 1;
    if (  ((ordsc[result[(numsc-2)]] % 2) == 1)              // ordsc[i] odd
       && (ordsc[result[(numsc-2)]] != ((ordsc[result[(numsc-1)]])+1)))
       problem = 1;

    // For test if no problem
    // if ((problem == 0) && (subcase == 122)) cout << "***" << endl;

    // Consequences if problem
    if (problem == 1)
    {
      cout << "***** problem with subcase " << subcase << " *****" << endl;
```

```
    // chainanalysisok = 1;

    // Checking analysis of cycles and chain
    for(i=0; i<=nocyclesinv; ++i)
    {
      for(j=1; j<lengthofcomponentinv[i]; j=j+2)
      {
        if ((ordsc[componentinv[i] [j]] != (ordsc[componentinv[i] [j-1]]+1))
          && (ordsc[componentinv[i] [j]] != (ordsc[componentinv[i] [j-1]]-1)))
           cout << "twin wrong " << endl;
      }
      for(j=1; j<lengthofcomponentinv[i]-2; j=j+2)
      {
        if (wbox[ordsc[componentinv[i] [j]]]
            + wbox[ordsc[componentinv[i] [j+1]]]
           < thradj)
           cout << "mate wrong " << endl;
      }
    }
    // Printing variables for building result
    cout << "ordcycinv " << ordcycinv[1] << " " << ordcycinv[2] << endl;
    cout << endl << endl;

    cout << "wbox[i]:  ";
    for(i=0; i<numsc; ++i)
      cout << wbox[i] << "  ";
    cout << endl;
    cout << "wbox[ordsc[i]]:  ";
    for(i=0; i<numsc; ++i)
      cout << wbox[ordsc[i]] << "  ";
    cout << endl;
    cout << "result:  ";
    for(i=0; i<numsc; i=i+2)
      cout << wbox[ordsc[result[i]]] << "(" << ordsc[result[i]] << ")--("
           << ordsc[result[(i+1)]] << ")" << wbox[ordsc[result[(i+1)]]]
           << "  ";
    cout << endl << "number of cycles: " << nocycles << endl; // non-INV only
    cout << endl;
    ++problemcounter;
    // Note: The following printout is correct only in INV case
    for(i=0; i<=nocyclesinv; ++i)
    {
      for(j=0; j<lengthofcomponentinv[i]; ++j)
        cout << wbox[ordsc[componentinv[i] [j]]] << "-";
      cout << endl;
    }
    cout << endl << endl;
  }

  // Return problem status
  return problem;
}
```

```
// ********************************************************************************
// Function main starts & initialisation
// ********************************************************************************
int main( )
{

// Welcome
   cout << "Welcome to MSSP-Heuristic 3.6." << endl;


// Initialisation
   feacounter = 0;               // initialising feasible instances counter
   infcounter = 0;               // initialising infeasible instances counter
   toomanyweak = 0;              // initialising case counters
   noncontwin  = 0;
   uncon = 0;
   perfmat = 0;
   poormat = 0;
   suffmat = 0;
   yeahcounter = 0;

   completechain = 0;
   wscyclesandchain = 0;

   completechaininv = 0;
   wscyclesandchaininv = 0;

   prob = 0;
   probinv = 0;

   resultcounter = 0;
   for(i=0; i<130; ++i)
     checkcasecounter[i] = 0;

   problemcounter = 0;
   numberofelsecases = 0;

   for(i=0; i<(numsc+1); ++i)
        lengthchain[i] = 0;
      for(i=0; i<nocomp; ++i)
      {
        distofcyclesstart[i] = 0;
        distofcyclesleft[i] = 0;
      }

   for(i=0; i<(numsc+1); ++i)
     lengthchaininv[i] = 0;
   for(i=0; i<nocomp; ++i)
   {
     distofcyclesstartinv[i] = 0;
     distofcyclesleftinv[i] = 0;
   }


// Calculate probabilities for triangular distribution
probability[0] = 2/(b2*b2);
probability[(b-1)] = probability[0];
for (i = 1; i < b/2; ++i)
{
       probability [i] = probability [(i-1)] + 4/(b2*b2);
       probability [(b-1-i)] = probability [i];
}
```

```
// Calculate partition of [0,1] interval for triangular distribution
endofpartition[0] = probability[0];
for (i=1; i < b; ++i)
{
       endofpartition [i] = endofpartition [(i-1)] + probability [i];
}


/*
for (i=0; i<b; ++i)
{
       cout << i << "     " << probability[i] << "    " << endofpartition[i] << "
";
}
*/

// Initialising random mumbers
srand( (unsigned) time(NULL) );

// ********************************************************************************
// Start of instances loop
// ********************************************************************************

for (instance = 0; instance < numinst; ++instance)
{

// cout << RAND_MAX << "HHHH   ";

       // Producing triangularly distributed random numbers for scores
   for (i=0; i < numsc; ++i)
   {
       // Generate random number between 0 and 1
          sto0 = rand();
        stochastic = static_cast<double> (sto0) / 32767;
          // cout << stochastic << "   ";
       // Check the probability interval of which the number is a member
          j = 0;
          while (stochastic > endofpartition[j]) {++j;};
       // Calculate the triangularly distributed number
        wbox[i] = minwidth + j;
   }

/*
// Producing uniformly distributed random numbers for scores
   //time4rand = clock();
   for(i = 0; i < numsc; ++i)
   {
      wbox[i] = minwidth + (rand() % (maxwidth - minwidth + 1));
   }
*/

// Test data
/*
   wbox[19] = 1; wbox[9] = 30; //wbox[10] = 35; wbox[11] = 36;

   for(i = 10; i < 19; ++i)
   {
     wbox[i] = 20 + (rand() % 6);
   }
   for(i = 12; i < 20; ++i)
   {
      wbox[i] = 10 + (rand() % 11);
   }
*/

/*AAA
```

```
// Output instance
   cout << endl;
   for(i = 0; i < numsc; ++i)
   {
      cout << wbox[i] << "   ";
   }
   cout << endl;
*/
   // Running time without generating instances
   // time4rand = clock() - time4rand;
   // cout << time4rand << endl;

// ***************************************************************************
// CASE (01): Less than numbox - 1 strong nodes
// ***************************************************************************
   j = 0;
   for(i = 0; i < numsc; ++i)
   {
      if (wbox[i] >= thrstrong)
         ++j;
   }
   if (j < numbox - 1)
   {
      ++infcounter;
      ++toomanyweak;
      //AAAcout << "inf: too many weak nodes, namely: " << numsc-j << endl;
      continue;
   }

// ***************************************************************************
// Sorting scores from smallest to largest
// ***************************************************************************
   // Initialising order
   for(i = 0; i < numsc; ++i)
   {
      ordsc[i] = i;
   }
   // Starting sorting procedure
   for(i = 1; i < numsc; ++i)
   {
      for(j = i-1; j >= 0; --j)
      {
         //cout << i << "  " << ordsc[i] << "  "
         //     << j << "   " << ordsc[j] << endl;
         if (wbox[i] < wbox[ordsc[j]])
         {
            ordsc[j+1] = ordsc[j];
            ordsc[j] = i;
            //cout << "after change  " << i << "  " << ordsc[i] << "  "
            //     << j << "    " << ordsc[j] << endl;
         }
         //else { cout << "no change, next i" << endl; break; }
      }
   }

/*
```

```
// Output sorted instance

   cout << "order: ";
   for(i = 0; i < numsc; ++i)
   {
         cout << wbox[ordsc[i]] << "   ";
   }
   cout << endl;
*/

// ****************************************************************************
// CASE (02): Two non-connectable twin nodes
// ****************************************************************************
   j = 0;
   for(i = 0; i < numbox; ++i)
   {
      if (wbox[2*i] + wbox[ordsc[(numsc-1)]] < thradj)
        {
           if (wbox[2*i + 1] + wbox[ordsc[(numsc-1)]] < thradj)
              {
               ++j;
               //AAAtwinno = i;
              }
        }
   }
   if (j >= 1)
   {
      ++infcounter;
      ++noncontwin;
      /*
      // Output instance
      for(k = 1; k <= numsc; ++k)
      {
         cout << wbox[k] << "   ";
      }
      cout << endl;
      // Output sorted instance
      cout << "order: ";
      for(k = 1; k <= numsc; ++k)
      {
         cout << wbox[ordsc[k]] << "   ";
      }
      cout << endl;
      */
      //AAAcout << "inf: two non-connectable twin nodes at pair "
      //        << twinno << endl;
      continue;
   }

// ****************************************************************************
// CASE (03): Three non-connectable nodes
// ****************************************************************************
   j = 0;
   for(i = 0; i < numsc; ++i)
   {
      if (wbox[i] + wbox[ordsc[(numsc-1)]] < thradj)
         ++j;
   }
   if (j >= 3)
   {
      ++infcounter;
      ++uncon;
      //AAAcout << "inf: too many unconnectable nodes,"
      //        << "largest number has order  " << ordsc[numsc] << endl;
      continue;
   }
```

```
// ***************************************************************************
// Matching algorithm (TGMAmin)
// ***************************************************************************

// Step 1: List with adjacent nodes disregarding twin nodes
   for(i = 0; i < numsc; ++i)  // general adjacency list
   {
     for(j = 0; j < numsc; ++j)
     {
       if (wbox[ordsc[i]] + wbox[ordsc[j]] >= thradj)
       {
         adjlist[i] [j] = 1;
       }
       else
       {
         adjlist[i] [j] = 0;
       }
     }
   }

   for (i=0; i<numsc; ++i)     // generating inverse function of ordsc[]
   {
     invordsc[ordsc[i]] = i;
   }

   for (i = 0; i < numbox; ++i)// twin nodes cannot be connected to eachother
   {
     adjlist[invordsc[(2*i)]] [invordsc[(2*i+1)]] = 0;
     adjlist[invordsc[(2*i+1)]] [invordsc[(2*i)]] = 0;
   }

// Step 2: Initialise matching list and counters
   for (i = 0; i < numsc; ++i)
   {
     matlist[i] = empty;
   }
   matcard = 0;
   unconpointer = 0;
   for (i = 0; i < numsc; ++i)
     {
       unconnode[i] = empty;
     }
   lastmatch = empty;

// Step 3: Matching algorithm
   for(i=0; i<numsc; ++i)// check all nodes
   {
     if (matlist[i] == empty)// does node need a mate?
     {
       for (j=(i+1); j<numsc; ++j)// look for a mate for node i
       {
         if (  (adjlist[i] [j] == 1)
            && (matlist[j] == empty))// if mate found
         {
           matlist[i] = j;
           matlist[j] = i;
           lastmatch = i;
           ++matcard;
           break;
         }
       }
       if (matlist[i] == empty)   // if there still is no mate:
       {                          // do twin node swap or finally acquiesce
         if (ordsc[i] % 2 == 0)     // find out twin node number
         {
```

```
          twinnomat = invordsc[(ordsc[i]+1)];
        }
        else
        {
          twinnomat = invordsc[(ordsc[i]-1)];
        }
        if                          // twin node swap possible?
        (  (wbox[ordsc[i]]+wbox[ordsc[twinnomat]]>=thradj)// match with twin?
        && (matlist[twinnomat] == empty)                  // twin unmatched?
        && (lastmatch != empty)                           // exchange possble?
        && (twinnomat > i)                                // twin larger?
        && (wbox[ordsc[lastmatch]]+wbox[ordsc[twinnomat]]>=thradj))// lastmtch
        {                                                         // with twin?
          matlist[i] = matlist[lastmatch];
          matlist[lastmatch] = twinnomat;
          matlist[twinnomat] = lastmatch;
          matlist[matlist[i]] = i;
          lastmatch = i;
          ++matcard;
        }
        else                        // otherwise: one more unconnected node
        {
          ++unconpointer;
          unconnode[(unconpointer-1)] = i;
        }
      }
    }
  }

/*
// Output matching list and unconnected nodes
   for(i=0; i<numsc; ++i)
   {
     cout << matlist[i] << " ";
   }
   cout << endl;

   for(i=0; i<numsc; ++i)
   {
     cout << unconnode[i] << " ";
   }
   cout << endl;
   cout << "matcard: " << matcard;
   cout << "  no of unconnodes:  " << unconpointer << endl;
*/

// ***************************************************************************
// CASE (04): Perfect matching (#M = n)
// ***************************************************************************
   if (matcard == numbox)
   {
     ++perfmat;
     ++feacounter;

/*AAA
     cout << "fea: perfmatch with card: " << matcard << "  ";
     for (i=0; i<numsc; ++i)
     {
       cout << " w1= " << wbox[ordsc[i]] << " w2= "
            << wbox[ordsc[matlist[i]]] << " -- ";
     }
     cout << endl;
*/

     continue;
   }
```

```
// ***************************************************************************
// CASE (05): Poor matching (#M < n-1)
// ***************************************************************************
   if (matcard < (numbox-1))
   {
     ++poormat;
     ++infcounter;

/*AAA
     cout << "inf: poor matching with card: " << matcard
          << " and uncon nodes: ";
     for (i=0; i<numsc; ++i)
     {
       if (unconnode[i] != empty)
         cout << unconnode[i] << " w= " << wbox[ordsc[unconnode[i]]]
              << " " << endl;
     }
     cout << endl;
*/
     continue;
   }

// ***************************************************************************
// CASE (06): Sufficient matching (#M = n-1)
// ***************************************************************************

  for (i=0; i<numsc; ++i)  // Find weakest strong node
  {
    if (wbox[ordsc[i]] >= thrstrong)
    {
       weakeststrong = i;
       break;
    }
  }

/*AAA
  cout << "card: " << matcard << " no of wkststr " << weakeststrong
       << " w= " << wbox[ordsc[weakeststrong]]
       << " ucn0 " << unconnode[0] << " w= " << wbox[ordsc[unconnode[0]]]
       << " ucn1 " << unconnode[1] << " w= " << wbox[ordsc[unconnode[1]]]
       << endl;
  cout << "matching: ";
  for (i=0; i<numsc; ++i)
  {
    if (matlist[i] != empty)
      cout << " w1= " << wbox[ordsc[i]]
           << " w2= " << wbox[ordsc[matlist[i]]] << " -- ";
  }
*/
/*
   if (matlist[numsc] == empty)
   {
     cout << "unconnode[0] " << unconnode[0]
          << "  unconnode[1]  " << unconnode[1]
          << "  matcard   " << matcard << endl;
   }
*/

  if (wbox[ordsc[unconnode[1]]]
      + wbox[ordsc[weakeststrong]] >= thradj) // Check sufficiency
  {
    ++suffmat;
    ++feacounter;
    //AAAcout << " yeah suff" << endl;
    if (unconnode[1] < weakeststrong)
```

```
      ++yeahcounter;  //cout << " YEAH!!!" << endl;
    continue;
  }
  //AAAcout << " not suff" << endl;

// ***************************************************************************
// Building up chain
// ***************************************************************************

// Step 1: Initialise data
  for(i=0; i<numsc; ++i)
  {
    analysed[i] = 0;
    for(j=0; j<nocomp; ++j)
      component[j] [i] = empty;
    if (ordsc[i] % 2 == 0) // find out twin node number
      twin[i] = invordsc[(ordsc[i]+1)];
    else
      twin[i] = invordsc[(ordsc[i]-1)];
    //cout << ordsc[i] << "-" << ordsc[twin[i]] << "  ";
  }
  //cout << endl;

// Step 2: Build up chain
  for(i=0; i<numsc; ++i)     // find smallest unconnected node
  {
    if (matlist[i] == empty)
    {
      smallestuncon = i;
      break;
    }
  }
  if (smallestuncon != unconnode[0])
    cout << "ALARM!!!" << endl;

  j = -1;                    // build up chain
  stack = smallestuncon;
  do
  {
    ++j;
    component[0] [j] = stack;
    analysed[stack] = 1;
    ++j;
    component[0] [j] = twin[stack];
    analysed[twin[stack]] = 1;
    stack = matlist[twin[stack]];
  }
  while (stack != empty);

  lengthofcomponent[0] = ++j;

// ***************************************************************************
// Case (07): Chain complete with length = numsc
// ***************************************************************************
  if (lengthofcomponent[0] == numsc)// is chain complete?
  {
    ++completechain;
    ++feacounter;
    nocycles = 0;
    for(i=0; i<numsc; ++i)
      result[i] = component[0] [i];
    i = checkresult(7);
/*
    for(i=0; i<numbox; ++i)
    {
      cout << wbox[ordsc[component[0] [(2*i)]]] << "("
```

```
            << ordsc[component[0] [(2*i)]] << ")-("
            << ordsc[component[0] [(2*i+1)]] << ")"
            << wbox[ordsc[component[0] [(2*i+1)]]] << " -- ";
    }
    cout << "*" << endl;
*/
    continue;
  }

// ***************************************************************************
// Building up cycles
// ***************************************************************************
  for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
    {
    if (analysed[i] == 0)
      {
        smallestconnotana = i;
        break;
      }
    }
  if (matlist[smallestconnotana] == empty)
       cout << "ARLARM2!!!" << endl;

  currentcomponent = 0;
  do
  {
     ++currentcomponent;      // Set component
     if (currentcomponent > nocomp-1)
       cout << "ARLARM3!!!" << endl;

     j = -1;                  // Build up cycle
     stack = smallestconnotana;
     do
     {
       ++j;
       component[currentcomponent] [j] = stack;
       analysed[stack] = 1;
       ++j;
       component[currentcomponent] [j] = twin[stack];
       analysed[twin[stack]] = 1;
       stack = matlist[twin[stack]];
     }
     while (stack != smallestconnotana); // = while not back 2 beginning of cyc

     lengthofcomponent[currentcomponent] = ++j;

     for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
       if (analysed[i] == 0)
       {
          smallestconnotana = i;
          break;
       };
     if (matlist[smallestconnotana] == empty)
       cout << "ARLARM2B!!!" << endl;
  }
  while (smallestconnotana != stack); // = while new not analysed node found
  nocycles = currentcomponent;        // save number of cycles

// ***************************************************************************
// Analysing cycles and chain
// ***************************************************************************
  for (i=0; i<nocomp; ++i)  // Initialising data
    {
      strongestnode[i] = empty;
      pss[i] = empty;
    }
```

```
  for (i=1; i<=nocycles; ++i)  // Check all cycles
  {
    // initialise for this component
    strongestnode[i] = component[i] [0];
    pss[i] = 0;

    // check for all elements of this component
    for (j=0; j<lengthofcomponent[i]; ++j) // Find weakest & strongest in cycle
    {
        // > cases
      if (wbox[ordsc[component[i] [j]]] > wbox[ordsc[strongestnode[i]]])
        {
          strongestnode[i] = component[i] [j];
          pss[i] = j;
        }

        // == cases
      if ((wbox[ordsc[component[i] [j]]] == wbox[ordsc[strongestnode[i]]])
       && (wbox[ordsc[matlist[component[i] [j]]]]
           > wbox[ordsc[matlist[strongestnode[i]]]]))
        {
          strongestnode[i] = component[i] [j];
          pss[i] = j;
        }
    }
  }

    // initialise charcateristics for all cycles
  weakeststrongest = strongestnode[1];
  pwss = pss[1];

    // check for all cycles
  for(i=1; i<=nocycles; ++i)
  {
    // < cases
    if (wbox[ordsc[strongestnode[i]]] < wbox[ordsc[weakeststrongest]])
      {
        weakeststrongest = strongestnode[i];
        pwss = pss[i];
      }

    // == cases
    if ((wbox[ordsc[strongestnode[i]]]
         == wbox[ordsc[weakeststrongest]])
     && (wbox[ordsc[matlist[strongestnode[i]]]]
          < wbox[ordsc[matlist[weakeststrongest]]]))
      {
        weakeststrongest = strongestnode[i];
        pwss = pss[i];
      }

  }

  // Counting number of cycles b4 analysis
  ++distofcyclesstart[nocycles];

// ***************************************************************************
// CASE 08: Connection of cycles with chain via weakest strongest strong node
// ***************************************************************************
// Step 1: Make sure that unconnode[1] really is the higher unconnected node
  if (wbox[ordsc[unconnode[0]]] > wbox[ordsc[unconnode[1]]])
    cout << "ALARM4!!!" << endl;

// Step 2: Check w/ wkst strgst strong
  if (wbox[ordsc[weakeststrongest]]
```

```
    + wbox[ordsc[unconnode[1]]] >= thradj)
{
++wscyclesandchain;
++feacounter;
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Step 2ba: Checking result for WSS node case if there is only one cycle
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 if (nocycles == 1)
 {
   /*
   for(i=0; i<lengthofcomponent[1]; ++i)//check where in cycle strngst node
   {
     if ((component[1][i])==strongestnode[1])
     {
       stack = i;
       break;
     }
   }
   */
   stack = pss[1];

   if (twin[strongestnode[1]]==component[1] [(stack+1)])//twin after str nd
   {
     for(i=0; i<lengthofcomponent[0]; ++i)
       result[i] = component[0] [i];
     for(i=stack; i<lengthofcomponent[1]; ++i)
       result[(i+lengthofcomponent[0]-stack)] = component[1] [i];
     if (stack != 0)                          // str nd is not first node
     {
       for(i=0; i<stack; ++i)
         result[(i+lengthofcomponent[0]+lengthofcomponent[1]-stack)]
         = component[1] [i];
     }
   }
   else   // twin before strongest node
   {
     for(i=0; i<lengthofcomponent[0]; ++i)
       result[i] = component[0] [i];
     for(i=stack; i>=0; --i)
       result[(lengthofcomponent[0]+stack-i)] = component[1] [i];
     if (stack != (lengthofcomponent[1]-1))  // str nd is not last node
     {
       for(i=(lengthofcomponent[1]-1); i>stack; --i)
         result[(lengthofcomponent[0]+stack+(lengthofcomponent[1]-i))]
         = component[1] [i];
     }
   }
 }
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Step 2bb: Checking result for WSS case if there is more than one cycle
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

 else                                          // There is more than one cycle
 {
   // Initialising order
   for(i = 1; i <= nocycles; ++i)
   {
      ordcyc[i] = i;
   }
   // Starting sorting procedure for cycles according to strngest strong nodes
   for(i = 1; i <= nocycles; ++i)
   {
      for(j = i-1; j >= 1; --j)
      {
       //cout << i << "  " << ordsc[i] << "  "
```

```
    //     << j << "   " << ordsc[j] << endl;
    // abbreviation = ;
     if (wbox[ordsc[strongestnode[i]]]
         < wbox[ordsc[strongestnode[ordcyc[j]]]])
     {
       ordcyc[j+1] = ordcyc[j];
       ordcyc[j] = i;
         //cout << "after change  " << i << "  " << ordsc[i] << "  "
         //     << j << "   " << ordsc[j] << endl;
     }
       //else { cout << "no change, next i" << endl; break; }
   }
}
// Looking for the place of strongest node in each cycle
// because this tells us where to start building the result out of cycles
/*
for(i = 1; i <= nocycles; ++i)
{
  placeofstrongestnode[i] = empty;
}
for(i = 1; i <= nocycles; ++i)
{
  for(j = 0; j < lengthofcomponent[i]; ++j)
  {
    if (strongestnode[i] == component[i][j])
    {
      placeofstrongestnode[i] = j;
      break;
    }
  }
}
*/
// Starting result array with chain
for(i=0; i<lengthofcomponent[0]; ++i)
    result[i] = component[0] [i];
currentplace = lengthofcomponent[0];
// Now add the cycles to the result
for(k = 1; k <= nocycles; ++k)
{
  // first option: twin comes after strongest node in this cycle
  if (twin[strongestnode[ordcyc[k]]]
      == component[ordcyc[k]] [(pss[ordcyc[k]]+1)])
  {
     for(i = pss[ordcyc[k]];
         i < lengthofcomponent[ordcyc[k]]; ++i)
       result[(  i
               + currentplace
               - pss[ordcyc[k]])]
       = component[ordcyc[k]] [i];
     if (pss[ordcyc[k]] != 0) // str nd is not first node
     {
       for(i=0; i<pss[ordcyc[k]]; ++i)
         result[(  i
                 + currentplace
                 + lengthofcomponent[ordcyc[k]]
                 - pss[ordcyc[k]])]
         = component[ordcyc[k]] [i];
     }
   }
   // second option: twin comes b4 strongest node in this cycle
   else
   {
     for(i=pss[ordcyc[k]]; i>=0; --i)
       result[(  currentplace
               + pss[ordcyc[k]]
               - i)]
```

```
          = component[ordcyc[k]] [i];
        if (pss[ordcyc[k]]
            != (lengthofcomponent[ordcyc[k]]-1))  // str nd is not last node
        {
           for(i = (lengthofcomponent[ordcyc[k]]-1);
               i > pss[ordcyc[k]]; --i)
            result[(  currentplace
                    + pss[ordcyc[k]]
                    + (lengthofcomponent[ordcyc[k]]-i))]
            = component[ordcyc[k]] [i];
        }
      }
      // store new position in building result array
      currentplace = currentplace + lengthofcomponent[ordcyc[k]];
    } // end of loop through all cycles
   ++numberofelsecases;
   } // end of else for case of more than one cycle

   status = checkresult(8);
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  continue;
  }

// ************************************************************************
// Statistics and some more checks for remaining cases
// ************************************************************************
  // Counting the length of the chain and no of cycles in remaining cases
  ++distofcyclesleft[nocycles];
  ++lengthchain[lengthofcomponent[0]];


  // Check in the case of one cycle
  if (nocycles == 1)
  {
    stack = 0;
    for(i=0; i<lengthofcomponent[1]; ++i)
    {
      if (wbox[ordsc[component[1] [i]]] > stack)
        stack = wbox[ordsc[component[1] [i]]];
    }
    if (stack + wbox[ordsc[unconnode[1]]] >= thradj)
      ++prob;
    //if (instance < 1000)
      //cout << "T";
  }

// ************************************************************************
// Inverse matching algorithm (TGMAmax)
// ************************************************************************
// Step 1 Inverse: Initialise matching list and counters
   for (i = 0; i < numsc; ++i)
   {
     matlistinv[i] = empty;
   }
   matcardinv = 0;
   unconpointerinv = 0;
   for (i = 0; i < numsc; ++i)
     {
       unconnodeinv[i] = empty;
     }
   lastmatchinv = empty;

// Step 2 Inverse: Matching algorithm
   for(i=0; i<numsc; ++i)// check all nodes
   {
     if (matlistinv[i] == empty)// does node need a mate?
```

```
    {
      for (j=(numsc-1); j>i; --j)// look for a mate for node i
      {
        if (  (adjlist[i] [j] == 1)
           && (matlistinv[j] == empty))// if mate found
        {
          matlistinv[i] = j;
          matlistinv[j] = i;
          lastmatchinv = i;
          ++matcardinv;
          break;
        }
      }
      if (matlistinv[i] == empty)   // if there still is no mate:
      {                           // do twin node swap or finally acquiesce
        if (ordsc[i] % 2 == 0)     // find out twin node number
        {
          twinnomatinv = invordsc[(ordsc[i]+1)];
        }
        else
        {
          twinnomatinv = invordsc[(ordsc[i]-1)];
        }
        if                          // twin node swap possible?
        (  (wbox[ordsc[i]]+wbox[ordsc[twinnomatinv]]>=thradj)// match w/ twin?
        && (matlistinv[twinnomatinv] == empty)               // twin unmatchd?
        && (lastmatchinv != empty)                           // exchnge pssble?
        && (twinnomatinv > i)                               // twin larger?
        && (  wbox[ordsc[lastmatchinv]]                 // lastmatch with twin?
            + wbox[ordsc[twinnomatinv]] >= thradj))
        {                           // then swap mates
          matlistinv[i] = matlistinv[lastmatchinv];
          matlistinv[lastmatchinv] = twinnomatinv;
          matlistinv[twinnomatinv] = lastmatchinv;
          matlistinv[matlistinv[i]] = i;
          lastmatchinv = i;
          ++matcardinv;
        }
        else                        // otherwise: one more unconnected node
        {
          ++unconpointerinv;
          unconnodeinv[(unconpointerinv-1)] = i;
        }
      }
    }
  }

// ****************************************************************************
// Double check inverse matching with other matching algorithm
// ****************************************************************************
  if (matcard != matcardinv)
  {
    cout << "ALARM5!!!" << endl;
    cout << matcard << "  " << matcardinv << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << wbox[i];
    cout << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << wbox[ordsc[i]];
    cout << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << matlist[i];
    cout << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << matlistinv[i];
    cout << endl;
```

```
  cout << adjlist[2] [matlist[2]] << endl;
  cout << wbox[ordsc[2]] << "  " << wbox[ordsc[matlist[2]]] << endl;
  cout << unconpointer << "  " << unconpointerinv << endl;

  //First matching algorithm once again (for checking mistakes
  // Step 2 once again: Initialise matching list and counters
for (i = 0; i < numsc; ++i)
{
  matlist[i] = empty;
}
matcard = 0;
unconpointer = 0;
for (i = 0; i < numsc; ++i)
  {
    unconnode[i] = empty;
  }
lastmatch = empty;

 // Step 3 once again: Matching algorithm
for(i=0; i<numsc; ++i)// check all nodes
{
  if (matlist[i] == empty)// does node need a mate?
  {
    for (j=(i+1); j<numsc; ++j)// look for a mate for node i
    {
      if (  (adjlist[i] [j] == 1)
         && (matlist[j] == empty))// if mate found
      {
        matlist[i] = j;
        matlist[j] = i;
        lastmatch = i;
        ++matcard;
        break;
      }
    }
    if (matlist[i] == empty)   // if there still is no mate:
    {                          // do twin node swap or finally acquiesce
      if (ordsc[i] % 2 == 0)     // find out twin node number
      {
        twinnomat = invordsc[(ordsc[i]+1)];
      }
      else
      {
        twinnomat = invordsc[(ordsc[i]-1)];
      }
      if                           // twin node swap possible?
      (  (wbox[ordsc[i]]+wbox[ordsc[twinnomat]]>=thradj)// match with twin?
      && (matlist[twinnomat] == empty)                  // twin unmatched?
      && (lastmatch != empty)                           // exchange possble?
      && (twinnomat > i))                               // twin larger?
      {                            // then swap mates
        matlist[i] = matlist[lastmatch];
        matlist[lastmatch] = twinnomat;
        matlist[twinnomat] = lastmatch;
        matlist[matlist[i]] = i;
        lastmatch = i;
        ++matcard;

        cout << "a twin node swap" << endl;
        cout << " i " << i << " twinnomat " << twinnomat
             << " ordsc[i] " << ordsc[i] << " ordsc[twinnomat] "
                                         << ordsc[twinnomat]
             << " wboxSumme " << wbox[ordsc[i]]+wbox[ordsc[twinnomat]]
             << " matlist[twinnomat] " << matlist[twinnomat]
             << " lastmatch " << lastmatch
             << " matlist[lastmatch] " << matlist[lastmatch] << endl;
```

```
            //matlist[i] = matlist[lastmatch];
            //matlist[lastmatch] = twinnomat;
            //matlist[twinnomat] = lastmatch;
            //matlist[matlist[i]] = i;
            //lastmatch = i;
            //++matcard;

          }
          else                        // otherwise: one more unconnected node
          {
            ++unconpointer;
            unconnode[(unconpointer-1)] = i;
            cout <<   " an unconnode " << i << endl;
          }
        }
      }
    }
    cout << "new results" << endl;
    cout << adjlist[2] [matlist[2]] << endl;
    cout << wbox[ordsc[2]] << "  " << wbox[ordsc[matlist[2]]] << endl;
    cout << unconpointer << "  " << unconpointerinv << endl;

// End if matcard != matcardinv
    }

// ***************************************************************************
// Building up chain from inverse matching
// ***************************************************************************

// Step 1INV: Initialise data
  for(i=0; i<numsc; ++i)
  {
    analysedinv[i] = 0;
    for(j=0; j<nocomp; ++j)
      componentinv[j] [i] = empty;
    if (ordsc[i] % 2 == 0) // find out twin node number
      twin[i] = invordsc[(ordsc[i]+1)];
    else
      twin[i] = invordsc[(ordsc[i]-1)];
    //cout << ordsc[i] << "-" << ordsc[twin[i]] << "  ";
  }
  //cout << endl;

// Step 2INV: Build up chain
  for(i=0; i<numsc; ++i)     // find smallest unconnected node
  {
    if (matlistinv[i] == empty)
    {
      smallestunconinv = i;
      break;
    }
  }
  if (smallestunconinv != unconnodeinv[0])
    cout << "ALARMinv!!!" << endl;

  j = -1;                     // build up chain
  stack = smallestunconinv;
  do
  {
    ++j;
    componentinv[0] [j] = stack;
    analysedinv[stack] = 1;
    ++j;
    componentinv[0] [j] = twin[stack];
    analysedinv[twin[stack]] = 1;
```

```
    stack = matlistinv[twin[stack]];
  }
  while (stack != empty);

  lengthofcomponentinv[0] = ++j;

// ********************************************************************************
// CASE (10): Chain from inverse matching complete with length = numsc
// ********************************************************************************
  if (lengthofcomponentinv[0] == numsc)// is chain complete?
  {
    ++completechaininv;
    ++feacounter;
    for(i=0; i<numsc; ++i)
      result[i] = componentinv[0] [i];
    i = checkresult(10);

/*
    for(i=0; i<numbox; ++i)
    {
      cout << wbox[ordsc[component[0] [(2*i)]]] << "("
           << ordsc[component[0] [(2*i)]] << ")-("
           << ordsc[component[0] [(2*i+1)]] << ")"
           << wbox[ordsc[component[0] [(2*i+1)]]] << " -- ";
    }
    cout << "*" << endl;
*/

    continue;
  }

// ********************************************************************************
// Building up cycles from inverse matching
// ********************************************************************************
  for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
    {
    if (analysedinv[i] == 0)
      {
        smallestconnotanainv = i;
        break;
      }
    }
  if (matlistinv[smallestconnotanainv] == empty)
       cout << "ARLARM2inv!!!" << endl;

  currentcomponent = 0;
  do
  {
     ++currentcomponent;      // Set component
     if (currentcomponent > nocomp-1)
       cout << "ARLARM3inv!!!" << endl;

     j = -1;                  // Build up cycle
     stack = smallestconnotanainv;
     do
     {
       ++j;
       componentinv[currentcomponent] [j] = stack;
       analysedinv[stack] = 1;
       ++j;
       componentinv[currentcomponent] [j] = twin[stack];
       analysedinv[twin[stack]] = 1;
       stack = matlistinv[twin[stack]];
     }
     while (stack != smallestconnotanainv); //= while not back 2 bgnning of cyc
```

```
     lengthofcomponentinv[currentcomponent] = ++j;

     for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
       if (analysedinv[i] == 0)
       {
          smallestconnotanainv = i;
          break;
       };
     if (matlistinv[smallestconnotanainv] == empty)
       cout << "ARLARM2Binv!!!" << endl;
   }
   while (smallestconnotanainv != stack); // = while new not analysed node found
   nocyclesinv = currentcomponent;        // save number of cycles

// ***************************************************************************
// Analysing cycles and chain from inverse matching
// ***************************************************************************
   for (i=0; i<nocomp; ++i)  // Initialising data
   {
     strongestnodeinv[i] = empty;
     pssinv[i] = empty;
   }

   for (i=1; i<=nocyclesinv; ++i)  // Check all cycles
   {

     // INV: Initialise for all cycles
     strongestnodeinv[i] = componentinv[i] [0];
     pssinv[i] = 0;

     // INV: Check for all elements of this component
     for (j=1; j<lengthofcomponentinv[i]; ++j) // Find weakst&strongst in cycle
     {

        // > cases
     if (wbox[ordsc[componentinv[i] [j]]]
         >= wbox[ordsc[strongestnodeinv[i]]])
       {
         strongestnodeinv[i] = componentinv[i] [j];
         pssinv[i] = j;
       }

     // == cases
     if ((wbox[ordsc[componentinv[i] [j]]]
          == wbox[ordsc[strongestnodeinv[i]]])
      && (wbox[ordsc[matlistinv[componentinv[i] [j]]]]
           > wbox[ordsc[matlistinv[strongestnodeinv[i]]]]))
       {
         strongestnodeinv[i] = componentinv[i] [j];
         pssinv[i] = j;
       }
     }
   }

     // INV :Initialise charcteristics for all cycles
   weakeststrongestinv = strongestnodeinv[1];
   pwssinv = pssinv[1];

     // INV: Check for all cycles
   for(i=1; i<=nocyclesinv; ++i)
   {
     // < cases
     if (wbox[ordsc[strongestnodeinv[i]]] < wbox[ordsc[weakeststrongestinv]])
       {
         weakeststrongestinv = strongestnodeinv[i];
         pwssinv = pssinv[i];
```

```
      }

     // == cases
     if ((wbox[ordsc[strongestnodeinv[i]]]
          == wbox[ordsc[weakeststrongestinv]])
      && (wbox[ordsc[matlistinv[strongestnodeinv[i]]]]
          < wbox[ordsc[matlistinv[weakeststrongestinv]]]))
      {
        weakeststrongestinv = strongestnodeinv[i];
        pwssinv = pssinv[i];
      }
  }

  // Counting number of cycles b4 analysis
  ++distofcyclesstartinv[nocyclesinv];

// ****************************************************************************
// CASE (11): Structure-preserving solution in INV/TGMAmaax case
// ****************************************************************************

// Step 1INV: Make sure that unconnodeinv[1] is the higher unconnected node
  if (wbox[ordsc[unconnodeinv[0]]] > wbox[ordsc[unconnodeinv[1]]])
    cout << "ALARM4inv!!!" << endl;

// Step 2INV: Check w/ wss
  if (wbox[ordsc[weakeststrongestinv]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++wscyclesandchaininv;
  ++feacounter;

  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // Step 2baINV: Checking result for WSS node case if there is only one cycle
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   if (nocyclesinv == 1)
   {
     /*
     for(i=0; i<lengthofcomponentinv[1]; ++i)//check where in cycle strngst node
     {
       if ((componentinv[1][i])==strongestnodeinv[1])
       {
         stackinv = i;
         break;
       }
     }
     */
     stackinv = pssinv[1];

     if (twin[strongestnodeinv[1]]==componentinv[1] [(stackinv+1)])
                                                        //twin after str nd
     {
       for(i=0; i<lengthofcomponentinv[0]; ++i)
         result[i] = componentinv[0] [i];
       for(i=stackinv; i<lengthofcomponentinv[1]; ++i)
         result[(i+lengthofcomponentinv[0]-stackinv)] = componentinv[1] [i];
       if (stackinv != 0)                          // str nd is not first node
       {
         for(i=0; i<stackinv; ++i)
           result[(i+lengthofcomponentinv[0]+lengthofcomponentinv[1]-stackinv)]
           = componentinv[1] [i];
       }
     }
     else   // twin before strongest node
     {
       for(i=0; i<lengthofcomponentinv[0]; ++i)
```

```
      result[i] = componentinv[0] [i];
    for(i=stackinv; i>=0; --i)
      result[(lengthofcomponentinv[0]+stackinv-i)] = componentinv[1] [i];
    if (stackinv != (lengthofcomponentinv[1]-1))  // str nd is not last node
    {
      for(i=(lengthofcomponentinv[1]-1); i>stackinv; --i)
        result[(lengthofcomponentinv[0]
        + stackinv
        + (lengthofcomponentinv[1]-i))]
        = componentinv[1] [i];
    }
  }
}
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Step 2bbINV: Checking result for WSS case if there is more than one cycle
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

else                                          // There is more than one cycle
{
  // Initialising order
  for(i = 1; i <= nocyclesinv; ++i)
  {
     ordcycinv[i] = i;
  }
  // Starting sorting procedure for cycles according to stronges strong nodes
  for(i = 1; i <= nocyclesinv; ++i)
  {
     for(j = i-1; j >= 1; --j)
     {
      //cout << i << "  " << ordsc[i] << "  "
      //     << j << "   " << ordsc[j] << endl;
      // abbreviation = ;
       if (wbox[ordsc[strongestnodeinv[i]]]
           < wbox[ordsc[strongestnodeinv[ordcycinv[j]]]])
       {
         ordcycinv[j+1] = ordcycinv[j];
         ordcycinv[j] = i;
           //cout << "after change  " << i << "  " << ordsc[i] << "  "
           //     << j << "   " << ordsc[j] << endl;
       }
         //else { cout << "no change, next i" << endl; break; }
     }
  }
  // Looking for the place of strongest node in each cycle
  // because this tells us where to start building the result out of cycles
  /*
  for(i = 1; i <= nocyclesinv; ++i)
  {
    placeofstrongestnodeinv[i] = empty;
  }
  for(i = 1; i <= nocyclesinv; ++i)
  {
    for(j = 0; j < lengthofcomponentinv[i]; ++j)
    {
      if (strongestnodeinv[i] == componentinv[i][j])
      {
        placeofstrongestnodeinv[i] = j;
        break;
      }
    }
  }
  */

  // Starting result array with chain
  for(i=0; i<lengthofcomponentinv[0]; ++i)
      result[i] = componentinv[0] [i];
```

```
    currentplaceinv = lengthofcomponentinv[0];
    // Now add the cycles to the result
    for(k = 1; k <= nocyclesinv; ++k)
    {
      // first option: twin comes after strongest node in this cycle
      if (twin[strongestnodeinv[ordcycinv[k]]]
          == componentinv[ordcycinv[k]]
                         [(pssinv[ordcycinv[k]]+1)])
      {
         for(i = pssinv[ordcycinv[k]];
             i < lengthofcomponentinv[ordcycinv[k]]; ++i)
           result[(  i
                   + currentplaceinv
                   - pssinv[ordcycinv[k]])]
           = componentinv[ordcycinv[k]] [i];
         if (pssinv[ordcycinv[k]] != 0)
                                                 // str nd is not first node
         {
           for(i=0; i<pssinv[ordcycinv[k]]; ++i)
             result[(  i
                     + currentplaceinv
                     + lengthofcomponentinv[ordcycinv[k]]
                     - pssinv[ordcycinv[k]])]
             = componentinv[ordcycinv[k]] [i];
         }
       }
       // second option: twin comes b4 strongest node in this cycle
       else
       {
         for(i=pssinv[ordcycinv[k]]; i>=0; --i)
           result[(  currentplaceinv
                   + pssinv[ordcycinv[k]]
                   - i)]
           = componentinv[ordcycinv[k]] [i];
         if (pssinv[ordcycinv[k]]
             != (lengthofcomponentinv[ordcycinv[k]]-1))
                                                  // str nd is not last node
         {
            for(i = (lengthofcomponentinv[ordcycinv[k]]-1);
                i > pssinv[ordcycinv[k]]; --i)
              result[(  currentplaceinv
                      + pssinv[ordcycinv[k]]
                      + (lengthofcomponentinv[ordcycinv[k]]-i))]
              = componentinv[ordcycinv[k]] [i];
         }
       }
       // store new position in building result array
       currentplaceinv = currentplaceinv + lengthofcomponentinv[ordcycinv[k]];
     } // end of loop through all cycles
   ++numberofelsecasesinv;
   } // end of else for case of more than one cycle

   status = checkresult(11);
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


  continue;
  }

// ***************************************************************************
// Statistics and some more checks for remaining cases after inverse match
// ***************************************************************************
  // Counting the number of cycles and the length of the chain
  ++distofcyclesleftinv[nocyclesinv];
  ++lengthchaininv[lengthofcomponentinv[0]];
```

```
  // Check in the case of one cycle
  if (nocyclesinv == 1)
  {
    stack = 0;
    for(i=0; i<lengthofcomponentinv[1]; ++i)
    {
      if (wbox[ordsc[componentinv[1] [i]]] > stack)
        stack = wbox[ordsc[componentinv[1] [i]]];
    }
    if (stack + wbox[ordsc[unconnodeinv[1]]] >= thradj)
      ++probinv;
    //if (instance < 1000)
      //cout << "T";
  }

// ***************************************************************************
// End of instances loop
// ***************************************************************************
/*
// Output scores, ordered scores and order numbers
  for(i = 1; i <= numsc; ++i)
  {
    cout << wbox[i] << "  ";
  }
  cout << endl;
  for(i = 1; i <= numsc; ++i)
  {
    cout << wbox[ordsc[i]] << "  ";
  }
  cout << endl;
  for(i = 1; i <= numsc; ++i)
  {
    cout << ordsc[i] << "  ";
  }
  cout << endl;
  cin.get();
*/

// End of instances loop
}

// Running time
   runtime1 = (double) (clock() / CLOCKS_PER_SEC);
   // runtime2 = (double) (clock() - (numinst*time4rand)) / CLOCKS_PER_SEC;

// ***************************************************************************
// Output of final statisctis
// ***************************************************************************
   // Main statistics
   cout << "Instances: " << numinst << "  Fea: " << feacounter
                                    << "  Inf: " << infcounter << endl;
   cout << "Instances" << endl
        << "- (01)  with too many weak nodes: " << toomanyweak << endl
        << "- (02)  with non-con twins:  " << noncontwin << endl
        << "- (03)  with too many unconnectable nodes:  " << uncon << endl
        << "- (04)  with poor matching: " << poormat << endl
        << "- (05)  with perfect matching: " << perfmat
        << ", CHECK: " << checkcasecounter[5] << endl
     // << "- (06)  with sufficient matching: " << suffmat
     // << ", CHECK: " << checkcasecounter[6]
           << endl
        << "- (07)  with TGMAmin complete chain built: " << completechain
    //  << ", CHECK: " << checkcasecounter[7]
           << endl
        << "- (08)  with TGMAmin structure-preserving solution: "
                                              << (wscyclesandchain + suffmat)
```

```
    //  << ", CHECK: " << checkcasecounter[8]
           << endl;

   cout << "- (10)  with TGMAmax complete chain built: " << completechaininv
   //   << ", CHECK : " << checkcasecounter[10]
        << endl
        << "- (11)  with TGMAmax structure-preserving solution: "
                                          << wscyclesandchaininv
   //   << ", CHECK: " << checkcasecounter[11]
        << endl;

   cout << "Percentage of instances solved: "
        << (double) (feacounter+infcounter)/numinst << endl;
   cout << "Running time: " << runtime1 << " seconds" << endl;
   cout << "Number of instances checked: " << resultcounter
        << " Failed checks among these: " << problemcounter;
   cout << endl << endl << endl;
/*
   // Other statistics
   cout << " Number of elsecases: " << numberofelsecases;
   cout << " Number of elsecasesINV: " << numberofelsecasesinv;
   cout << endl << endl;

   for(i=0; i<=numsc; ++i)
     cout << lengthchain[i] << " times " << i << " scores" << endl;
   for(i=0; i<=numsc; ++i)
   {
     cout << lengthchaininv[i] << " times "
          << i << " scores in INV case" << endl;
   }
   // cout << "Running time without generation of instances: "
   //      << runtime2 << " seconds" << endl;
   for(i=1; i<nocomp; ++i)
   {
     cout << distofcyclesstart[i] << " times " << i
          << " cycles originally, afterwards "
          << distofcyclesleft[i] << " times." << endl;

   }
   for(i=1; i<nocomp; ++i)
   {
     cout << distofcyclesstartinv[i] << " times " << i
          << " cycles originally, afterwards "
          << distofcyclesleftinv[i] << " times in INV case." << endl;

   }

   cout << prob << " problematic cases" << endl;
   cout << probinv << " problematic cases in INV case" << endl;
*/

// ****************************************************************************
// End of function main
// ****************************************************************************
  cin.get();
  return 0;
}

// ****************************************************************************
// ************************  END OF PROGRAMME *********************************
// ****************************************************************************
```

B TGHRA 3.6: C++ source code

```
// ************************************************************************
// ------------------------------------------------------------------------
// ************************   TGHRA 3.6   ********************************
// ------------------------------------------------------------------------
// ************************************************************************

// 07 April 2010, Kai Helge Becker

// ************************************************************************
// Header files, constants, global variables
// ************************************************************************

#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

// Global variables and constants
// Constants
   int const numinst = 1000000;      // number of instances
   int const numbox = 22;            // number of boxes + 1
   int const numsc  = 2 * numbox;    // number of scores + 2
   int const minwidth = 41;          // minimal width of each score
   int const maxwidth = 70;          // maximal width of each score
   int const thradj = 70;            // threshold for adjacency
   int const thrstrong = thradj / 2; // threshold for strong node (= thradj/2)
   int const empty = 999;            // flag for empty variable entry
                                     // (used w/ matlist, unconnode, lastmatch)
   int const nocomp
           = (numbox+(numbox%2))/2;  // no of components (no of cycles)
   int const b
             = maxwidth - minwidth + 1; // no of possible score widths
   double const b2 = b;

// Variables
   int sto0;
   double stochastic;                // random number between 0 and 1
                                     // (for score width)
   double probability[b];            // triangular probability
   double endofpartition[b];         // for calculating triangular distribution
   int wbox[numsc];                  // width of all scores
   int i, j, k, q, qstar;            // loops
   int stack;                        // stack
   int stackinv;
   int ordsc[numsc];                 // array index of ith smallest score
   int invordsc[numsc];              // inverse function of the above
   int instance;                     // counters for instances and cases
   int feacounter;
   int infcounter;
   int poormat;
   int completecycle;
   int nofamily;
   int patgraphcon;
   int patgraphuncon;

   int twinno1b;                  // place of an unconnectable box (case 1b)
   double runtime1;               // running time
   //double runtime2;             // running time without generating instances
   //long time4rand;              // time for generating one instance
   int adjlist[numsc] [numsc];    // adjacency list (based on sorted indices)
   int emptyflag;                 // flag used in matching algorithm to mark edges
                                  // have not been matched with highest node pssble
                                  // due to twin node conflict. these edges
                                  // will not be used for FCA.
   int cyclenode[numsc];          // during TGMAmax: flag for valid edge (=1) 4 FCA
                                  // (based on sorted indices)
```

```
                                // is "empty" if edge is due to mate swap
                                // contains later: number of the cycle
                                // that node belongs to
   int matlistinv[numsc];       // matching list for inverse matching
   int matcardinv;              // cardinality of inverse matching
   int unconpointerinv;         // number of unconnected nodes
   int twinnomatinv;            // place of twinnode for matching (sorted ind)
   int lastmatchinv;            // place of last matched node (sorted ind)
   int unconnodeinv[numsc];     // place of unconected nodes (sorted ind)
   int smallestunconinv;        // smallest unconnected node (sorted ind)
   int twin[numsc];             // place of twin node (sorted ind)
   int analysedinv[numsc];      // flag if node already inclded in chain/cycle
   int componentinv[nocomp] [numsc];// nodes in cycles [0..nocomp-1]
   int lengthofcomponentinv[nocomp];// length of component
   int lengthfirstcycleinv[(numsc+1)];   // distribution of chain length

   int currentcomponent;        // no of cycle being analysed
   int smallestconnotanainv;    // smallest connected node not analysed yet
   int nocyclesinv;             // number of cycles in INV case

   int currentedge;             // counter used for list of edges
   int noedges;                 // number of (non-empty) edges
   int edge[numbox];            // number of lower node of each (non-empty) edge

   int T [nocomp] [numbox];     // Tq-cycles
   int S [nocomp] [nocomp];     // TIS-edges in Tq-cycle
   int SqIntersectionS;         // ==0 iff (Sq intersection S) == empty set
   int SSet [nocomp];           // TIS-cycles already glued together
   int SSum;                    // Number of TIS-cycles already glued together
   int QSet [nocomp];           // Tq-cycles already used for glueing

   int distofcyclesstartinv[nocomp];// distribution of cycles b4 cycle analysis
   int distofcyclesleftinv[nocomp]; // distribution of cycles after cycle analysis

   int probinv;                     // counter for some problematic cases


// ***************************************************************************
// Function main starts & initialisation
// ***************************************************************************

int main( )
{

// Welcome
   cout << "Welcome to TGHRA 3.6." << endl;


// Initialisation
   feacounter = 0;              // initialising feasible instances counter
   infcounter = 0;              // initialising infeasible instances counter
   poormat = 0;              // initialising case counters
   completecycle = 0;
   nofamily = 0;
   patgraphcon = 0;
   patgraphuncon = 0;

   probinv = 0;

   for(i=0; i<(numsc+1); ++i)
     lengthfirstcycleinv[i] = 0;
   for(i=0; i<nocomp; ++i)
   {
     distofcyclesstartinv[i] = 0;
     distofcyclesleftinv[i] = 0;
   }
```

```
// Calculate probabilities for triangular distribution
probability[0] = 2/(b2*b2);
probability[(b-1)] = probability[0];
for (i = 1; i < b/2; ++i)
{
       probability [i] = probability [(i-1)] + 4/(b2*b2);
       probability [(b-1-i)] = probability [i];
}
// Calculate partition of [0,1] interval for triangular distribution
endofpartition[0] = probability[0];
for (i=1; i < b; ++i)
{
       endofpartition [i] = endofpartition [(i-1)] + probability [i];
}

/*
for (i=0; i<b; ++i)
{
       cout << i << "     " << probability[i] << "    " << endofpartition[i] << "
";
}
*/

// Initialising random mumbers
srand( (unsigned) time(NULL) );

// ********************************************************************************
// Start of instances loop
// ********************************************************************************

for (instance = 0; instance < numinst; ++instance)
{

/*
// Producing uniformly distributed random numbers for scores
   //time4rand = clock();
   for(i = 0; i < (numsc - 2); ++i)
   {
      wbox[i] = minwidth + (rand() % (maxwidth - minwidth + 1));
   }
*/
// cout << RAND_MAX << "HHHH   ";

       // Producing triangularly distributed random numbers for scores
   for (i=0; i < (numsc - 2); ++i)
   {
       // Generate random number between 0 and 1
          sto0 = rand();
        stochastic = static_cast<double> (sto0) / 32767;
          // cout << stochastic << "   ";
       // Check the probability interval of which the number is a member
          j = 0;
          while (stochastic > endofpartition[j]) {++j;};
       // Calculate the triangularly distributed number
        wbox[i] = minwidth + j;
   }

/*
// Test data
wbox[0] = 1;
wbox[1] = 2;
wbox[2] = 5;
wbox[3] = 10;
wbox[4] = 6;
wbox[5] = 66;
```

```
wbox[6] = 62;
wbox[7] = 70;
wbox[8] = 20;
wbox[9] = 21;
wbox[10] = 61;
wbox[11] = 51;
wbox[12] = 25;
wbox[13] = 26;
wbox[14] = 27;
wbox[15] = 46;
wbox[16] = 45;
wbox[17] = 47;
*/

// Add two dominating nodes
   wbox[(numsc - 2)] = thradj + 1;
   wbox[(numsc - 1)] = thradj + 1;

// Test data
/*
   wbox[19] = 1; wbox[9] = 30; //wbox[10] = 35; wbox[11] = 36;

   for(i = 10; i < 19; ++i)
   {
     wbox[i] = 20 + (rand() % 6);
   }
   for(i = 12; i < 20; ++i)
   {
      wbox[i] = 10 + (rand() % 11);
   }
*/

/*
// Output instance
   cout << endl;
   for(i = 0; i < numsc; ++i)
   {
      cout << wbox[i] << "   ";
   }
   cout << endl;
*/

   // Running time without generating instances
   // time4rand = clock() - time4rand;
   // cout << time4rand << endl;

// ***************************************************************************
// Sorting scores from smallest to largest
// ***************************************************************************

   // Initialising order
   for(i = 0; i < numsc; ++i)
   {
      ordsc[i] = i;
   }

   // Starting sorting procedure
   for(i = 1; i < numsc; ++i)
   {
      for(j = i-1; j >= 0; --j)
      {
         //cout << i << "  " << ordsc[i] << "  "
         //     << j << "   " << ordsc[j] << endl;
         if (wbox[i] < wbox[ordsc[j]])
         {
            ordsc[j+1] = ordsc[j];
```

```
            ordsc[j] = i;
            //cout << "after change  " << i << "  " << ordsc[i] << "  "
            //     << j << "    " << ordsc[j] << endl;
          }
          //else { cout << "no change, next i" << endl; break; }
       }
   }

/*
// Output sorted instance
   cout << "order: ";
   for(i = 0; i < numsc; ++i)
   {
         cout << wbox[ordsc[i]] << "   ";
   }
   cout << endl;
*/

// ***************************************************************************
// Matching algorithm (TGMAmax)
// ***************************************************************************

// Step 1: List with adjacent nodes disregarding twin nodes
   for(i = 0; i < numsc; ++i)  // general adjacency list
   {
     for(j = 0; j < numsc; ++j)
     {
       if (wbox[ordsc[i]] + wbox[ordsc[j]] >= thradj)
       {
         adjlist[i] [j] = 1;
               adjlist[j] [i] = 1;
       }
       else
       {
         adjlist[i] [j] = 0;
               adjlist[j] [i] = 0;
       }
     }
   }

   for (i=0; i<numsc; ++i)     // generating inverse function of ordsc[]
   {
     invordsc[ordsc[i]] = i;
   }

   // we mark twin nodes qua adjacency list
   for (i = 0; i < numbox; ++i)
   {
     adjlist[invordsc[(2*i)]] [invordsc[(2*i+1)]] = 2;
     adjlist[invordsc[(2*i+1)]] [invordsc[(2*i)]] = 2;
   }


// Step 2: Initialise matching list and counters
   for (i = 0; i < numsc; ++i)
   {
     matlistinv[i] = empty;
        cyclenode[i] = 1; // will be set to "empty" if resulting from mate swap
   }
   matcardinv = 0;
   unconpointerinv = 0;
   for (i = 0; i < numsc; ++i)
     {
       unconnodeinv[i] = empty;
     }
   lastmatchinv = empty;
```

```
// Step 3: Matching algorithm
   for(i=0; i<numsc; ++i)// check all nodes
   {
     emptyflag = 0;
     if (matlistinv[i] == empty)// does node need a mate?
     {
       for (j=(numsc-1); j>i; --j)// look for a mate for node i
       {
         if (  (adjlist[i] [j] >= 1)
            && (matlistinv[j] == empty))// if potential mate found
         {
                     if (adjlist [i] [j] != 2) // if potential mate != twin
                     {
                       matlistinv[i] = j;
                  matlistinv[j] = i;
                  lastmatchinv = i;
                  ++matcardinv;
                            if (emptyflag == 1) // delete edge for FCA if matching
                                                // was not with highest node
                                                          // due to this node
being twin node
                            {
                                   cyclenode[i] = empty;
                                   cyclenode[j] = empty;
                            }
                  break;
                    };
                    if (adjlist [i] [j] == 2) // if potential mate == twin
                    {
                           emptyflag = 1; // mark this case to make sure that the
                                          // matching edge will be left out in FCA
                    }
              }
       }
       if (matlistinv[i] == empty)   // if there still is no mate:
       {                           // do twin node swap or finally acquiesce
         if (ordsc[i] % 2 == 0)     // find out twin node number
         {
           twinnomatinv = invordsc[(ordsc[i]+1)];
         }
         else
         {
           twinnomatinv = invordsc[(ordsc[i]-1)];
         }
         if                         // twin node swap possible?
         (  (wbox[ordsc[i]]+wbox[ordsc[twinnomatinv]]>=thradj)// match w/ twin?
         && (matlistinv[twinnomatinv] == empty)               // twin unmatchd?
         && (lastmatchinv != empty)                           // exchnge pssble?
         && (twinnomatinv > i)                               // twin larger?
         && (  wbox[ordsc[lastmatchinv]]                 // lastmatch with twin?
             + wbox[ordsc[twinnomatinv]] >= thradj))
         {                          // then swap mates
           matlistinv[i] = matlistinv[lastmatchinv];
                matlistinv[lastmatchinv] = twinnomatinv;
           matlistinv[twinnomatinv] = lastmatchinv;
           matlistinv[matlistinv[i]] = i;
                cyclenode[lastmatchinv] = empty; // edge from mate swap will not
count for FCA
                cyclenode[twinnomatinv] = empty;
           lastmatchinv = i;
           ++matcardinv;
         }
         else                       // otherwise: one more unconnected node
         {
           ++unconpointerinv;
```

```
          unconnodeinv[(unconpointerinv-1)] = i;
        }
      }
    }
  }

// ***************************************************************************
// Poor matching (#M <= n-1)
// ***************************************************************************

  if (matcardinv < numbox)
  {
    ++poormat;
    ++infcounter;

/*AAA
    cout << "inf: poor matching with card: " << matcard
         << " and uncon nodes: ";
    for (i=0; i<numsc; ++i)
    {
      if (unconnode[i] != empty)
        cout << unconnode[i] << " w= " << wbox[ordsc[unconnode[i]]]
             << " " << endl;
    }
    cout << endl;
*/
    continue;
  }

// ***************************************************************************
// Building up twin-induced structure of matching from TGMAmax
// ***************************************************************************

   // Initialise data
  for (i=0; i<numsc; ++i)
  {
        analysedinv[i] = 0;
        for (j=0; j<nocomp; ++j)
              componentinv [j] [i] = empty;
        if (ordsc[i] % 2 == 0) // find out twin-node number
              twin[i] = invordsc[(ordsc[i]+1)];
        else
              twin[i] = invordsc[(ordsc[i]-1)];
        // cout << ordsc[i] << "-" << ordsc[twin[i]] << "  ";
  }
  // cout << endl;

  // Find smallest connected node not analysed yet = first node
  for (i=0; i<numsc; ++i)
    {
    if (analysedinv[i] == 0)
      {
        smallestconnotanainv = i;
        break;
      }
    }
  if (matlistinv[smallestconnotanainv] == empty)
      cout << "ARLARM2inv!!!" << endl;

  currentcomponent = -1;  // Implies that first cycle is component 0

  // Build up all components = cycles
  do
  {
    ++currentcomponent;      // Set component
    if (currentcomponent > nocomp-1)
```

```
      cout << "ARLARM3inv!!!" << endl;

    j = -1;                   // Build up cycle
    stack = smallestconnotanainv;
    do
    {
      ++j;
      componentinv[currentcomponent] [j] = stack;
      analysedinv[stack] = 1;
      ++j;
      componentinv[currentcomponent] [j] = twin[stack];
      analysedinv[twin[stack]] = 1;
      stack = matlistinv[twin[stack]];
    }
    while (stack != smallestconnotanainv); //= while not back 2 bgnning of cyc

    lengthofcomponentinv[currentcomponent] = ++j;

    for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
      if (analysedinv[i] == 0)
      {
         smallestconnotanainv = i;
         break;
      };
    if (matlistinv[smallestconnotanainv] == empty)
      cout << "ARLARM2Binv!!!" << endl;
  }
  while (smallestconnotanainv != stack); // = while new not analysed node found
  nocyclesinv = ++currentcomponent;        // save number of cycles

// ****************************************************************************
// Does twin-induced structure consist of only one cycle?
// ****************************************************************************

  if (lengthofcomponentinv[0] == numsc)// is first cycle complete?
  {
    ++completecycle;
    ++feacounter;

/*
    for(i=0; i<numbox; ++i)
    {
      cout << wbox[ordsc[component[0] [(2*i)]]] << "("
           << ordsc[component[0] [(2*i)]] << ")-("
           << ordsc[component[0] [(2*i+1)]] << ")"
           << wbox[ordsc[component[0] [(2*i+1)]]] << " -- ";
    }
    cout << "*" << endl;
*/

    continue;
  }

// Counting number of cycles b4 analysis
  ++distofcyclesstartinv[nocyclesinv];

// ****************************************************************************
// Create for FCA the list cyclenode[i]
// This list contains for each node the cycle that its edge belongs to
// ****************************************************************************

   for (i=0; i < nocyclesinv; ++i)
   {
         for (j=0; j < lengthofcomponentinv[i]; ++j)
         {
               // if edge is not deleted for FCA
```

```
                if (cyclenode[(componentinv[i] [j])] != empty)
                {
                        cyclenode[(componentinv[i] [j])] = i;
                }
        }
    }

// ***************************************************************************
// Create sorted list of edges
// ***************************************************************************

// Problem: For FCA we need an - array edge[matcardinv]
//                               (contains (sorted) number of lower node)
//                             - list that gives cycle for each edge
//                                       - higher and lower node of each edge

// The list of the edges is given by the list of nodes up to matcardinv
// (we just have to remove empty edges)
// We already have for each node the cycle (cyclenode)
// The lower node of an edge is equivalent to the number of the edge
// The higher node of an edge is equivalent to its matching mate

// Create list of edges without empty edges (those generated by mate swap)
   // Initialisation
   currentedge = 0;
   for (i=0; i<matcardinv; ++i)
   {
        edge[i] = empty;
   }
   // Create list
   for (i=0; i<matcardinv; ++i)
   {
        while (cyclenode[i] == empty) {++i;}
        edge[currentedge] = i;
        ++currentedge;
   }
   noedges = currentedge;

// ***************************************************************************
// Family construction algorithm
// ***************************************************************************

   // Initialisation
   qstar = -1;
   for (q=0; q < nocomp; ++q)
   {
        for (i=0; i < matcardinv; ++i)
        {
                T[q] [i] = empty; // set of edges in alternating T-cycle q
        }
        for (i=0; i < nocyclesinv; ++i)
        {
                S[q] [i] = 0; // set of indices of cycles that have edge in T[q]
                              // == 1 iff T-cycle q has an edge
                                      // from twin-induced-structure cycle i
        };
   }
   k = 0; // edge from matching under consideration

   // Start (Take into account: edges k with cyclenode[k]==empty should not be in Tq-
cycle)
   do
   {
        // look for beginning of new Tq-cycle
        while
                ((k < noedges - 2)
```

```
                &&     ((adjlist[edge[k]] [matlistinv[edge[k+1]]] != 1)
                           || (cyclenode[edge[k]] == cyclenode[edge[k+1]])))
        {
                ++k;
        }

        // start new Tq-cycle and add edges to it as long as it is reasonable
        if    ((adjlist[edge[k]] [matlistinv[edge[k+1]]] == 1)
                && (cyclenode[edge[k]] != cyclenode[edge[k+1]]))
        {
                // Assign new number (qstar) to Tq-cycle and add first edge to Tq-
cycle
                ++qstar;
                i = 0;
                T[qstar] [i] = k;
                S[qstar] [cyclenode[edge[k]]] = 1;
                // add more edges to Tq-cycle
         while ((k < noedges - 1)
                        && (adjlist[edge[k]] [matlistinv[edge[k+1]]] == 1)
                        && (S[qstar] [cyclenode[edge[k+1]]] == 0))
                {
                                ++i;
                                ++k;
                                T[qstar] [i] = k;
                                S[qstar] [cyclenode[edge[k]]] = 1;
                }
        }

        ++k;
    }
    while (k < noedges - 1);


// ***************************************************************************
// No potentially appropriate family of alternating T-cycles found
// ***************************************************************************

   if (qstar == -1)
   {
          ++nofamily;
          ++infcounter;
          continue;
   }

// ***************************************************************************
// Check if patching graph connected
// ***************************************************************************

   // Initialisation

   for (q=1; q<=qstar; ++q)
   {
          QSet [q] = 0; // ==1 iff Tq-cycle number q has already been considered
   }
   q = 0; // start with first Tq-cycle
   QSet [0] = 1;

   for (i=0; i<nocyclesinv; ++i)
   {
          SSet [i] = S [q] [i]; // ==1 iff TIS-cycle i has been included
   }

   SSum = 0; // == number of TIS-cycles that have been included
   for (i=0; i<nocyclesinv; ++i)
   {
          SSum = SSum + SSet [i];
```

```
   }

   // Start connectivity check
   while ((q <= qstar) && (SSum < nocyclesinv))
   {
          do // Look for a Tq-cycle that leads to enlargement
          {
                 ++q; // Consider next Tq-cycle
                 SqIntersectionS = empty;
                 if (q <= qstar)
                 {
                     for (j=0; j<nocyclesinv; ++j) // Is there a [j] for which
S[q][j]=1 and SSet[j]=1?
                     {
                        if ((S [q] [j] == 1) && (SSet [j] == 1))
                        {
                               SqIntersectionS = 1;
                            }
                     }
              }
          }
       while ((q < qstar + 1) && ((QSet[q] == 1) || (SqIntersectionS == empty)));

          if (q <= qstar) // if Tq-cycle for enlargement has been found
          {
                 for (i=0; i<nocyclesinv; ++i)
                        {
                               if ((SSet [i] == 0) && (S [q] [i] == 1))
                               {
                                      SSet [i] = 1;
                                      ++SSum;
                               }
                        }
                 QSet [q] = 1;
                 q = 0;
          }
   }

 /*
   // Output for testing
   for (i=0;i<numsc;++i)
   {
          cout << "ordsci" << i << "   " << ordsc[i] << endl;
   }

   for (i=0;i<numsc;++i)
   {
          cout << "matlistinvi" << i << "   " << matlistinv[i] << endl;
   }

   for (q=0;q<nocomp;++q)
   {
          for (i=0;i<numsc;++i)
          {
                 cout << "componentq" << q << "i" << i << "   " << componentinv[q][i]
<< endl;
          }
   }

   for (i=0;i<numsc;++i)
   {
          cout << "cycledgei" << i << "   " << cyclenode[i] << endl;
   }

   for (i=0;i<matcardinv;++i)
   {
```

```
		cout << "edgei" << i << "   " << edge[i] << endl;
	}
	cout << "noedges" << "   " << noedges << endl;

	for (q=0;q<nocomp;++q)
	{
		for (i=0;i<matcardinv;++i)
		{
			cout << "Tq" << q << "i" << i << "   " << T[q][i] << endl;
		}
	}
	for (q=0;q<nocomp;++q)
	{
		for (i=0;i<nocomp;++i)
		{
			cout << "Sq" << q << "i" << i << "   " << S[q][i] << endl;
		}
	}
	for (q=0;q<nocomp;++q)
	{
		cout << "QSet" << q << "   " << QSet[q] << endl;
	}
	for (i=0;i<nocomp;++i)
	{
		cout << "SSet" << i << "   " << SSet[i] << endl;
	}
	cout << "qstar" << qstar << endl;
	cout << "SSum " << SSum << endl;
	cout << "nocycles " << nocyclesinv << endl;
	cout << "numbox " << numbox << endl;
	cout << "nocomp " << nocomp << endl;
	cout << "matcardinv " << matcardinv << endl;
*/

// ***************************************************************************
// If patching graph connected: FEASIBLE, else: INFEASIBLE
// ***************************************************************************

	if (SSum == nocyclesinv)
	{
		++patgraphcon;
		++feacounter;
		continue;
	}

	if (SSum < nocyclesinv)
	{
		++patgraphuncon;
		++infcounter;
		continue;
	}

	if (SSum > nocyclesinv)
	{
		++probinv;
		continue;
	}

// ***************************************************************************
// Statistics and some more checks for remaining cases after inverse match
// ***************************************************************************

	// Counting the number of cycles and the length of the chain
  ++distofcyclesleftinv[nocyclesinv];
  ++lengthfirstcycleinv[lengthofcomponentinv[0]];
```

```
// ***************************************************************************
// End of instances loop
// ***************************************************************************
/*
// Output scores, ordered scores and order numbers
  for(i = 1; i <= numsc; ++i)
  {
    cout << wbox[i] << "  ";
  }
  cout << endl;
  for(i = 1; i <= numsc; ++i)
  {
    cout << wbox[ordsc[i]] << "  ";
  }
  cout << endl;
  for(i = 1; i <= numsc; ++i)
  {
    cout << ordsc[i] << "  ";
  }
  cout << endl;
  cin.get();
*/

// End of instances loop
}

// Running time
   runtime1 = (double) (clock() / CLOCKS_PER_SEC);
   // runtime2 = (double) (clock() - (numinst*time4rand)) / CLOCKS_PER_SEC;

// ***************************************************************************
// Output of final statisctis
// ***************************************************************************

   // Main statistics
   cout << "Instances: " << numinst << "  Fea: " << feacounter
                                    << "  Inf: " << infcounter << endl;
   cout << "Number of boxes: " << (numbox-2) << endl;
   cout << "Distribution uniform in: " << minwidth << " to " << maxwidth << endl;
   cout << "Threshold: " << thradj << endl;
   cout << "Instances" << endl
        << "- (01)  with poor matching: " << poormat << endl
        << "- (02)  with TGMAmax complete cycle built: " << completecycle << endl
             << "- (03)  with no Tq-family: " << nofamily << endl
        << "- (04)  with patching graph unconnected: " << patgraphuncon << endl
        << "- (05)  with patching graph connected: " << patgraphcon << endl;

   cout << "Percentage of instances solved: "
        << (double) 100*(feacounter+infcounter)/numinst << endl;
   cout << "Running time: " << runtime1 << " seconds" << endl;
   cout << endl << endl << endl;

   // Other statistics
   for(i=0; i<=numsc; ++i)
   {
     cout << lengthfirstcycleinv[i] << " times "
          << i << " scores" << endl;
   }
   // cout << "Running time without generation of instances: "
   //      << runtime2 << " seconds" << endl;
   for(i=1; i<nocomp; ++i)
   {
     cout << distofcyclesstartinv[i] << " times " << i
          << " cycles originally, afterwards "
          << distofcyclesleftinv[i] << " times." << endl;
```

```
   }
   cout << probinv << " problematic cases in INV case" << endl;

// ********************************************************************************
// End of function main
// ********************************************************************************
  cin.get();
  return 0;
}

// ********************************************************************************
// ************************  END OF PROGRAMME *************************************
// ********************************************************************************
```

C MSSP 3.4: C++ source code

```
// ****************************************************************************
// ----------------------------------------------------------------------------
// ************************   MSSP 3.4   ************************************
// ----------------------------------------------------------------------------
// ****************************************************************************

// 15 April 2010, Kai Helge Becker

// ****************************************************************************
// Header files, constants, global variables
// ****************************************************************************

#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

// Global variables and constants
// Constants
   int const numinst = 10000000;     // number of instances
   int const numbox = 10;            // number of boxes
   int const numsc  = 2 * numbox;    // number of scores
   int const minwidth = 1;           // minimal width of each score
   int const maxwidth = 69;          // maximal width of each score
   int const thradj = 70;            // threshold for adjacency
   int const thrstrong = thradj / 2; // threshold for strong node (= thradj/2)
   int const empty = 999;            // flag for empty variable entry
                                     // (used w/ matlist, unconnode, lastmatch)
   int const nocomp
           = (numbox+(numbox%2))/2;  // no of components (chain + no of cycles)

// Variables
   int wbox[numsc];                  // width of all scores
   int i, j, k;                      // loops
   int stack;                        // stack
   int stackinv;
   int ordsc[numsc];                 // array index of ith smallest score
   int invordsc[numsc];              // inverse function of the above
   int instance;                     // counters for instances and cases
   int feacounter;
   int infcounter;
   int toomanyweak;
   int noncontwin;
   int uncon;
   int perfmat;
   int poormat;
   int suffmat;
   int yeahcounter;
   int completechain;
   int wscyclesandchain;
   int msscyclesandchain;
   int wwcyclesandchain;
   int mswcyclesandchain;
   int wwscyclesandchain;
   int mswscyclesandchain;
   int wswcyclesandchain;
   int msswcyclesandchain;
   int cyc1chainsplit;
   int cyc2chainsplit;
   int numberof09bcases;
   int ws1chainsplit;
   int ws2chainsplit;
   int ww1chainsplit;
   int ww2chainsplit;
   int wws1chainsplit;
   int wws2chainsplit;
```

```
int wsw1chainsplit;
int wsw2chainsplit;
int completechaininv;
int wscyclesandchaininv;
int msscyclesandchaininv;
int wwcyclesandchaininv;
int mswcyclesandchaininv;
int wwscyclesandchaininv;
int mswscyclesandchaininv;
int wswcyclesandchaininv;
int msswcyclesandchaininv;
int cyc1chainsplitinv;
int cyc2chainsplitinv;
int numberof12bcases;
int ws1chainsplitinv;
int ws2chainsplitinv;
int ww1chainsplitinv;
int ww2chainsplitinv;
int wws1chainsplitinv;
int wws2chainsplitinv;
int wsw1chainsplitinv;
int wsw2chainsplitinv;


int twinno1b;                 // place of an unconnectable box (case 1b)
double runtime1;              // running time
//double runtime2;            // running time without generating instances
//long time4rand;             // time for generating one instance
int adjlist[numsc] [numsc];   // adjacency list (based on sorted indices)
int matlist[numsc];           // matching list (based on sorted indices)
int matlistinv[numsc];        // matching list for inverse matching
int matcard;                  // cardinality of matching
int matcardinv;               // cardinality of inverse matching
int unconpointer;             // number of unconnected nodes
int unconpointerinv;          // same for inverse matching
int twinnomat;                // place of twinnode for matching (sorted ind)
int twinnomatinv;             // same 4 inverse matching
int lastmatch;                // place of last matched node (sorted ind)
int lastmatchinv;             // same 4 inverse matching
int weakeststrong;            // place of weakest strong node (sorted ind)
int unconnode[numsc];         // place of unconected nodes (sorted ind)
int unconnodeinv[numsc];      // place of unconectd node in inverse matching
int smallestuncon;            // smallest unconnected node (sorted ind)
int smallestunconinv;             // for INV case
int twin[numsc];              // place of twin node (sorted ind)
int analysed[numsc];          // flag if node already inclded in chain/cycle
int analysedinv[numsc];           // for INV case
int component[nocomp] [numsc];// nodes in chain [0] and cycles [1..nocomp-1]
int componentinv[nocomp] [numsc]; // for INV case
int lengthofcomponent[nocomp];// length of component (chain is component 0)
int lengthofcomponentinv[nocomp]; // for INV case
int lengthchain[(numsc+1)];   // distribution of chain length
int lengthchaininv[(numsc+1)];

int currentcomponent;         // no of cycle being analysed
int smallestconnotana;        // smallest connected node not analysed yet
int smallestconnotanainv;         // for INV case
int nocycles;                 // number of cycles
int nocyclesinv;              // number of cycles in INV case
int strongestnode[nocomp];    // characteristic of each cycle
int strongestweaknode[nocomp];
int strongestnodeinv[nocomp];
int strongestweaknodeinv[nocomp];
int weakestnode[nocomp];      // both used from 1..nocycles
int weakeststrongnode[nocomp];
int weakestnodeinv[nocomp];
```

```
int weakeststrongnodeinv[nocomp];
int strongeststrongest;       // characteristics of all cycles
int strongeststrongestinv;
int weakeststrongest;
int weakeststrongestinv;
int weakestweakest;
int weakestweakestinv;
int strongestweakest;
int strongestweakestinv;
int strongeststrongestweak;
int strongeststrongestweakinv;
int weakeststrongestweak;
int weakeststrongestweakinv;
int strongestweakeststrong;
int strongestweakeststronginv;
int weakestweakeststrong;
int weakestweakeststronginv;
int pss[nocomp];
int psw[nocomp];
int pws[nocomp];
int pww[nocomp];
int psss;
int pwss;
int psws;
int pwws;
int pssw;
int pwsw;
int psww;
int pwww;
int pssinv[nocomp];
int pswinv[nocomp];
int pwsinv[nocomp];
int pwwinv[nocomp];
int psssinv;
int pwssinv;
int pswsinv;
int pwwsinv;
int psswinv;
int pwswinv;
int pswwinv;
int pwwwinv;

int ordcyc[nocomp];            // order of cycles according to weakness of
                               // strongest node, for checkresult(8)
int ordcycinv[nocomp];         // chckresult 11
int placeofstrongestnode[nocomp]; // pl of s node in original order of cycles
                                  // input for component[nocomp] [XXX]
                                  // used for checkresult(8)
int placeofstrongestnodeinv[nocomp]; // chckresult (11)
int currentplace;              // next place in array "result" to be filled
int currentplaceinv;
int numberofelsecases;         // more than one cycle
int numberofelsecasesinv;

int stackcyc;                    // for result check (09a)
int stackcycinv;                 // for result check (12a)
int stackchain;                  // for result check (09a)
int stackchaininv;               // for result check (12a)
int splitplace;                  // for result check (09b)
int splitplaceinv;               // for result check (12b)
int connector[nocomp];           // for result check (09b)
int connectorinv[nocomp];        // for result check (12b)
int placeofconnector[nocomp];    // for result check (09b)
int placeofconnectorinv[nocomp]; // for result check (12b)
int caseno;                      // for result check (09ab) and (12ab)
int casetype;
```

```
   int distofcyclesstart[nocomp];// distribution of cycles b4 cycle analysis
   int distofcyclesleft[nocomp]; // distribution of cycles after cycle analysis
   int distofcyclesstartinv[nocomp]; // same for INV case
   int distofcyclesleftinv[nocomp];
   int prob;                     // counter for some problematic cases
   int probinv;                      // smallest for INV

   int result[numsc];                              // RESULT
   int resultcounter;
   int checkcasecounter[130];
   int problemcounter;
   int status;                                     // result of result check

// ***************************************************************************
// Function for checking results
// ***************************************************************************
  int checkresult (int subcase)
  {
    // Test
    // result[17] = 0;

    // Count check
    ++resultcounter;
    ++checkcasecounter[subcase];

    // Local variable
    int problem;                                          // flag for problem
    problem = 0;

    // Checking if there is a "999" node
    for(i=0; i<numsc; ++i)
    {
      if (result[i] == 999)
      {
        problem = 1;
        cout << endl << "999 case" << endl;
      }
    }

    // Checking twin node and matching characteristic except for last pair
    for(i=0; i<=(numsc-4); i=i+2)
    {
       // Checking twin node chacteristic
       if (  (((ordsc[result[i]]) % 2) == 0)                    // ordsc[i] even
          && (ordsc[result[i]] != ((ordsc[result[(i+1)]])-1)))
       {
         problem = 1;
         break;
       }
       if (  (((ordsc[result[i]]) % 2) == 1)                    // ordsc[i] odd
          && (ordsc[result[i]] != ((ordsc[result[(i+1)]])+1)))
       {
         problem = 1;
         break;
       }

       // Checking matching characeristic
       if (wbox[ordsc[result[(i+1)]]] + wbox[ordsc[result[(i+2)]]] < thradj)
       {
         problem = 1;
         break;
       }
    }

    // Checking twin node characteristic for last pair
```

```
if (  ((ordsc[result[(numsc-2)]] % 2) == 0)              // ordsc[i] even
   && (ordsc[result[(numsc-2)]] != ((ordsc[result[(numsc-1)]])-1)))
   problem = 1;
if (  ((ordsc[result[(numsc-2)]] % 2) == 1)              // ordsc[i] odd
   && (ordsc[result[(numsc-2)]] != ((ordsc[result[(numsc-1)]])+1)))
   problem = 1;

// For test if no problem
// if ((problem == 0) && (subcase == 122)) cout << "***" << endl;

// Consequences if problem
if (problem == 1)
{
  cout << "***** problem with subcase " << subcase << " *****" << endl;
  if (subcase == 92) cout << casetype << " / " << caseno << endl;
  if (subcase == 122) cout << casetype << " / " << caseno << endl;
  // chainanalysisok = 1;

  // Checking analysis of cycles and chain
  for(i=0; i<=nocyclesinv; ++i)
  {
    for(j=1; j<lengthofcomponentinv[i]; j=j+2)
    {
      if ((ordsc[componentinv[i] [j]] != (ordsc[componentinv[i] [j-1]]+1))
        && (ordsc[componentinv[i] [j]] != (ordsc[componentinv[i] [j-1]]-1)))
         cout << "twin wrong " << endl;
    }
    for(j=1; j<lengthofcomponentinv[i]-2; j=j+2)
    {
      if (wbox[ordsc[componentinv[i] [j]]]
          + wbox[ordsc[componentinv[i] [j+1]]]
         < thradj)
         cout << "mate wrong " << endl;
    }
  }
  // Printing variables for building result
  cout << "ordcycinv " << ordcycinv[1] << " " << ordcycinv[2] << endl;
  cout << "splitplace " << splitplaceinv << endl;
  cout << "connectorinv ";
  for (i=1; i<=nocyclesinv; ++i)
    cout << connectorinv[i] << "  ";
  cout << endl << "placeofconnectorinv ";
  for(i=1; i<=nocyclesinv; ++i)
    cout << placeofconnectorinv[i] << "  ";
  cout << endl << endl;

  cout << "wbox[i]:  ";
  for(i=0; i<numsc; ++i)
    cout << wbox[i] << "  ";
  cout << endl;
  cout << "wbox[ordsc[i]]:  ";
  for(i=0; i<numsc; ++i)
    cout << wbox[ordsc[i]] << "  ";
  cout << endl;
  cout << "result:  ";
  for(i=0; i<numsc; i=i+2)
    cout << wbox[ordsc[result[i]]] << "(" << ordsc[result[i]] << ")--("
         << ordsc[result[(i+1)]] << ")" << wbox[ordsc[result[(i+1)]]]
         << "  ";
  cout << endl << "number of cycles: " << nocycles << endl; // non-INV only
  cout << endl;
  ++problemcounter;
  // Note: The following printout is correct only in INV case
  for(i=0; i<=nocyclesinv; ++i)
  {
    for(j=0; j<lengthofcomponentinv[i]; ++j)
```

```
          cout << wbox[ordsc[componentinv[i] [j]]] << "-";
        cout << endl;
       }
       cout << endl << endl;
     }

     // Return problem status
     return problem;
   }

// ***************************************************************************
// Function main starts & initialisation
// ***************************************************************************
int main( )
{

// Welcome
   cout << "Welcome to MSSP 3.4." << endl;


// Initialisation
   feacounter = 0;               // initialising feasible instances counter
   infcounter = 0;               // initialising infeasible instances counter
   toomanyweak = 0;              // initialising case counters
   noncontwin  = 0;
   uncon = 0;
   perfmat = 0;
   poormat = 0;
   suffmat = 0;
   yeahcounter = 0;

   completechain = 0;
   wscyclesandchain = 0;
   msscyclesandchain = 0;
   wwcyclesandchain = 0;
   mswcyclesandchain = 0;
   wwscyclesandchain = 0;
   mswscyclesandchain = 0;
   wswcyclesandchain = 0;
   msswcyclesandchain = 0;
   cyc1chainsplit = 0;
   cyc2chainsplit = 0;
   numberof09bcases = 0;
   ws1chainsplit = 0;
   ws2chainsplit = 0;
   ww1chainsplit = 0;
   ww2chainsplit = 0;
   wws1chainsplit = 0;
   wws2chainsplit = 0;
   wsw1chainsplit = 0;
   wsw2chainsplit = 0;

   completechaininv = 0;
   wscyclesandchaininv = 0;
   msscyclesandchaininv = 0;
   wwcyclesandchaininv = 0;
   mswcyclesandchaininv = 0;
   wwscyclesandchaininv = 0;
   mswscyclesandchaininv = 0;
   wswcyclesandchaininv = 0;
   msswcyclesandchaininv = 0;
   cyc1chainsplitinv = 0;
   cyc2chainsplitinv = 0;
   numberof12bcases = 0;
   ws1chainsplitinv = 0;
   ws2chainsplitinv = 0;
```

```
   ww1chainsplitinv = 0;
   ww2chainsplitinv = 0;
   wws1chainsplitinv = 0;
   wws2chainsplitinv = 0;
   wsw1chainsplitinv = 0;
   wsw2chainsplitinv = 0;

   prob = 0;
   probinv = 0;

   resultcounter = 0;
   for(i=0; i<130; ++i)
     checkcasecounter[i] = 0;

   problemcounter = 0;
   numberofelsecases = 0;

   for(i=0; i<(numsc+1); ++i)
        lengthchain[i] = 0;
      for(i=0; i<nocomp; ++i)
      {
        distofcyclesstart[i] = 0;
        distofcyclesleft[i] = 0;
      }

   for(i=0; i<(numsc+1); ++i)
     lengthchaininv[i] = 0;
   for(i=0; i<nocomp; ++i)
   {
     distofcyclesstartinv[i] = 0;
     distofcyclesleftinv[i] = 0;
   }

// Initialising random mumbers
srand( (unsigned) time(NULL) );

// ******************************************************************************
// Start of instances loop
// ******************************************************************************
for (instance = 0; instance < numinst; ++instance)
{

// Producing uniformly distributed random numbers for scores
   //time4rand = clock();
   for(i = 0; i < numsc; ++i)
   {
      wbox[i] = minwidth + (rand() % (maxwidth - minwidth + 1));
   }

// Test data
/*
   wbox[19] = 1; wbox[9] = 30; //wbox[10] = 35; wbox[11] = 36;

   for(i = 10; i < 19; ++i)
   {
     wbox[i] = 20 + (rand() % 6);
   }
   for(i = 12; i < 20; ++i)
   {
      wbox[i] = 10 + (rand() % 11);
   }
*/


/*AAA
```

```
// Output instance
   cout << endl;
   for(i = 0; i < numsc; ++i)
   {
      cout << wbox[i] << "   ";
   }
   cout << endl;
*/

   // Running time without generating instances
   // time4rand = clock() - time4rand;
   // cout << time4rand << endl;

// ****************************************************************************
// CASE (01): Less than numbox - 1 strong nodes
// ****************************************************************************
   j = 0;
   for(i = 0; i < numsc; ++i)
   {
      if (wbox[i] >= thrstrong)
         ++j;
   }
   if (j < numbox - 1)
   {
      ++infcounter;
      ++toomanyweak;
      //AAAcout << "inf: too many weak nodes, namely: " << numsc-j << endl;
      continue;
   }

// ****************************************************************************
// Sorting scores from smallest to largest
// ****************************************************************************
   // Initialising order
   for(i = 0; i < numsc; ++i)
   {
      ordsc[i] = i;
   }
   // Starting sorting procedure
   for(i = 1; i < numsc; ++i)
   {
      for(j = i-1; j >= 0; --j)
      {
         //cout << i << "  " << ordsc[i] << "  "
         //     << j << "   " << ordsc[j] << endl;
         if (wbox[i] < wbox[ordsc[j]])
         {
            ordsc[j+1] = ordsc[j];
            ordsc[j] = i;
            //cout << "after change  " << i << "  " << ordsc[i] << "  "
            //     << j << "    " << ordsc[j] << endl;
         }
         //else { cout << "no change, next i" << endl; break; }
      }
   }

/*
// Insertion sort
   for(i = 2; i <= numsc; ++i)
   {
      j = 1;
      while ((j < i) & (wbox[ordsc[j]] <= wbox[i]))
      {
      ++j;
      }
      if  (j < i)  // this implies that wbox[ordsc[j]] > wbox[i]
```

```
      {
         stack = i;
         for(k = i-1; k >= j; --k)
         {
            ordsc[k+1] = ordsc[k]
         }
         ordsc[j] = i
      }
   }
*/
/*
// Output sorted instance
   cout << "order: ";
   for(i = 0; i < numsc; ++i)
   {
         cout << wbox[ordsc[i]] << "   ";
   }
   cout << endl;
*/

// ****************************************************************************
// CASE (02): Two non-connectable twin nodes
// ****************************************************************************
   j = 0;
   for(i = 0; i < numbox; ++i)
   {
      if (wbox[2*i] + wbox[ordsc[(numsc-1)]] < thradj)
        {
           if (wbox[2*i + 1] + wbox[ordsc[(numsc-1)]] < thradj)
              {
               ++j;
               //AAAtwinno = i;
              }
        }
   }
   if (j >= 1)
   {
      ++infcounter;
      ++noncontwin;
      /*
      // Output instance
      for(k = 1; k <= numsc; ++k)
      {
         cout << wbox[k] << "   ";
      }
      cout << endl;
      // Output sorted instance
      cout << "order: ";
      for(k = 1; k <= numsc; ++k)
      {
         cout << wbox[ordsc[k]] << "   ";
      }
      cout << endl;
      */
      //AAAcout << "inf: two non-connectable twin nodes at pair "
      //        << twinno << endl;
      continue;
   }

// ****************************************************************************
// CASE (03): Three non-connectable nodes
// ****************************************************************************
   j = 0;
   for(i = 0; i < numsc; ++i)
   {
      if (wbox[i] + wbox[ordsc[(numsc-1)]] < thradj)
```

```
      ++j;
   }
   if (j >= 3)
   {
      ++infcounter;
      ++uncon;
      //AAAcout << "inf: too many unconnectable nodes,"
      //        << "largest number has order  " << ordsc[numsc] << endl;
      continue;
   }

// ****************************************************************************
// Matching algorithm
// ****************************************************************************

// Step 1: List with adjacent nodes disregarding twin nodes
   for(i = 0; i < numsc; ++i)  // general adjacency list
   {
     for(j = 0; j < numsc; ++j)
     {
       if (wbox[ordsc[i]] + wbox[ordsc[j]] >= thradj)
       {
         adjlist[i] [j] = 1;
       }
       else
       {
         adjlist[i] [j] = 0;
       }
     }
   }

   for (i=0; i<numsc; ++i)     // generating inverse function of ordsc[]
   {
     invordsc[ordsc[i]] = i;
   }

   for (i = 0; i < numbox; ++i)// twin nodes cannot be connected to eachother
   {
     adjlist[invordsc[(2*i)]] [invordsc[(2*i+1)]] = 0;
     adjlist[invordsc[(2*i+1)]] [invordsc[(2*i)]] = 0;
   }

// Step 2: Initialise matching list and counters
   for (i = 0; i < numsc; ++i)
   {
     matlist[i] = empty;
   }
   matcard = 0;
   unconpointer = 0;
   for (i = 0; i < numsc; ++i)
     {
       unconnode[i] = empty;
     }
   lastmatch = empty;

// Step 3: Matching algorithm
   for(i=0; i<numsc; ++i)// check all nodes
   {
     if (matlist[i] == empty)// does node need a mate?
     {
       for (j=(i+1); j<numsc; ++j)// look for a mate for node i
       {
         if (  (adjlist[i] [j] == 1)
            && (matlist[j] == empty))// if mate found
         {
           matlist[i] = j;
```

```
          matlist[j] = i;
          lastmatch = i;
          ++matcard;
          break;
        }
      }
      if (matlist[i] == empty)   // if there still is no mate:
      {                          // do twin node swap or finally acquiesce
        if (ordsc[i] % 2 == 0)     // find out twin node number
        {
          twinnomat = invordsc[(ordsc[i]+1)];
        }
        else
        {
          twinnomat = invordsc[(ordsc[i]-1)];
        }
        if                         // twin node swap possible?
        (  (wbox[ordsc[i]]+wbox[ordsc[twinnomat]]>=thradj)// match with twin?
        && (matlist[twinnomat] == empty)                  // twin unmatched?
        && (lastmatch != empty)                           // exchange possble?
        && (twinnomat > i)                                // twin larger?
        && (wbox[ordsc[lastmatch]]+wbox[ordsc[twinnomat]]>=thradj))// lastmtch
        {                                                         // with twin?
          matlist[i] = matlist[lastmatch];
          matlist[lastmatch] = twinnomat;
          matlist[twinnomat] = lastmatch;
          matlist[matlist[i]] = i;
          lastmatch = i;
          ++matcard;
        }
        else                       // otherwise: one more unconnected node
        {
          ++unconpointer;
          unconnode[(unconpointer-1)] = i;
        }
      }
    }
  }

/*
// Output matching list and unconnected nodes
   for(i=0; i<numsc; ++i)
   {
     cout << matlist[i] << " ";
   }
   cout << endl;

   for(i=0; i<numsc; ++i)
   {
     cout << unconnode[i] << " ";
   }
   cout << endl;
   cout << "matcard: " << matcard;
   cout << "  no of unconnodes:  " << unconpointer << endl;
*/

// ***************************************************************************
// CASE (05): Perfect matching
// ***************************************************************************
   if (matcard == numbox)
   {
     ++perfmat;
     ++feacounter;

/*AAA
     cout << "fea: perfmatch with card: " << matcard << "  ";
```

```
     for (i=0; i<numsc; ++i)
     {
       cout << " w1= " << wbox[ordsc[i]] << " w2= "
            << wbox[ordsc[matlist[i]]] << " -- ";
     }
     cout << endl;
*/

     continue;
   }

// ****************************************************************************
// CASE (04): Poor matching
// ****************************************************************************
   if (matcard < (numbox-1))
   {
     ++poormat;
     ++infcounter;

/*AAA
     cout << "inf: poor matching with card: " << matcard
          << " and uncon nodes: ";
     for (i=0; i<numsc; ++i)
     {
       if (unconnode[i] != empty)
         cout << unconnode[i] << " w= " << wbox[ordsc[unconnode[i]]]
              << " " << endl;
     }
     cout << endl;
*/
     continue;
   }

// ****************************************************************************
// CASE (06): Sufficient matching
// ****************************************************************************

  for (i=0; i<numsc; ++i)  // Find weakest strong node
  {
    if (wbox[ordsc[i]] >= thrstrong)
    {
       weakeststrong = i;
       break;
    }
  }

/*AAA
  cout << "card: " << matcard << " no of wkststr " << weakeststrong
       << " w= " << wbox[ordsc[weakeststrong]]
       << " ucn0 " << unconnode[0] << " w= " << wbox[ordsc[unconnode[0]]]
       << " ucn1 " << unconnode[1] << " w= " << wbox[ordsc[unconnode[1]]]
       << endl;
  cout << "matching: ";
  for (i=0; i<numsc; ++i)
  {
    if (matlist[i] != empty)
      cout << " w1= " << wbox[ordsc[i]]
           << " w2= " << wbox[ordsc[matlist[i]]] << " -- ";
  }
*/
/*
   if (matlist[numsc] == empty)
   {
     cout << "unconnode[0] " << unconnode[0]
          << "  unconnode[1]  " << unconnode[1]
          << "  matcard  " << matcard << endl;
```

```
   }
*/

  if (wbox[ordsc[unconnode[1]]]
      + wbox[ordsc[weakeststrong]] >= thradj) // Check sufficiency
  {
    ++suffmat;
    ++feacounter;
    //AAAcout << " yeah suff" << endl;
    if (unconnode[1] < weakeststrong)
      ++yeahcounter;  //cout << " YEAH!!!" << endl;
    continue;
  }
  //AAAcout << " not suff" << endl;

// ****************************************************************************
// Building up chain
// ****************************************************************************

// Step 1: Initialise data
  for(i=0; i<numsc; ++i)
  {
    analysed[i] = 0;
    for(j=0; j<nocomp; ++j)
      component[j] [i] = empty;
    if (ordsc[i] % 2 == 0) // find out twin node number
      twin[i] = invordsc[(ordsc[i]+1)];
    else
      twin[i] = invordsc[(ordsc[i]-1)];
    //cout << ordsc[i] << "-" << ordsc[twin[i]] << "  ";
  }
  //cout << endl;

// Step 2: Build up chain
  for(i=0; i<numsc; ++i)     // find smallest unconnected node
  {
    if (matlist[i] == empty)
    {
      smallestuncon = i;
      break;
    }
  }
  if (smallestuncon != unconnode[0])
    cout << "ALARM!!!" << endl;

  j = -1;                    // build up chain
  stack = smallestuncon;
  do
  {
    ++j;
    component[0] [j] = stack;
    analysed[stack] = 1;
    ++j;
    component[0] [j] = twin[stack];
    analysed[twin[stack]] = 1;
    stack = matlist[twin[stack]];
  }
  while (stack != empty);

  lengthofcomponent[0] = ++j;

// ****************************************************************************
// Case (07): Chain complete with length = numsc
// ****************************************************************************
  if (lengthofcomponent[0] == numsc)// is chain complete?
  {
```

```
    ++completechain;
    ++feacounter;
    nocycles = 0;
    for(i=0; i<numsc; ++i)
      result[i] = component[0] [i];
    i = checkresult(7);
/*
    for(i=0; i<numbox; ++i)
    {
      cout << wbox[ordsc[component[0] [(2*i)]]] << "("
           << ordsc[component[0] [(2*i)]] << ")-("
           << ordsc[component[0] [(2*i+1)]] << ")"
           << wbox[ordsc[component[0] [(2*i+1)]]] << " -- ";
    }
    cout << "*" << endl;
*/
    continue;
  }

// ***************************************************************************
// Building up cycles
// ***************************************************************************
  for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
    {
    if (analysed[i] == 0)
      {
        smallestconnotana = i;
        break;
      }
    }
  if (matlist[smallestconnotana] == empty)
       cout << "ARLARM2!!!" << endl;

  currentcomponent = 0;
  do
  {
     ++currentcomponent;      // Set component
     if (currentcomponent > nocomp-1)
       cout << "ARLARM3!!!" << endl;

     j = -1;                  // Build up cycle
     stack = smallestconnotana;
     do
     {
       ++j;
       component[currentcomponent] [j] = stack;
       analysed[stack] = 1;
       ++j;
       component[currentcomponent] [j] = twin[stack];
       analysed[twin[stack]] = 1;
       stack = matlist[twin[stack]];
     }
     while (stack != smallestconnotana); // = while not back 2 beginning of cyc

     lengthofcomponent[currentcomponent] = ++j;

     for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
       if (analysed[i] == 0)
       {
          smallestconnotana = i;
          break;
       };
     if (matlist[smallestconnotana] == empty)
       cout << "ARLARM2B!!!" << endl;
  }
  while (smallestconnotana != stack); // = while new not analysed node found
```

```
  nocycles = currentcomponent;        // save number of cycles

// ****************************************************************************
// Analysing cycles and chain
// ****************************************************************************
  for (i=0; i<nocomp; ++i)  // Initialising data
    {
      strongestnode[i] = empty;
      pss[i] = empty;
      strongestweaknode[i] = empty;
      psw[i] = empty;
      weakestnode[i] = empty;
      pww[i] = empty;
      weakeststrongnode[i] = empty;
      pws[i] = empty;
    }

  for (i=1; i<=nocycles; ++i)  // Check all cycles
  {
    // initialise for this component
    strongestnode[i] = component[i] [0];
    pss[i] = 0;
    weakestnode[i] = component[i] [0];
    pww[i] = 0;
    if (wbox[ordsc[component[i] [1]]] >= thrstrong)
    {
      weakeststrongnode[i] = component[i] [1];
      pws[i] = 1;
      strongestweaknode[i] = component[i] [2];
      psw[i] = 2;
      // ATTN: not clear if weak node!!!
      // (will be repaired down below if any cycle has a 'real' (= truly weak)
      // strongestweaknode)
    }
    else
    {
      weakeststrongnode[i] = component[i] [2];
      pws[i] = 2;
      strongestweaknode[i] = component[i] [1];
      psw[i] = 1;
    }
    // try to repair if current strongestweaknode[i] is not weak
    if (wbox[ordsc[strongestweaknode[i]]] >= thrstrong)
    {
      for (j=0; j<lengthofcomponent[i]; ++j)
      {
        if (wbox[ordsc[component[i] [j]]] < thrstrong)
          {
            strongestweaknode[i] = component[i] [j];
            psw[i] = j;
          }
      }
    }
    // note1: if there is no weak node in this cycle at all, we will later,
    // at (**), set strongestweaknode[i] := weakeststrongnode[i]
    // note2: if there is no weak node in this cycle at all, the algorithm
    // will automatically set weakest(weak)node[i] := weakeststrongnode[i],
    // so this case is repaired automatically

    // check for all elements of this component
    for (j=0; j<lengthofcomponent[i]; ++j) // Find weakest & strongest in cycle
    { // > cases

      if (wbox[ordsc[component[i] [j]]] > wbox[ordsc[strongestnode[i]]])
        {
          strongestnode[i] = component[i] [j];
```

```
        pss[i] = j;
      }
    if (wbox[ordsc[component[i] [j]]] < wbox[ordsc[weakestnode[i]]])
      {
        weakestnode[i] = component[i] [j];
        pww[i] = j;
      }
    if ((wbox[ordsc[component[i] [j]]] > wbox[ordsc[strongestweaknode[i]]])
       && (wbox[ordsc[component[i] [j]]] < thrstrong))
      {
        strongestweaknode[i] = component[i] [j];
        psw[i] = j;
      }
    if ((wbox[ordsc[component[i] [j]]] < wbox[ordsc[weakeststrongnode[i]]])
       && (wbox[ordsc[component[i] [j]]] >= thrstrong))
      {
        weakeststrongnode[i] = component[i] [j];
        pws[i] = j;
      }
    // == cases
    if ((wbox[ordsc[component[i] [j]]] == wbox[ordsc[strongestnode[i]]])
     && (wbox[ordsc[matlist[component[i] [j]]]]
         > wbox[ordsc[matlist[strongestnode[i]]]]))
      {
        strongestnode[i] = component[i] [j];
        pss[i] = j;
      }
    if ((wbox[ordsc[component[i] [j]]] == wbox[ordsc[weakestnode[i]]])
     && (wbox[ordsc[matlist[component[i] [j]]]]
         < wbox[ordsc[matlist[weakestnode[i]]]]))
      {
        weakestnode[i] = component[i] [j];
        pww[i] = j;
      }
    if ((wbox[ordsc[component[i] [j]]] == wbox[ordsc[strongestweaknode[i]]])
     && (wbox[ordsc[matlist[component[i] [j]]]]
         > wbox[ordsc[matlist[strongestweaknode[i]]]]))
      {
        strongestweaknode[i] = component[i] [j];
        psw[i] = j;
      }
    if ((wbox[ordsc[component[i] [j]]] == wbox[ordsc[weakeststrongnode[i]]])
     && (wbox[ordsc[matlist[component[i] [j]]]]
         < wbox[ordsc[matlist[weakeststrongnode[i]]]]))
      {
        weakeststrongnode[i] = component[i] [j];
        pws[i] = j;
      }
  }

  // (**) if cycle has no weak node (see note1 above): repair
  if (wbox[ordsc[strongestweaknode[i]]] >= thrstrong)
    {
      strongestweaknode[i] = weakeststrongnode[i];
      psw[i] = pws[i];
    }
}

  // initialise charcateristics for all cycles
weakeststrongest = strongestnode[1];
pwss = pss[1];
strongeststrongest = strongestnode[1];
psss = pss[1];
weakestweakest = weakestnode[1];    // could be strong
pwww = pww[1];
strongestweakest = weakestnode[1];  // could be strong
```

```
psww = pww[1];
weakeststrongestweak = strongestweaknode[1]; // could be strong
pwsw = psw[1];
strongeststrongestweak = strongestweaknode[1]; // could be strong
                         // forall 4 cases: iff there is no weak node in cycle
pssw = psw[1];
weakestweakeststrong = weakeststrongnode[1];
pwws = pws[1];
strongestweakeststrong = weakeststrongnode[1];
psws = pws[1];

  // check for all cycles
for(i=1; i<=nocycles; ++i)
{
  // < cases
  if (wbox[ordsc[strongestnode[i]]] < wbox[ordsc[weakeststrongest]])
    {
      weakeststrongest = strongestnode[i];
      pwss = pss[i];
    }
  if (wbox[ordsc[strongestnode[i]]] > wbox[ordsc[strongeststrongest]])
    {
      strongeststrongest = strongestnode[i];
      psss = pss[i];
    }

  if (wbox[ordsc[weakestnode[i]]] < wbox[ordsc[weakestweakest]])
    {
      weakestweakest = weakestnode[i]; // only strong if there were no wk node
      pwww = pww[i];
    }
  if (wbox[ordsc[weakestnode[i]]] > wbox[ordsc[strongestweakest]])
    {
      strongestweakest = weakestnode[i]; // mayb str, but ok 4 buildin solutn
      psww = pww[i];
    }
  if (wbox[ordsc[strongestweaknode[i]]] < wbox[ordsc[weakeststrongestweak]])
    {
      weakeststrongestweak = strongestweaknode[i]; // only str s'ilyavai no wk
      pwsw = psw[i];
    }
  if (wbox[ordsc[strongestweaknode[i]]] > wbox[ordsc[strongeststrongestweak]])
    {
      strongeststrongestweak = strongestweaknode[i]; // mayb strong, but ok
      pssw = psw[i];
    }
  if (wbox[ordsc[weakeststrongnode[i]]] < wbox[ordsc[weakestweakeststrong]])
    {
      weakestweakeststrong = weakeststrongnode[i];
      pwws = pws[i];
    }
  if (wbox[ordsc[weakeststrongnode[i]]] > wbox[ordsc[strongestweakeststrong]])
    {
      strongestweakeststrong = weakeststrongnode[i];
      psws = pws[i];
    }
  // == cases
  if ((wbox[ordsc[strongestnode[i]]]
       == wbox[ordsc[weakeststrongest]])
   && (wbox[ordsc[matlist[strongestnode[i]]]]
       < wbox[ordsc[matlist[weakeststrongest]]]))
    {
      weakeststrongest = strongestnode[i];
      pwss = pss[i];
    }
  if ((wbox[ordsc[strongestnode[i]]]
```

```
          == wbox[ordsc[strongeststrongest]])
       && (wbox[ordsc[matlist[strongestnode[i]]]]
           > wbox[ordsc[matlist[strongeststrongest]]]))
        {
          strongeststrongest = strongestnode[i];
          psss = pss[i];
        }

      if ((wbox[ordsc[weakestnode[i]]]
           == wbox[ordsc[weakestweakest]])
       && (wbox[ordsc[matlist[weakestnode[i]]]]
           < wbox[ordsc[matlist[weakestweakest]]]))
        {
          weakestweakest = weakestnode[i]; // only strong if there were no wk node
          pwww = pww[i];
        }
      if ((wbox[ordsc[weakestnode[i]]]
           == wbox[ordsc[strongestweakest]])
       && (wbox[ordsc[matlist[weakestnode[i]]]]
           > wbox[ordsc[matlist[strongestweakest]]]))
        {
          strongestweakest = weakestnode[i]; // mayb str, but ok 4 buildin solutn
          psww = pww[i];
        }
      if ((wbox[ordsc[strongestweaknode[i]]]
           == wbox[ordsc[weakeststrongestweak]])
       && (wbox[ordsc[matlist[strongestweaknode[i]]]]
           < wbox[ordsc[matlist[weakeststrongestweak]]]))
        {
          weakeststrongestweak = strongestweaknode[i]; // only str s'ilyavai no wk
          pwsw = psw[i];
        }
      if ((wbox[ordsc[strongestweaknode[i]]]
           == wbox[ordsc[strongeststrongestweak]])
       && (wbox[ordsc[matlist[strongestweaknode[i]]]]
           > wbox[ordsc[matlist[strongeststrongestweak]]]))
        {
          strongeststrongestweak = strongestweaknode[i]; // mayb strong, but ok
          pssw = psw[i];
        }
      if ((wbox[ordsc[weakeststrongnode[i]]]
           == wbox[ordsc[weakestweakeststrong]])
       && (wbox[ordsc[matlist[weakeststrongnode[i]]]]
           < wbox[ordsc[matlist[weakestweakeststrong]]]))
        {
          weakestweakeststrong = weakeststrongnode[i];
          pwws = pws[i];
        }
      if ((wbox[ordsc[weakeststrongnode[i]]]
           == wbox[ordsc[strongestweakeststrong]])
       && (wbox[ordsc[matlist[weakeststrongnode[i]]]]
           > wbox[ordsc[matlist[strongestweakeststrong]]]))
        {
          strongestweakeststrong = weakeststrongnode[i];
          psws = pws[i];
        }

    }

    // Counting number of cycles b4 analysis
    ++distofcyclesstart[nocycles];

  // ***************************************************************************
  // CASE 08: Connection of cycles with chain via weakest strongest strong node
  // ***************************************************************************
  // Step 1: Make sure that unconnode[1] really is the higher unconnected node
```

```
  if (wbox[ordsc[unconnode[0]]] > wbox[ordsc[unconnode[1]]])
    cout << "ALARM4!!!" << endl;

// Step 2: Check w/ wkst strgst strong
  if (wbox[ordsc[weakeststrongest]]
      + wbox[ordsc[unconnode[1]]] >= thradj)
  {
  ++wscyclesandchain;
  ++feacounter;
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // Step 2ba: Checking result for WSS node case if there is only one cycle
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   if (nocycles == 1)
   {
     /*
     for(i=0; i<lengthofcomponent[1]; ++i)//check where in cycle strngst node
     {
       if ((component[1][i])==strongestnode[1])
       {
         stack = i;
         break;
       }
     }
     */
     stack = pss[1];

     if (twin[strongestnode[1]]==component[1] [(stack+1)])//twin after str nd
     {
       for(i=0; i<lengthofcomponent[0]; ++i)
         result[i] = component[0] [i];
       for(i=stack; i<lengthofcomponent[1]; ++i)
         result[(i+lengthofcomponent[0]-stack)] = component[1] [i];
       if (stack != 0)                          // str nd is not first node
       {
         for(i=0; i<stack; ++i)
           result[(i+lengthofcomponent[0]+lengthofcomponent[1]-stack)]
           = component[1] [i];
       }
     }
     else   // twin before strongest node
     {
       for(i=0; i<lengthofcomponent[0]; ++i)
         result[i] = component[0] [i];
       for(i=stack; i>=0; --i)
         result[(lengthofcomponent[0]+stack-i)] = component[1] [i];
       if (stack != (lengthofcomponent[1]-1))  // str nd is not last node
       {
         for(i=(lengthofcomponent[1]-1); i>stack; --i)
           result[(lengthofcomponent[0]+stack+(lengthofcomponent[1]-i))]
           = component[1] [i];
       }
     }
   }
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // Step 2bb: Checking result for WSS case if there is more than one cycle
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

   else                                        // There is more than one cycle
   {
     // Initialising order
     for(i = 1; i <= nocycles; ++i)
     {
        ordcyc[i] = i;
     }
     // Starting sorting procedure for cycles according to strngest strong nodes
```

```
for(i = 1; i <= nocycles; ++i)
{
   for(j = i-1; j >= 1; --j)
   {
    //cout << i << "  " << ordsc[i] << "  "
    //     << j << "   " << ordsc[j] << endl;
    // abbreviation = ;
     if (wbox[ordsc[strongestnode[i]]]
         < wbox[ordsc[strongestnode[ordcyc[j]]]])
     {
       ordcyc[j+1] = ordcyc[j];
       ordcyc[j] = i;
         //cout << "after change  " << i << "  " << ordsc[i] << "  "
         //     << j << "    " << ordsc[j] << endl;
     }
       //else { cout << "no change, next i" << endl; break; }
   }
}
// Looking for the place of strongest node in each cycle
// because this tells us where to start building the result out of cycles
/*
for(i = 1; i <= nocycles; ++i)
{
  placeofstrongestnode[i] = empty;
}
for(i = 1; i <= nocycles; ++i)
{
  for(j = 0; j < lengthofcomponent[i]; ++j)
  {
    if (strongestnode[i] == component[i][j])
    {
      placeofstrongestnode[i] = j;
      break;
    }
  }
}
*/
// Starting result array with chain
for(i=0; i<lengthofcomponent[0]; ++i)
    result[i] = component[0] [i];
currentplace = lengthofcomponent[0];
// Now add the cycles to the result
for(k = 1; k <= nocycles; ++k)
{
  // first option: twin comes after strongest node in this cycle
  if (twin[strongestnode[ordcyc[k]]]
      == component[ordcyc[k]] [(pss[ordcyc[k]]+1)])
  {
     for(i = pss[ordcyc[k]];
         i < lengthofcomponent[ordcyc[k]]; ++i)
       result[(  i
               + currentplace
               - pss[ordcyc[k]])]
       = component[ordcyc[k]] [i];
     if (pss[ordcyc[k]] != 0) // str nd is not first node
     {
       for(i=0; i<pss[ordcyc[k]]; ++i)
         result[(  i
                 + currentplace
                 + lengthofcomponent[ordcyc[k]]
                 - pss[ordcyc[k]])]
         = component[ordcyc[k]] [i];
     }
   }
   // second option: twin comes b4 strongest node in this cycle
   else
```

```
        {
          for(i=pss[ordcyc[k]]; i>=0; --i)
            result[(  currentplace
                    + pss[ordcyc[k]]
                    - i)]
            = component[ordcyc[k]] [i];
          if (pss[ordcyc[k]]
              != (lengthofcomponent[ordcyc[k]]-1))  // str nd is not last node
          {
             for(i = (lengthofcomponent[ordcyc[k]]-1);
                 i > pss[ordcyc[k]]; --i)
               result[(  currentplace
                       + pss[ordcyc[k]]
                       + (lengthofcomponent[ordcyc[k]]-i))]
               = component[ordcyc[k]] [i];
          }
        }
        // store new position in building result array
        currentplace = currentplace + lengthofcomponent[ordcyc[k]];
      } // end of loop through all cycles
     ++numberofelsecases;
     } // end of else for case of more than one cycle

     status = checkresult(8);
    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    continue;
    }

  // Step 3: Check for mate of strgst strgst strong,
  //                    wkst wkst weak, and mate of strgst wkst weak
    if (wbox[ordsc[matlist[strongeststrongest]]]
        + wbox[ordsc[unconnode[1]]] >= thradj)
    {
    ++msscyclesandchain;
    ++feacounter;
    continue;
    }
    if (wbox[ordsc[weakestweakest]]
        + wbox[ordsc[unconnode[1]]] >= thradj)
    {
    ++wwcyclesandchain;
    ++feacounter;
    continue;
    }
    if (wbox[ordsc[matlist[strongestweakest]]]
        + wbox[ordsc[unconnode[1]]] >= thradj)
    {
    ++mswcyclesandchain;
    ++feacounter;
    continue;
    }

  // Step 4: Check w/ wkst wkst strong, mate of strgst wkst strong
  //                    wkst strgst weak, mate of strgst strgst weak
    if (wbox[ordsc[weakestweakeststrong]]
        + wbox[ordsc[unconnode[1]]] >= thradj)
    {
    ++wwscyclesandchain;
    ++feacounter;
    continue;
    }
    if (wbox[ordsc[matlist[strongestweakeststrong]]]
        + wbox[ordsc[unconnode[1]]] >= thradj)
    {
    ++mswscyclesandchain;
    ++feacounter;
```

```
  continue;
  }
  if (wbox[ordsc[weakeststrongestweak]]
      + wbox[ordsc[unconnode[1]]] >= thradj)
  {
  ++wswcyclesandchain;
  ++feacounter;
  continue;
  }
  if (wbox[ordsc[matlist[strongeststrongestweak]]]
      + wbox[ordsc[unconnode[1]]] >= thradj)
  {
  ++msswcyclesandchain;
  ++feacounter;
  continue;
  }

// ***************************************************************************
// CASE (09a): Splitting up chain to connect to cyle if nocycles == 1
// ***************************************************************************

  if ((nocycles == 1) && (lengthofcomponent[0] >= 4))
  {
    k = 0; // counter if a subcase has been detected
    for (i=1; i<=(lengthofcomponent[0]-3); i=i+2)
    {
      for (j=1; j<=(lengthofcomponent[1]-1); j=j+2)
      {
        if ((adjlist[component[1] [j]] [component[0] [i]] == 1)
           && (adjlist[matlist[component[1] [j]]] [component[0] [(i+1)]] == 1))
           {
           stackcyc = j;
           stackchain = i;
           caseno = 1;
           ++cyc1chainsplit;
           ++feacounter;
           ++k;
           break;
           }
        if ((adjlist[component[1] [j]] [component[0] [i+1]] == 1)
           && (adjlist[matlist[component[1] [j]]] [component[0] [i]] == 1))
           {
           stackcyc = j;
           stackchain = i;
           caseno = 2;
           ++cyc2chainsplit;
           ++feacounter;
           ++k;
           break;
           }
      }
      if (k == 1) break;
    }
    if (k == 1)
    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    // Start check for chainsplit with one cycle
    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    {
      // relevant component of cyc in var "stackcyc", "stackchain" similar
      // caseno = 1 or 2 signifies the case above
      // First: case 1 && stackchain is not last node of cycle
      if ((caseno == 1) && (stackcyc != (lengthofcomponent[1] - 1)))
      {
        for (i=0; i<=lengthofcomponent[0]-stackchain-2; ++i)// upper prt  chain
          result[i] = component[0] [lengthofcomponent[0]-1-i];
```

```
      for (i=0; i<=lengthofcomponent[1]-stackcyc-2; ++i)// upper prt of cycle
        result[i+lengthofcomponent[0]-stackchain-1]
        = component[1] [stackcyc+1+i];

      for (i=0; i<=stackcyc; ++i)                      // lower part of cycle
        result[i+lengthofcomponent[1]-stackcyc-2
               +lengthofcomponent[0]-stackchain] = component[1] [i];

      for (i=0; i<=stackchain; ++i)                    // lower part of chain
        result[i+lengthofcomponent[1]-1+lengthofcomponent[0]-stackchain]
        = component[0] [stackchain-i];
    }
    // Second: case 1 && stackchain is last node of cycle
    if ((caseno == 1) && (stackcyc == (lengthofcomponent[1] - 1)))
    {
      for (i=0; i<=stackchain; ++i)
        result[i] = component[0] [i];               // lower part of chain
      for (i=0; i<=(lengthofcomponent[1] - 1); ++i)   // whole cycle backwards
        result[i+stackchain+1] = component[1] [lengthofcomponent[1]-1-i];
      for (i=0; i<=(lengthofcomponent[0]-2-stackchain); ++i) // upper chain
        result[i+stackchain+1+lengthofcomponent[1]]
        = component[0] [i+stackchain+1];
    }
    // Third: case 2 && stackchain is not last node of cycle
    if ((caseno == 2) && (stackcyc != (lengthofcomponent[1] - 1)))
    {
      for (i=0; i<=stackchain; ++i)              // lower part of chain
        result[i] = component[0] [i];
      for (i=0; i <= (lengthofcomponent[1] - 2 - stackcyc); ++i) // upper cyc
        result[i+1+stackchain] = component[1] [i+stackcyc+1];
      for (i=0; i<=stackcyc; ++i)                    // lower cyc
        result[i+stackchain+lengthofcomponent[1]-stackcyc] = component[1] [i];
      for (i=0; i<=(lengthofcomponent[0] - 2 - stackchain); ++i) // upper chn
        result[i+1+stackchain+lengthofcomponent[1]]
        = component[0] [i+1+stackchain];
    }
     // Fourth: case 2 && stackchain is last node of cycle
    if ((caseno == 2) && (stackcyc == (lengthofcomponent[1] - 1)))
    {
      for(i=0; i<=stackchain; ++i)     // lower part of chain
        result[i] = component[0] [i];
      for(i=0; i<=(lengthofcomponent[1]-1); ++i)  // cycle forward
        result[i+1+stackchain] = component[1] [i];
      for(i=0; i<=(lengthofcomponent[0]-stackchain-2); ++i) // upper chain
        result[i+1+stackchain+lengthofcomponent[1]]
        = component[0] [stackchain+1+i];
    }

    status = checkresult(91);
    // ++++++++++++++++++++++++++++++++++++++++
    // End of check Chainsplit with one cycle
    // ++++++++++++++++++++++++++++++++++++++++
    continue;
  }

 }   // End of case (09a)

// ************************************************************************
// CASE (09b): Splitting up chain to connect to ends of cycles, 8 subcases
// ************************************************************************
  if ((nocycles > 1) && (lengthofcomponent[0] >= 4))
  {

    j = 0; // counter if a subcase has been detected
    for (i=1; i<=(lengthofcomponent[0]-3); i=i+2)
    {
```

```
if ((adjlist[weakeststrongest] [component[0] [i]] == 1)
   && (adjlist[matlist[strongeststrongest]] [component[0] [(i+1)]] == 1))
{
caseno = 1;
casetype = 22;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = strongestnode[k];
    placeofconnector[k] = pss[k];
  }
splitplace = i;
++ws1chainsplit;
++feacounter;
++j;
break;
}
if ((adjlist[weakeststrongest] [component[0] [(i+1)]] == 1)
   && (adjlist[matlist[strongeststrongest]] [component[0] [i]] == 1))
{
caseno = 2;
casetype = 22;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = strongestnode[k];
    placeofconnector[k] = pss[k];
  }
splitplace = i;
++ws2chainsplit;
++feacounter;
++j;
break;
}

if ((adjlist[weakestweakeststrong] [component[0] [i]] == 1)
   && (adjlist[matlist[strongestweakeststrong]] [component[0] [(i+1)]]
   == 1))
{
caseno = 1;
casetype = 12;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = weakeststrongnode[k];
    placeofconnector[k] = pws[k];
  }
splitplace = i;
++wws1chainsplit;
++feacounter;
++j;
break;
}
if ((adjlist[weakestweakeststrong] [component[0] [(i+1)]] == 1)
   && (adjlist[matlist[strongestweakeststrong]] [component[0] [i]]
   == 1))
{
caseno = 2;
casetype = 12;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = weakeststrongnode[k];
    placeofconnector[k] = pws[k];
  }
splitplace = i;
++wws2chainsplit;
++feacounter;
++j;
break;
```

```
}
if ((adjlist[weakeststrongestweak] [component[0] [i]] == 1)
   && (adjlist[matlist[strongeststrongestweak]] [component[0] [(i+1)]]
   == 1))
{
caseno = 1;
casetype = 21;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = strongestweaknode[k];
    placeofconnector[k] = psw[k];
  }
splitplace = i;
++wsw1chainsplit;
++feacounter;
++j;
break;
}
if ((adjlist[weakeststrongestweak] [component[0] [(i+1)]] == 1)
   && (adjlist[matlist[strongeststrongestweak]] [component[0] [i]]
   == 1))
{
caseno = 2;
casetype = 21;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = strongestweaknode[k];
    placeofconnector[k] = psw[k];
  }
splitplace = i;
++wsw2chainsplit;
++feacounter;
++j;
break;
}
if ((adjlist[weakestweakest] [component[0] [i]] == 1)
   && (adjlist[matlist[strongestweakest]] [component[0] [(i+1)]] == 1))
{
caseno = 1;
casetype = 11;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = weakestnode[k];
    placeofconnector[k] = pww[k];
  }
splitplace = i;
++ww1chainsplit;
++feacounter;
++j;
break;
}
if ((adjlist[weakestweakest] [component[0] [(i+1)]] == 1)
   && (adjlist[matlist[strongestweakest]] [component[0] [i]] == 1))
{
caseno = 2;
casetype = 11;
for(k=1; k<=nocycles; ++k)
  {
    connector[k] = weakestnode[k];
    placeofconnector[k] = pww[k];
  }
splitplace = i;
++ww2chainsplit;
++feacounter;
++j;
break;
```

```
  }
} // End of going through chain for finding a place where to split it

if (j == 1) // case that one of the above subcases has been detected
{

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Checking result for (09b) case (Chainsplit with more than one cycle)
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Note: this is built on the caseno and connector information from above

// Step1: Initialising order
for(i = 1; i <= nocycles; ++i)
{
   ordcyc[i] = i;
}

// Step2: Sorting procedure for cycles according to strngest connectors
for(i = 1; i <= nocycles; ++i)
{
   for(j = i-1; j >= 1; --j)
   {
    //cout << i << "  " << ordsc[i] << "  "
    //     << j << "   " << ordsc[j] << endl;
    // abbreviation = ;
     if (wbox[ordsc[connector[i]]]
         < wbox[ordsc[connector[ordcyc[j]]]])
     {
       ordcyc[j+1] = ordcyc[j];
       ordcyc[j] = i;
         //cout << "after change  " << i << "  " << ordsc[i] << "  "
         //     << j << "   " << ordsc[j] << endl;
     }
     if ((wbox[ordsc[connector[i]]]         // in case of tie ...
         == wbox[ordsc[connector[ordcyc[j]]]]) &&
        (wbox[ordsc[matlist[connector[i]]]] // decide 4 node w/ higher m8
         <  wbox[ordsc[matlist[connector[ordcyc[j]]]]]))
     {
       ordcyc[j+1] = ordcyc[j];
       ordcyc[j] = i;
     }
   }
}
// Step3: Looking for the place of connector in each cycle
// because this tells us where to start building the result out of cycles
/*
for(i = 1; i <= nocycles; ++i)
{
  placeofconnector[i] = empty;
}
for(i = 1; i <= nocycles; ++i)
{
  for(j = 0; j < lengthofcomponent[i]; ++j)
  {
    if (connector[i] == component[i][j])
    {
      placeofconnector[i] = j;
      break;
    }
  }
}
*/
// Note: placeofconnector[i] has already been defined above

// Step4: Starting building result array with chain
if (caseno == 1) // starting from beginning of chain
```

```
{
  for(i=0; i<=splitplace; ++i)
    result[i] = component[0] [i];
  currentplace = splitplace + 1;
}
if (caseno == 2)
{
  for(i=0; i<=(lengthofcomponent[0]-splitplace-2); ++i)
    result[i] = component[0] [(lengthofcomponent[0]-1-i)];
  currentplace = lengthofcomponent[0]-splitplace-2 + 1;
}

// Step5: Now add the cycles to the result
for(k = 1; k <= nocycles; ++k)
{
  // first option: twin comes after strongest node in this cycle
  if (twin[connector[ordcyc[k]]]
      == component[ordcyc[k]] [(placeofconnector[ordcyc[k]]+1)])
  {
     for(i = placeofconnector[ordcyc[k]];
         i < lengthofcomponent[ordcyc[k]]; ++i)
       result[(  i
               + currentplace
               - placeofconnector[ordcyc[k]])]
       = component[ordcyc[k]] [i];
     if (placeofconnector[ordcyc[k]] != 0) // str nd is not first node
     {
       for(i=0; i<placeofconnector[ordcyc[k]]; ++i)
         result[(  i
                 + currentplace
                 + lengthofcomponent[ordcyc[k]]
                 - placeofconnector[ordcyc[k]])]
         = component[ordcyc[k]] [i];
     }
   }
   // second option: twin comes b4 strongest node in this cycle
   else
   {
     for(i=placeofconnector[ordcyc[k]]; i>=0; --i)
       result[(  currentplace
               + placeofconnector[ordcyc[k]]
               - i)]
       = component[ordcyc[k]] [i];
     if (placeofconnector[ordcyc[k]]
         != (lengthofcomponent[ordcyc[k]]-1))  // str nd is not last node
     {
        for(i = (lengthofcomponent[ordcyc[k]]-1);
            i > placeofconnector[ordcyc[k]]; --i)
          result[(  currentplace
                  + placeofconnector[ordcyc[k]]
                  + (lengthofcomponent[ordcyc[k]]-i))]
          = component[ordcyc[k]] [i];
     }
   }
   // store new position in building result array
   currentplace = currentplace + lengthofcomponent[ordcyc[k]];
} // end of loop through all cycles

 //Step6: Add second part of chain to result array
 if (caseno == 1) // running forward from splitplace+1
 {
   for(i=0; i<=(lengthofcomponent[0]-splitplace-2); ++i)
     result[(currentplace+i)] = component[0] [splitplace+1+i];
   // currentplace = ...;
 }
 if (caseno == 2)  // running backwards from splitplace
```

```
     {
       for(i=0; i<=splitplace; ++i)
         result[(currentplace+i)] = component[0] [(splitplace-i)];
       //  currentplace = ...;
     }

     ++numberof09bcases;
     status = checkresult(92);

    // +++++++++ End of building and checking result ++++++++++++++++++++++++++

    continue;
    }  // End of: a case (09b) phenomenon has been detected

  } // End of case (09b)


// ***************************************************************************
// Statistics and some more checks for remaining cases
// ***************************************************************************
  // Counting the length of the chain and no of cycles in remaining cases
  ++distofcyclesleft[nocycles];
  ++lengthchain[lengthofcomponent[0]];


  // Check in the case of one cycle
  if (nocycles == 1)
  {
    stack = 0;
    for(i=0; i<lengthofcomponent[1]; ++i)
    {
      if (wbox[ordsc[component[1] [i]]] > stack)
        stack = wbox[ordsc[component[1] [i]]];
    }
    if (stack + wbox[ordsc[unconnode[1]]] >= thradj)
      ++prob;
    //if (instance < 1000)
      //cout << "T";
  }

// ***************************************************************************
// Inverse matching algorithm
// ***************************************************************************
// Step 1 Inverse: Initialise matching list and counters
   for (i = 0; i < numsc; ++i)
   {
     matlistinv[i] = empty;
   }
   matcardinv = 0;
   unconpointerinv = 0;
   for (i = 0; i < numsc; ++i)
     {
       unconnodeinv[i] = empty;
     }
   lastmatchinv = empty;

// Step 2 Inverse: Matching algorithm
   for(i=0; i<numsc; ++i)// check all nodes
   {
     if (matlistinv[i] == empty)// does node need a mate?
     {
       for (j=(numsc-1); j>i; --j)// look for a mate for node i
       {
         if (  (adjlist[i] [j] == 1)
            && (matlistinv[j] == empty))// if mate found
         {
```

```
          matlistinv[i] = j;
          matlistinv[j] = i;
          lastmatchinv = i;
          ++matcardinv;
          break;
        }
      }
      if (matlistinv[i] == empty)   // if there still is no mate:
      {                           // do twin node swap or finally acquiesce
        if (ordsc[i] % 2 == 0)     // find out twin node number
        {
          twinnomatinv = invordsc[(ordsc[i]+1)];
        }
        else
        {
          twinnomatinv = invordsc[(ordsc[i]-1)];
        }
        if                         // twin node swap possible?
        (  (wbox[ordsc[i]]+wbox[ordsc[twinnomatinv]]>=thradj)// match w/ twin?
        && (matlistinv[twinnomatinv] == empty)               // twin unmatchd?
        && (lastmatchinv != empty)                           // exchnge pssble?
        && (twinnomatinv > i)                               // twin larger?
        && (  wbox[ordsc[lastmatchinv]]                 // lastmatch with twin?
            + wbox[ordsc[twinnomatinv]] >= thradj))
        {                          // then swap mates
          matlistinv[i] = matlistinv[lastmatchinv];
          matlistinv[lastmatchinv] = twinnomatinv;
          matlistinv[twinnomatinv] = lastmatchinv;
          matlistinv[matlistinv[i]] = i;
          lastmatchinv = i;
          ++matcardinv;
        }
        else                       // otherwise: one more unconnected node
        {
          ++unconpointerinv;
          unconnodeinv[(unconpointerinv-1)] = i;
        }
      }
    }
  }

// ****************************************************************************
// Double check inverse matching with other matching algorithm
// ****************************************************************************
  if (matcard != matcardinv)
  {
    cout << "ALARM5!!!" << endl;
    cout << matcard << "  " << matcardinv << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << wbox[i];
    cout << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << wbox[ordsc[i]];
    cout << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << matlist[i];
    cout << endl;
    for (i=0; i<numsc; ++i)
      cout << "  " << matlistinv[i];
    cout << endl;
    cout << adjlist[2] [matlist[2]] << endl;
    cout << wbox[ordsc[2]] << "  " << wbox[ordsc[matlist[2]]] << endl;
    cout << unconpointer << "  " << unconpointerinv << endl;

    //First matching algorithm once again (for checking mistakes
    // Step 2 once again: Initialise matching list and counters
```

```
for (i = 0; i < numsc; ++i)
{
  matlist[i] = empty;
}
matcard = 0;
unconpointer = 0;
for (i = 0; i < numsc; ++i)
  {
    unconnode[i] = empty;
  }
lastmatch = empty;

 // Step 3 once again: Matching algorithm
for(i=0; i<numsc; ++i)// check all nodes
{
  if (matlist[i] == empty)// does node need a mate?
  {
    for (j=(i+1); j<numsc; ++j)// look for a mate for node i
    {
      if (  (adjlist[i] [j] == 1)
         && (matlist[j] == empty))// if mate found
      {
        matlist[i] = j;
        matlist[j] = i;
        lastmatch = i;
        ++matcard;
        break;
      }
    }
    if (matlist[i] == empty)   // if there still is no mate:
    {                          // do twin node swap or finally acquiesce
      if (ordsc[i] % 2 == 0)     // find out twin node number
      {
        twinnomat = invordsc[(ordsc[i]+1)];
      }
      else
      {
        twinnomat = invordsc[(ordsc[i]-1)];
      }
      if                           // twin node swap possible?
      (  (wbox[ordsc[i]]+wbox[ordsc[twinnomat]]>=thradj)// match with twin?
      && (matlist[twinnomat] == empty)                  // twin unmatched?
      && (lastmatch != empty)                           // exchange possble?
      && (twinnomat > i))                               // twin larger?
      {                            // then swap mates
        matlist[i] = matlist[lastmatch];
        matlist[lastmatch] = twinnomat;
        matlist[twinnomat] = lastmatch;
        matlist[matlist[i]] = i;
        lastmatch = i;
        ++matcard;

        cout << "a twin node swap" << endl;
        cout << " i " << i << " twinnomat " << twinnomat
             << " ordsc[i] " << ordsc[i] << " ordsc[twinnomat] "
                                         << ordsc[twinnomat]
             << " wboxSumme " << wbox[ordsc[i]]+wbox[ordsc[twinnomat]]
             << " matlist[twinnomat] " << matlist[twinnomat]
             << " lastmatch " << lastmatch
             << " matlist[lastmatch] " << matlist[lastmatch] << endl;

        //matlist[i] = matlist[lastmatch];
        //matlist[lastmatch] = twinnomat;
        //matlist[twinnomat] = lastmatch;
        //matlist[matlist[i]] = i;
        //lastmatch = i;
```

```
          //++matcard;

        }
        else                        // otherwise: one more unconnected node
        {
          ++unconpointer;
          unconnode[(unconpointer-1)] = i;
          cout <<  " an unconnode " << i << endl;
        }
      }
    }
  }
  cout << "new results" << endl;
  cout << adjlist[2] [matlist[2]] << endl;
  cout << wbox[ordsc[2]] << "  " << wbox[ordsc[matlist[2]]] << endl;
  cout << unconpointer << "  " << unconpointerinv << endl;

// End if matcard != matcardinv
  }

// ****************************************************************************
// Building up chain from inverse matching
// ****************************************************************************

// Step 1INV: Initialise data
  for(i=0; i<numsc; ++i)
  {
    analysedinv[i] = 0;
    for(j=0; j<nocomp; ++j)
      componentinv[j] [i] = empty;
    if (ordsc[i] % 2 == 0) // find out twin node number
      twin[i] = invordsc[(ordsc[i]+1)];
    else
      twin[i] = invordsc[(ordsc[i]-1)];
    //cout << ordsc[i] << "-" << ordsc[twin[i]] << "  ";
  }
  //cout << endl;

// Step 2INV: Build up chain
  for(i=0; i<numsc; ++i)     // find smallest unconnected node
  {
    if (matlistinv[i] == empty)
    {
      smallestunconinv = i;
      break;
    }
  }
  if (smallestunconinv != unconnodeinv[0])
    cout << "ALARMinv!!!" << endl;

  j = -1;                     // build up chain
  stack = smallestunconinv;
  do
  {
    ++j;
    componentinv[0] [j] = stack;
    analysedinv[stack] = 1;
    ++j;
    componentinv[0] [j] = twin[stack];
    analysedinv[twin[stack]] = 1;
    stack = matlistinv[twin[stack]];
  }
  while (stack != empty);

  lengthofcomponentinv[0] = ++j;
```

```
// ***************************************************************************
// CASE (10): Chain from inverse matching complete with length = numsc
// ***************************************************************************
  if (lengthofcomponentinv[0] == numsc)// is chain complete?
  {
    ++completechaininv;
    ++feacounter;
    for(i=0; i<numsc; ++i)
      result[i] = componentinv[0] [i];
    i = checkresult(10);

/*
    for(i=0; i<numbox; ++i)
    {
      cout << wbox[ordsc[component[0] [(2*i)]]] << "("
           << ordsc[component[0] [(2*i)]] << ")-("
           << ordsc[component[0] [(2*i+1)]] << ")"
           << wbox[ordsc[component[0] [(2*i+1)]]] << " -- ";
    }
    cout << "*" << endl;
*/

    continue;
  }

// ***************************************************************************
// Building up cycles from inverse matching
// ***************************************************************************
  for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
    {
    if (analysedinv[i] == 0)
      {
        smallestconnotanainv = i;
        break;
      }
    }
  if (matlistinv[smallestconnotanainv] == empty)
       cout << "ARLARM2inv!!!" << endl;

  currentcomponent = 0;
  do
  {
     ++currentcomponent;      // Set component
     if (currentcomponent > nocomp-1)
       cout << "ARLARM3inv!!!" << endl;

     j = -1;                  // Build up cycle
     stack = smallestconnotanainv;
     do
     {
       ++j;
       componentinv[currentcomponent] [j] = stack;
       analysedinv[stack] = 1;
       ++j;
       componentinv[currentcomponent] [j] = twin[stack];
       analysedinv[twin[stack]] = 1;
       stack = matlistinv[twin[stack]];
     }
     while (stack != smallestconnotanainv); //= while not back 2 bgnning of cyc

     lengthofcomponentinv[currentcomponent] = ++j;

     for (i=0; i<numsc; ++i) // Find smallest connected node not analysed yet
       if (analysedinv[i] == 0)
       {
          smallestconnotanainv = i;
```

```
        break;
      };
    if (matlistinv[smallestconnotanainv] == empty)
      cout << "ARLARM2Binv!!!" << endl;
  }
  while (smallestconnotanainv != stack); // = while new not analysed node found
  nocyclesinv = currentcomponent;        // save number of cycles

// ***************************************************************************
// Analysing cycles and chain from inverse matching
// ***************************************************************************
  for (i=0; i<nocomp; ++i)  // Initialising data
  {
    strongestnodeinv[i] = empty;
    pssinv[i] = empty;
    weakestnodeinv[i] = empty;
    pwwinv[i] = empty;
    strongestweaknodeinv[i] = empty;
    pswinv[i] = empty;
    weakeststrongnodeinv[i] = empty;
    pwsinv[i] = empty;
  }

  for (i=1; i<=nocyclesinv; ++i)  // Check all cycles
  {

    // INV: Initialise for all cycles
    strongestnodeinv[i] = componentinv[i] [0];
    pssinv[i] = 0;
    weakestnodeinv[i] = componentinv[i] [0];
    pwwinv[i] = 0;
    if (wbox[ordsc[componentinv[i] [1]]] >= thrstrong)
    {
      weakeststrongnodeinv[i] = componentinv[i] [1];
      pwsinv[i] = 1;
      strongestweaknodeinv[i] = componentinv[i] [2];
      pswinv[i] = 2;
      // ATTN: not clear if weak node!!!
      // (might be repaired later if any cycle has a 'real' (= truly weak)
      // strongestweaknode)
    }
    else
    {
      weakeststrongnodeinv[i] = componentinv[i] [2];
      pwsinv[i] = 2;
      strongestweaknodeinv[i] = componentinv[i] [1];
      pswinv[i] = 1;
    }
    // try to repair if current strongestweaknode[i] is not weak
    if (wbox[ordsc[strongestweaknodeinv[i]]] >= thrstrong)
    {
      for (j=0; j<lengthofcomponentinv[i]; ++j)
      {
        if (wbox[ordsc[componentinv[i] [j]]] < thrstrong)
          {
            strongestweaknodeinv[i] = componentinv[i] [j];
            pswinv[i] = j;
          }
      }
    }
    // note1: if there is no weak node in this cycle at all, we will later,
    // at (**), set strongestweaknodeinv[i] := weakeststrongnodeinv[i]
    // note2: if there is no weak node in this cycle at all, the algorithm
    // will automatclly set weakest(weak)nodeinv[i] := weakeststrongnodeinv[i],
    // so this case is repaired automatically
```

```
// INV: Check for all elements of this component
for (j=1; j<lengthofcomponentinv[i]; ++j) // Find weakst&strongst in cycle
{
// > cases
if (wbox[ordsc[componentinv[i] [j]]]
    >= wbox[ordsc[strongestnodeinv[i]]])
  {
    strongestnodeinv[i] = componentinv[i] [j];
    pssinv[i] = j;
  }
if (wbox[ordsc[componentinv[i] [j]]]
    <= wbox[ordsc[weakestnodeinv[i]]])
  {
    weakestnodeinv[i] = componentinv[i] [j];
    pwwinv[i] = j;
  }
if ((wbox[ordsc[componentinv[i] [j]]]
     >= wbox[ordsc[strongestweaknodeinv[i]]])
     && (wbox[ordsc[componentinv[i] [j]]] < thrstrong))
  {
    strongestweaknodeinv[i] = componentinv[i] [j];
    pswinv[i] = j;
  }
if ((wbox[ordsc[componentinv[i] [j]]]
     <= wbox[ordsc[weakeststrongnodeinv[i]]])
     && (wbox[ordsc[componentinv[i] [j]]] >= thrstrong))
  {
    weakeststrongnodeinv[i] = componentinv[i] [j];
    pwsinv[i] = j;
  }
// == cases
if ((wbox[ordsc[componentinv[i] [j]]]
     == wbox[ordsc[strongestnodeinv[i]]])
 && (wbox[ordsc[matlistinv[componentinv[i] [j]]]]
     > wbox[ordsc[matlistinv[strongestnodeinv[i]]]]))
  {
    strongestnodeinv[i] = componentinv[i] [j];
    pssinv[i] = j;
  }
if ((wbox[ordsc[componentinv[i] [j]]]
     == wbox[ordsc[weakestnodeinv[i]]])
 && (wbox[ordsc[matlistinv[componentinv[i] [j]]]]
     < wbox[ordsc[matlistinv[weakestnodeinv[i]]]]))
  {
    weakestnodeinv[i] = componentinv[i] [j];
    pwwinv[i] = j;
  }
if ((wbox[ordsc[componentinv[i] [j]]]
     == wbox[ordsc[strongestweaknodeinv[i]]])
 && (wbox[ordsc[matlistinv[componentinv[i] [j]]]]
     > wbox[ordsc[matlistinv[strongestweaknodeinv[i]]]]))
  {
    strongestweaknodeinv[i] = componentinv[i] [j];
    pswinv[i] = j;
  }
if ((wbox[ordsc[componentinv[i] [j]]]
     == wbox[ordsc[weakeststrongnodeinv[i]]])
 && (wbox[ordsc[matlistinv[componentinv[i] [j]]]]
     < wbox[ordsc[matlistinv[weakeststrongnodeinv[i]]]]))
  {
    weakeststrongnodeinv[i] = componentinv[i] [j];
    pwsinv[i] = j;
  }

}
```

```
  // (**) if cycle has no weak node (see note1 above): repair
  if (wbox[ordsc[strongestweaknodeinv[i]]] >= thrstrong)
    {
      strongestweaknodeinv[i] = weakeststrongnodeinv[i];
      pswinv[i] = pwsinv[i];
    }

}

  // INV :Initialise charcteristics for all cycles
weakeststrongestinv = strongestnodeinv[1];
pwssinv = pssinv[1];
strongeststrongestinv = strongestnodeinv[1];
psssinv = pssinv[1];
weakestweakestinv = weakestnodeinv[1];
pwwwinv = pwwinv[1];
strongestweakestinv = weakestnodeinv[1];
pswwinv = pwwinv[1];
weakeststrongestweakinv = strongestweaknodeinv[1];
pwswinv = pswinv[1];
strongeststrongestweakinv = strongestweaknodeinv[1];// this one could b strng
psswinv = pswinv[1];                              // due to the 'mistake' above
weakestweakeststronginv = weakeststrongnodeinv[1];
pwwsinv = pwsinv[1];
strongestweakeststronginv = weakeststrongnodeinv[1];
pswsinv = pwsinv[1];

  // INV: Check for all cycles
for(i=1; i<=nocyclesinv; ++i)
{
  // < cases
  if (wbox[ordsc[strongestnodeinv[i]]] < wbox[ordsc[weakeststrongestinv]])
    {
      weakeststrongestinv = strongestnodeinv[i];
      pwssinv = pssinv[i];
    }
  if (wbox[ordsc[strongestnodeinv[i]]] > wbox[ordsc[strongeststrongestinv]])
    {
      strongeststrongestinv = strongestnodeinv[i];
      psssinv = pssinv[i];
    }
  if (wbox[ordsc[weakestnodeinv[i]]] < wbox[ordsc[weakestweakestinv]])
    {
      weakestweakestinv = weakestnodeinv[i];
      pwwwinv = pwwinv[i];
    }
  if (wbox[ordsc[weakestnodeinv[i]]] > wbox[ordsc[strongestweakestinv]])
    {
      strongestweakestinv = weakestnodeinv[i];
      pswwinv = pwwinv[i];
    }
  if (wbox[ordsc[strongestweaknodeinv[i]]]
      < wbox[ordsc[weakeststrongestweakinv]])
    {
      weakeststrongestweakinv = strongestweaknodeinv[i];
      pwswinv = pswinv[i];
    }
  if (wbox[ordsc[strongestweaknodeinv[i]]]
      > wbox[ordsc[strongeststrongestweakinv]])
    {
      strongeststrongestweakinv = strongestweaknodeinv[i];
      psswinv = pswinv[i];
    }
  if (wbox[ordsc[weakeststrongnodeinv[i]]]
      < wbox[ordsc[weakestweakeststronginv]])
```

```
  {
    weakestweakeststronginv = weakeststrongnodeinv[i];
    pwwsinv = pwsinv[i];
  }
if (wbox[ordsc[weakeststrongnodeinv[i]]]
    > wbox[ordsc[strongestweakeststronginv]])
  {
    strongestweakeststronginv = weakeststrongnodeinv[i];
    pswsinv = pwsinv[i];
  }

// == cases
if ((wbox[ordsc[strongestnodeinv[i]]]
     == wbox[ordsc[weakeststrongestinv]])
 && (wbox[ordsc[matlistinv[strongestnodeinv[i]]]]
     < wbox[ordsc[matlistinv[weakeststrongestinv]]]))
  {
    weakeststrongestinv = strongestnodeinv[i];
    pwssinv = pssinv[i];
  }
if ((wbox[ordsc[strongestnodeinv[i]]]
     == wbox[ordsc[strongeststrongestinv]])
 && (wbox[ordsc[matlistinv[strongestnodeinv[i]]]]
     > wbox[ordsc[matlistinv[strongeststrongestinv]]]))
  {
    strongeststrongestinv = strongestnodeinv[i];
    psssinv = pssinv[i];
  }

if ((wbox[ordsc[weakestnodeinv[i]]]
     == wbox[ordsc[weakestweakestinv]])
 && (wbox[ordsc[matlistinv[weakestnodeinv[i]]]]
     < wbox[ordsc[matlistinv[weakestweakestinv]]]))
  {
    weakestweakestinv = weakestnodeinv[i];
                                  // only strong if there were no wk node
    pwwwinv = pwwinv[i];
  }
if ((wbox[ordsc[weakestnodeinv[i]]]
     == wbox[ordsc[strongestweakestinv]])
 && (wbox[ordsc[matlistinv[weakestnodeinv[i]]]]
     > wbox[ordsc[matlistinv[strongestweakestinv]]]))
  {
    strongestweakestinv = weakestnodeinv[i];
                                     // mayb str, but ok 4 buildin solutn
    pswwinv = pwwinv[i];
  }
if ((wbox[ordsc[strongestweaknodeinv[i]]]
     == wbox[ordsc[weakeststrongestweakinv]])
 && (wbox[ordsc[matlistinv[strongestweaknodeinv[i]]]]
     < wbox[ordsc[matlistinv[weakeststrongestweakinv]]]))
  {
    weakeststrongestweakinv = strongestweaknodeinv[i];
                                               // only str s'ilyavai no wk
    pwswinv = pswinv[i];
  }
if ((wbox[ordsc[strongestweaknodeinv[i]]]
     == wbox[ordsc[strongeststrongestweakinv]])
 && (wbox[ordsc[matlistinv[strongestweaknodeinv[i]]]]
     > wbox[ordsc[matlistinv[strongeststrongestweakinv]]]))
  {
    strongeststrongestweakinv = strongestweaknodeinv[i];
                                                    // mayb strong, but ok
    psswinv = pswinv[i];
  }
if ((wbox[ordsc[weakeststrongnodeinv[i]]]
```

```
          == wbox[ordsc[weakestweakeststronginv]])
      && (wbox[ordsc[matlistinv[weakeststrongnodeinv[i]]]]
          < wbox[ordsc[matlistinv[weakestweakeststronginv]]]))
       {
         weakestweakeststronginv = weakeststrongnodeinv[i];
         pwwsinv = pwsinv[i];
       }
     if ((wbox[ordsc[weakeststrongnodeinv[i]]]
          == wbox[ordsc[strongestweakeststronginv]])
      && (wbox[ordsc[matlistinv[weakeststrongnodeinv[i]]]]
          > wbox[ordsc[matlistinv[strongestweakeststronginv]]]))
       {
         strongestweakeststronginv = weakeststrongnodeinv[i];
         pswsinv = pwsinv[i];
       }

  }

  // Counting number of cycles b4 analysis
  ++distofcyclesstartinv[nocyclesinv];

// ****************************************************************************
// CASE (11): Connection of cycles w/ chain via weakest strongest strong node
// ****************************************************************************

// Step 1INV: Make sure that unconnodeinv[1] is the higher unconnected node
  if (wbox[ordsc[unconnodeinv[0]]] > wbox[ordsc[unconnodeinv[1]]])
    cout << "ALARM4inv!!!" << endl;

// Step 2INV: Check w/ wss
  if (wbox[ordsc[weakeststrongestinv]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++wscyclesandchaininv;
  ++feacounter;

  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  // Step 2baINV: Checking result for WSS node case if there is only one cycle
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   if (nocyclesinv == 1)
   {
     /*
     for(i=0; i<lengthofcomponentinv[1]; ++i)//check where in cycle strngst node
     {
       if ((componentinv[1][i])==strongestnodeinv[1])
       {
         stackinv = i;
         break;
       }
     }
     */
     stackinv = pssinv[1];

     if (twin[strongestnodeinv[1]]==componentinv[1] [(stackinv+1)])
                                                          //twin after str nd
     {
       for(i=0; i<lengthofcomponentinv[0]; ++i)
         result[i] = componentinv[0] [i];
       for(i=stackinv; i<lengthofcomponentinv[1]; ++i)
         result[(i+lengthofcomponentinv[0]-stackinv)] = componentinv[1] [i];
       if (stackinv != 0)                          // str nd is not first node
       {
         for(i=0; i<stackinv; ++i)
           result[(i+lengthofcomponentinv[0]+lengthofcomponentinv[1]-stackinv)]
           = componentinv[1] [i];
```

```
    }
  }
  else   // twin before strongest node
  {
    for(i=0; i<lengthofcomponentinv[0]; ++i)
      result[i] = componentinv[0] [i];
    for(i=stackinv; i>=0; --i)
      result[(lengthofcomponentinv[0]+stackinv-i)] = componentinv[1] [i];
    if (stackinv != (lengthofcomponentinv[1]-1))  // str nd is not last node
    {
      for(i=(lengthofcomponentinv[1]-1); i>stackinv; --i)
        result[(lengthofcomponentinv[0]
        + stackinv
        + (lengthofcomponentinv[1]-i))]
        = componentinv[1] [i];
    }
  }
}
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Step 2bbINV: Checking result for WSS case if there is more than one cycle
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

else                                          // There is more than one cycle
{
  // Initialising order
  for(i = 1; i <= nocyclesinv; ++i)
  {
     ordcycinv[i] = i;
  }
  // Starting sorting procedure for cycles according to stronges strong nodes
  for(i = 1; i <= nocyclesinv; ++i)
  {
     for(j = i-1; j >= 1; --j)
     {
      //cout << i << "  " << ordsc[i] << "  "
      //     << j << "   " << ordsc[j] << endl;
      // abbreviation = ;
       if (wbox[ordsc[strongestnodeinv[i]]]
           < wbox[ordsc[strongestnodeinv[ordcycinv[j]]]])
       {
         ordcycinv[j+1] = ordcycinv[j];
         ordcycinv[j] = i;
           //cout << "after change  " << i << "  " << ordsc[i] << "  "
           //     << j << "   " << ordsc[j] << endl;
       }
         //else { cout << "no change, next i" << endl; break; }
     }
  }
  // Looking for the place of strongest node in each cycle
  // because this tells us where to start building the result out of cycles
  /*
  for(i = 1; i <= nocyclesinv; ++i)
  {
    placeofstrongestnodeinv[i] = empty;
  }
  for(i = 1; i <= nocyclesinv; ++i)
  {
    for(j = 0; j < lengthofcomponentinv[i]; ++j)
    {
      if (strongestnodeinv[i] == componentinv[i][j])
      {
        placeofstrongestnodeinv[i] = j;
        break;
      }
    }
  }
```

```
    */

    // Starting result array with chain
    for(i=0; i<lengthofcomponentinv[0]; ++i)
        result[i] = componentinv[0] [i];
    currentplaceinv = lengthofcomponentinv[0];
    // Now add the cycles to the result
    for(k = 1; k <= nocyclesinv; ++k)
    {
      // first option: twin comes after strongest node in this cycle
      if (twin[strongestnodeinv[ordcycinv[k]]]
          == componentinv[ordcycinv[k]]
                         [(pssinv[ordcycinv[k]]+1)])
      {
         for(i = pssinv[ordcycinv[k]];
             i < lengthofcomponentinv[ordcycinv[k]]; ++i)
           result[(  i
                   + currentplaceinv
                   - pssinv[ordcycinv[k]])]
           = componentinv[ordcycinv[k]] [i];
         if (pssinv[ordcycinv[k]] != 0)
                                                 // str nd is not first node
         {
           for(i=0; i<pssinv[ordcycinv[k]]; ++i)
             result[(  i
                     + currentplaceinv
                     + lengthofcomponentinv[ordcycinv[k]]
                     - pssinv[ordcycinv[k]])]
            = componentinv[ordcycinv[k]] [i];
         }
       }
       // second option: twin comes b4 strongest node in this cycle
       else
       {
         for(i=pssinv[ordcycinv[k]]; i>=0; --i)
           result[(  currentplaceinv
                   + pssinv[ordcycinv[k]]
                   - i)]
           = componentinv[ordcycinv[k]] [i];
         if (pssinv[ordcycinv[k]]
             != (lengthofcomponentinv[ordcycinv[k]]-1))
                                                  // str nd is not last node
         {
            for(i = (lengthofcomponentinv[ordcycinv[k]]-1);
                i > pssinv[ordcycinv[k]]; --i)
              result[(  currentplaceinv
                      + pssinv[ordcycinv[k]]
                      + (lengthofcomponentinv[ordcycinv[k]]-i))]
             = componentinv[ordcycinv[k]] [i];
         }
       }
       // store new position in building result array
       currentplaceinv = currentplaceinv + lengthofcomponentinv[ordcycinv[k]];
     } // end of loop through all cycles
   ++numberofelsecasesinv;
   } // end of else for case of more than one cycle

   status = checkresult(11);
  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


  continue;
  }

// Step3INV: Check w/ mate of sss, www, and mate of sww from invrse match
  if (wbox[ordsc[matlistinv[strongeststrongestinv]]]
```

```
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++msscyclesandchaininv;
  ++feacounter;
  continue;
  }
  if (wbox[ordsc[weakestweakestinv]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++wwcyclesandchaininv;
  ++feacounter;
  continue;
  }
  if (wbox[ordsc[matlistinv[strongestweakestinv]]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++mswcyclesandchaininv;
  ++feacounter;
  continue;
  }

// Step 4INV: Check w/ wws, mate of sws, wsw, and mate of ssw from inv match
  if (wbox[ordsc[weakestweakeststronginv]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++wwscyclesandchaininv;
  ++feacounter;
  continue;
  }
  if (wbox[ordsc[matlistinv[strongestweakeststronginv]]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++mswscyclesandchaininv;
  ++feacounter;
  continue;
  }
  if (wbox[ordsc[weakeststrongestweakinv]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++wswcyclesandchaininv;
  ++feacounter;
  continue;
  }
  if (wbox[ordsc[matlistinv[strongeststrongestweakinv]]]
      + wbox[ordsc[unconnodeinv[1]]] >= thradj)
  {
  ++msswcyclesandchaininv;
  ++feacounter;
  continue;
  }

// ***************************************************************************
// CASE (12a): Splitting up INV chain to connect to cyle if nocycles == 1
// ***************************************************************************

  if ((nocyclesinv == 1) && (lengthofcomponentinv[0] >= 4))
  {
    k = 0; // counter if a subcase has been detected
    for (i=1; i<=(lengthofcomponentinv[0]-3); i=i+2)
    {
      for (j=1; j<=(lengthofcomponentinv[1]-1); j=j+2)
      {
        if ((adjlist[componentinv[1] [j]] [componentinv[0] [i]] == 1)
           && (adjlist[matlistinv[componentinv[1] [j]]]
                      [componentinv[0] [(i+1)]]         == 1))
           {
```

```
        caseno = 1;
        stackcycinv = j;
        stackchaininv = i;
        ++cyc1chainsplitinv;
        ++feacounter;
        ++k;
        break;
        }
     if ((adjlist[componentinv[1] [j]] [componentinv[0] [i+1]] == 1)
        && (adjlist[matlistinv[componentinv[1] [j]]]
                   [componentinv[0] [i]]             ==1 ))
        {
        caseno = 2;
        stackcycinv = j;
        stackchaininv = i;
        ++cyc2chainsplitinv;
        ++feacounter;
        ++k;
        break;
        }
    }
    if (k == 1) break;
 }
 if (k == 1)
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Start check for chainsplit with one cycle
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 {
  // relevant component of cyc in var "stackcyc", "stackchain" similar
  // caseno = 1 or 2 signifies the case above
  // First: case 1 && stackchain is not last node of cycle
  if ((caseno == 1) && (stackcycinv != (lengthofcomponentinv[1] - 1)))
  {
    for (i=0; i<=lengthofcomponentinv[0]-stackchaininv-2; ++i)// upper prt  chain
      result[i] = componentinv[0] [lengthofcomponentinv[0]-1-i];

    for (i=0; i<=lengthofcomponentinv[1]-stackcycinv-2; ++i)// upper prt of cycle
      result[i+lengthofcomponentinv[0]-stackchaininv-1]
      = componentinv[1] [stackcycinv+1+i];

    for (i=0; i<=stackcycinv; ++i)                      // lower part of cycle
      result[i+lengthofcomponentinv[1]-stackcycinv-2
             +lengthofcomponentinv[0]-stackchaininv] = componentinv[1] [i];

    for (i=0; i<=stackchaininv; ++i)                    // lower part of chain
      result[i+lengthofcomponentinv[1]-1+lengthofcomponentinv[0]-stackchaininv]
      = componentinv[0] [stackchaininv-i];
  }
  // Second: case 1 && stackchain is last node of cycle
  if ((caseno == 1) && (stackcycinv == (lengthofcomponentinv[1] - 1)))
  {
    for (i=0; i<=stackchaininv; ++i)
      result[i] = componentinv[0] [i];               // lower part of chain
    for (i=0; i<=(lengthofcomponentinv[1] - 1); ++i)   // whole cycle backwards
      result[i+stackchaininv+1] = componentinv[1] [lengthofcomponentinv[1]-1-i];
    for (i=0; i<=(lengthofcomponentinv[0]-2-stackchaininv); ++i) // upper chain
      result[i+stackchaininv+1+lengthofcomponentinv[1]]
      = componentinv[0] [i+stackchaininv+1];
  }
  // Third: case 2 && stackchain is not last node of cycle
  if ((caseno == 2) && (stackcycinv != (lengthofcomponentinv[1] - 1)))
  {
    for (i=0; i<=stackchaininv; ++i)              // lower part of chain
      result[i] = componentinv[0] [i];
    for (i=0; i <= (lengthofcomponentinv[1] - 2 - stackcycinv); ++i) // upper cyc
      result[i+1+stackchaininv] = componentinv[1] [i+stackcycinv+1];
```

```
       for (i=0; i<=stackcycinv; ++i)                    // lower cyc
         result[i+stackchaininv+lengthofcomponentinv[1]-stackcycinv] =
componentinv[1] [i];
       for (i=0; i<=(lengthofcomponentinv[0] - 2 - stackchaininv); ++i) // upper chn
         result[i+1+stackchaininv+lengthofcomponentinv[1]]
         = componentinv[0] [i+1+stackchaininv];
     }
     // Fourth: case 2 && stackchain is last node of cycle
     if ((caseno == 2) && (stackcycinv == (lengthofcomponentinv[1] - 1)))
     {
       for(i=0; i<=stackchaininv; ++i)     // lower part of chain
         result[i] = componentinv[0] [i];
       for(i=0; i<=(lengthofcomponentinv[1]-1); ++i)  // cycle forward
         result[i+1+stackchaininv] = componentinv[1] [i];
       for(i=0; i<=(lengthofcomponentinv[0]-stackchaininv-2); ++i) // upper chain
         result[i+1+stackchaininv+lengthofcomponentinv[1]]
         = componentinv[0] [stackchaininv+1+i];
     }

     status = checkresult(121);
     // ++++++++++++++++++++++++++++++++++++++++
     // End of check Chainsplit with one cycle
     // ++++++++++++++++++++++++++++++++++++++++
     continue;
   }

 } // End of case (12a)

// ***************************************************************************
// CASE (12b): Splitting up INV chain to connect to ends of cycles, 8 subcases
// ***************************************************************************
  if ((nocyclesinv > 1) && (lengthofcomponentinv[0] >= 4))
  {
    j = 0; // counter if a subcase has been detected
    for (i=1; i<=(lengthofcomponentinv[0]-3); i=i+2)
    {
      if ((adjlist[weakeststrongestinv] [componentinv[0] [i]] == 1)
       && (adjlist[matlistinv[strongeststrongestinv]] [componentinv[0] [(i+1)]]
       == 1))
      {
      caseno = 1;
      casetype = 22;
      for(k=1; k<=nocyclesinv; ++k)
        {
          connectorinv[k] = strongestnodeinv[k];
          placeofconnectorinv[k] = pssinv[k];
        }
      splitplaceinv = i;
      ++ws1chainsplitinv;
      ++feacounter;
      ++j;
      break;
      }
      if ((adjlist[weakeststrongestinv] [componentinv[0] [(i+1)]] == 1)
       && (adjlist[matlistinv[strongeststrongestinv]] [componentinv[0] [i]]
       == 1))
      {
      caseno = 2;
      casetype = 22;
      for(k=1; k<=nocyclesinv; ++k)
        {
          connectorinv[k] = strongestnodeinv[k];
          placeofconnectorinv[k] = pssinv[k];
        }
      splitplaceinv = i;
      ++ws2chainsplitinv;
```

```
    ++feacounter;
    ++j;
    break;
    }

    if ((adjlist[weakestweakeststronginv] [componentinv[0] [i]] == 1)
  && (adjlist[matlistinv[strongestweakeststronginv]] [componentinv[0] [(i+1)]]
      == 1))
    {
    caseno = 1;
    casetype = 12;
    for(k=1; k<=nocyclesinv; ++k)
      {
        connectorinv[k] = weakeststrongnodeinv[k];
        placeofconnectorinv[k] = pwsinv[k];
      }
    splitplaceinv = i;
    ++wws1chainsplitinv;
    ++feacounter;
    ++j;
    break;
    }
    if ((adjlist[weakestweakeststronginv] [componentinv[0] [(i+1)]] == 1)
     && (adjlist[matlistinv[strongestweakeststronginv]] [componentinv[0] [i]]
      == 1))
    {
    caseno = 2;
    casetype = 12;
    for(k=1; k<=nocyclesinv; ++k)
      {
        connectorinv[k] = weakeststrongnodeinv[k];
        placeofconnectorinv[k] = pwsinv[k];
      }
    splitplaceinv = i;
    ++wws2chainsplitinv;
    ++feacounter;
    ++j;
    break;
    }
    if ((adjlist[weakeststrongestweakinv] [componentinv[0] [i]] == 1)
 && (adjlist[matlistinv[strongeststrongestweakinv]] [componentinv[0] [(i+1)]]
      == 1))
    {
    caseno = 1;
    casetype = 21;
    for(k=1; k<=nocyclesinv; ++k)
      {
        connectorinv[k] = strongestweaknodeinv[k];
        placeofconnectorinv[k] = pswinv[k];
      }
    splitplaceinv = i;
    ++wsw1chainsplitinv;
    ++feacounter;
    ++j;
    break;
    }
    if ((adjlist[weakeststrongestweakinv] [componentinv[0] [(i+1)]] == 1)
     && (adjlist[matlistinv[strongeststrongestweakinv]] [componentinv[0] [i]]
      == 1))
    {
    caseno = 2;
    casetype = 21;
    for(k=1; k<=nocyclesinv; ++k)
      {
        connectorinv[k] = strongestweaknodeinv[k];
        placeofconnectorinv[k] = pswinv[k];
```

```
    }
  splitplaceinv = i;
  ++wsw2chainsplitinv;
  ++feacounter;
  ++j;
  break;
  }
  if ((adjlist[weakestweakestinv] [componentinv[0] [i]] == 1)
   && (adjlist[matlistinv[strongestweakestinv]] [componentinv[0] [(i+1)]]
   == 1))
  {
  caseno = 1;
  casetype = 11;
  for(k=1; k<=nocyclesinv; ++k)
    {
      connectorinv[k] = weakestnodeinv[k];
      placeofconnectorinv[k] = pwwinv[k];
    }
  splitplaceinv = i;
  ++ww1chainsplitinv;
  ++feacounter;
  ++j;
  break;
  }
  if ((adjlist[weakestweakestinv] [componentinv[0] [(i+1)]] == 1)
   && (adjlist[matlistinv[strongestweakestinv]] [componentinv[0] [i]]
   == 1))
  {
  caseno = 2;
  casetype = 11;
  for(k=1; k<=nocyclesinv; ++k)
    {
      connectorinv[k] = weakestnodeinv[k];
      placeofconnectorinv[k] = pwwinv[k];
    }
  splitplaceinv = i;
  ++ww2chainsplitinv;
  ++feacounter;
  ++j;
  break;
  }

}
if (j == 1) // case that one of the above subcases has been detected
{

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Checking result for (12b) case (Chainsplit with more than one cycle)
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Note: this is built on the caseno and connector information from above

// Step1: Initialising order and other
for(i = 1; i <= nocyclesinv; ++i)
{
   ordcycinv[i] = i;
}

// Step2: Sorting procedure for cycles according to strngest connectors
for(i = 1; i <= nocyclesinv; ++i)
{
   for(j = i-1; j >= 1; --j)
   {
    //cout << i << "  " << ordsc[i] << "  "
    //     << j << "   " << ordsc[j] << endl;
    // abbreviation = ;
     if (wbox[ordsc[connectorinv[i]]]
```

```
         < wbox[ordsc[connectorinv[ordcycinv[j]]]])
      {
        ordcycinv[j+1] = ordcycinv[j];
        ordcycinv[j] = i;
          //cout << "after change  " << i << "  " << ordsc[i] << "  "
          //     << j << "    " << ordsc[j] << endl;
      }
      if ((wbox[ordsc[connectorinv[i]]]         // in case of tie ...
          == wbox[ordsc[connectorinv[ordcycinv[j]]]]) &&
         (wbox[ordsc[matlistinv[connectorinv[i]]]] // dcde 4 nd w/ higher m8
          <  wbox[ordsc[matlistinv[connectorinv[ordcycinv[j]]]]]))
      {
        ordcycinv[j+1] = ordcycinv[j];
        ordcycinv[j] = i;
      }
     }
}
// Step3: Looking for the place of connector in each cycle
// because this tells us where to start building the result out of cycles

/*for(i = 1; i <= nocyclesinv; ++i)
{
  placeofconnectorinv[i] = empty;
}
for(i = 1; i <= nocyclesinv; ++i)
{
  for(j = 0; j < lengthofcomponentinv[i]; ++j)
  {
    if (connectorinv[i] == componentinv[i][j])
    {
      placeofconnectorinv[i] = j;
      break;
    }
  }
}
*/ // already done above


// Step4: Starting building result array with chain
if (caseno == 1) // starting from beginning of chain
{
  for(i=0; i<=splitplaceinv; ++i)
    result[i] = componentinv[0] [i];
  currentplaceinv = splitplaceinv + 1;
}
if (caseno == 2)
{
  for(i=0; i<=(lengthofcomponentinv[0]-splitplaceinv-2); ++i)
    result[i] = componentinv[0] [(lengthofcomponentinv[0]-1-i)];
  currentplaceinv = lengthofcomponentinv[0]-splitplaceinv-2 + 1;
}

// Step5: Now add the cycles to the result
for(k = 1; k <= nocyclesinv; ++k)
{
  // first option: twin comes after strongest node in this cycle
  if (twin[connectorinv[ordcycinv[k]]]
      == componentinv[ordcycinv[k]] [(placeofconnectorinv[ordcycinv[k]]+1)])
  {
     // cout << "twin after strong" << endl;
     for(i = placeofconnectorinv[ordcycinv[k]];
         i < lengthofcomponentinv[ordcycinv[k]]; ++i)
       result[(  i
               + currentplaceinv
               - placeofconnectorinv[ordcycinv[k]])]
       = componentinv[ordcycinv[k]] [i];
```

```
        if (placeofconnectorinv[ordcycinv[k]] != 0) // str nd is not first node
        {
          for(i=0; i<placeofconnectorinv[ordcycinv[k]]; ++i)
            result[(  i
                    + currentplaceinv
                    + lengthofcomponentinv[ordcycinv[k]]
                    - placeofconnectorinv[ordcycinv[k]])]
             = componentinv[ordcycinv[k]] [i];
        }
      }
      // second option: twin comes b4 strongest node in this cycle
      else
      {
        for(i=placeofconnectorinv[ordcycinv[k]]; i>=0; --i)
          result[(  currentplaceinv
                  + placeofconnectorinv[ordcycinv[k]]
                  - i)]
           = componentinv[ordcycinv[k]] [i];
        if (placeofconnectorinv[ordcycinv[k]]
            != (lengthofcomponentinv[ordcycinv[k]]-1))  // str nd is not last node
        {
           for(i = (lengthofcomponentinv[ordcycinv[k]]-1);
               i > placeofconnectorinv[ordcycinv[k]]; --i)
            result[(  currentplaceinv
                    + placeofconnectorinv[ordcycinv[k]]
                    + (lengthofcomponentinv[ordcycinv[k]]-i))]
             = componentinv[ordcycinv[k]] [i];
        }
      }
      // store new position in building result array
      currentplaceinv = currentplaceinv + lengthofcomponentinv[ordcycinv[k]];
    } // end of loop through all cycles

     //Step6: Add second part of chain to result array
     if (caseno == 1) // running forward from splitplace+1
     {
       for(i=0; i<=(lengthofcomponentinv[0]-splitplaceinv-2); ++i)
         result[(currentplaceinv+i)] = componentinv[0] [splitplaceinv+1+i];
       // currentplace = ...;
     }
     if (caseno == 2)  // running backwards from splitplace
     {
       for(i=0; i<=splitplaceinv; ++i)
         result[(currentplaceinv+i)] = componentinv[0] [(splitplaceinv-i)];
       //  currentplace = ...;
     }

     ++numberof12bcases;
     status = checkresult(122);

    // +++++++++ End of building and checking result ++++++++++++++++++++++++++
    continue;
    }  // End of: a case (12b) phenomenon has been detected

  } // End of case (12b)

// ****************************************************************************
// Statistics and some more checks for remaining cases after inverse match
// ****************************************************************************
  // Counting the number of cycles and the length of the chain
  ++distofcyclesleftinv[nocyclesinv];
  ++lengthchaininv[lengthofcomponentinv[0]];

  // Check in the case of one cycle
  if (nocyclesinv == 1)
  {
```

```
    stack = 0;
    for(i=0; i<lengthofcomponentinv[1]; ++i)
    {
      if (wbox[ordsc[componentinv[1] [i]]] > stack)
        stack = wbox[ordsc[componentinv[1] [i]]];
    }
    if (stack + wbox[ordsc[unconnodeinv[1]]] >= thradj)
      ++probinv;
    //if (instance < 1000)
      //cout << "T";
  }

// ***************************************************************************
// End of instances loop
// ***************************************************************************
/*
// Output scores, ordered scores and order numbers
  for(i = 1; i <= numsc; ++i)
  {
    cout << wbox[i] << "  ";
  }
  cout << endl;
  for(i = 1; i <= numsc; ++i)
  {
    cout << wbox[ordsc[i]] << "  ";
  }
  cout << endl;
  for(i = 1; i <= numsc; ++i)
  {
    cout << ordsc[i] << "  ";
  }
  cout << endl;
  cin.get();
*/

// End of instances loop
}

// Running time
   runtime1 = (double) (clock() / CLOCKS_PER_SEC);
   // runtime2 = (double) (clock() - (numinst*time4rand)) / CLOCKS_PER_SEC;

// ***************************************************************************
// Output of final statisctis
// ***************************************************************************
   // Main statistics
   cout << "Instances: " << numinst << "  Fea: " << feacounter
                                    << "  Inf: " << infcounter << endl;
   cout << "Instances" << endl
        << "- (01)  with too many weak nodes: " << toomanyweak << endl
        << "- (02)  with non-con twins:  " << noncontwin << endl
        << "- (03)  with too many unconnectable nodes:  " << uncon << endl
        << "- (04)  with poor matching: " << poormat << endl
        << "- (05)  with perfect matching: " << perfmat
        << ", CHECK: " << checkcasecounter[5] << endl
        << "- (06)  with sufficient matching: " << suffmat
        << ", CHECK: " << checkcasecounter[6] << endl
        << "------  out of which w/ connectable weak node: " << yeahcounter
                                                             << endl
        << "- (07)  with complete chain built: " << completechain
        << ", CHECK: " << checkcasecounter[7] << endl
        << "- (08)  with attaching wss node in cycles to chain: "
                                               << wscyclesandchain
        << ", CHECK: " << checkcasecounter[8] << endl;

   if ((msscyclesandchain + wwcyclesandchain + mswcyclesandchain
```

```
     + wwscyclesandchain + mswscyclesandchain + wswcyclesandchain
     + msswcyclesandchain) > 0)
    cout
      << "- mate of str str in cycles + chain: " << msscyclesandchain << endl
      << "- weakest weakest in cycles + chain: " << wwcyclesandchain << endl
      << "- mate of str wk in cycles + chain: " << mswcyclesandchain << endl
      << "- wkst wkst strong in cycles + chain: " << wwscyclesandchain << endl
      << "- m8 of strst wkst str in cyc+chn: " << mswscyclesandchain << endl
      << "- wkst strst wk in cycles + chain: " << wswcyclesandchain << endl
      << "- m8 of strst strst wk in cyc+chn: " << msswcyclesandchain << endl;

cout << "- (09a) with splitting chain to attach to one cycle: "
                            << (cyc1chainsplit + cyc2chainsplit)
     << ", CHECK: " << checkcasecounter[91] << endl;
cout << "- (09b) with splitting chain and attaching to cycles: "
                                                 << numberof09bcases
     << ", CHECK: " << checkcasecounter[92] << endl;
cout << "------- weakest strongest chainsplit 1: " << ws1chainsplit
                                                     << endl
     << "------- weakest strongest chainsplit 2: " << ws2chainsplit
                                                      << endl

     << "------- weakest weakest strong chainsplit 1: " << wws1chainsplit
                                                          << endl
     << "------- weakest weakest strong chainsplit 2: " << wws2chainsplit
                                                          << endl
     << "------- weakest strongest weak chainsplit 1: " << wsw1chainsplit
                                                          << endl
     << "------- weakest strongest weak chainsplit 2: " << wsw2chainsplit
                                                          << endl
     << "------- weakest weakest chainsplit 1: "   << ww1chainsplit
                                                          << endl
     << "------- weakest weakest chainsplit 2: "   << ww2chainsplit
                                                          << endl

     << "- (10)  with INV complete chain built: " << completechaininv
     << ", CHECK : " << checkcasecounter[10] << endl
     << "- (11)  with INV attaching wss node in cycles to chain: "
                                          << wscyclesandchaininv
     << ", CHECK: " << checkcasecounter[11] << endl;

if ((msscyclesandchaininv + wwcyclesandchaininv + mswcyclesandchaininv
    + wwscyclesandchaininv + mswscyclesandchaininv + wswcyclesandchaininv
    + msswcyclesandchaininv) > 0)
    cout
      << "- INV mate of str str in cycles + chain: " << msscyclesandchaininv
                                                      << endl
      << "- INV weakest weakest in cycles + chain: " << wwcyclesandchaininv
                                                      << endl
      << "- INV mate of str wk in cycles + chain: " << mswcyclesandchaininv
                                                     << endl
      << "- INV wkst wkst strong in cycles + chain: " << wwscyclesandchaininv
                                                       << endl
      << "- INV m8 of strst wkst str in cyc+chn: " << mswscyclesandchaininv
                                                    << endl
      << "- INV wkst strst wk in cycles + chain: " << wswcyclesandchaininv
                                                    << endl
      << "- INV m8 of strst strst wk in cyc+chn: " << msswcyclesandchaininv
                                                    << endl;

cout << "- (12a) with INV splitting chain to attach to one cycle: "
                    << (cyc1chainsplitinv + cyc2chainsplitinv)
     << ", CHECK: " << checkcasecounter[121] << endl;
cout << "- (12b) with INV splitting chain and attaching to cycles: "
                                          << numberof12bcases
     << ", CHECK: " << checkcasecounter[122] << endl
```

```
        << "------- INV wkst strongest chainsplit 1: " << ws1chainsplitinv
                                                          << endl
        << "------- INV wkst strongest chainsplit 2: " << ws2chainsplitinv
                                                          << endl

        << "------- INV wkst weakest strong chnsplt 1: "
                                                  << wws1chainsplitinv << endl
        << "------- INV wkst weakest strong chnsplt 2: "
                                                  << wws2chainsplitinv << endl
        << "------- INV wkst strongest weak chnsplt 1: "
                                                  << wsw1chainsplitinv << endl
        << "------- INV wkst strongest weak chnsplt 2: "
                                                  << wsw2chainsplitinv << endl
        << "------- INV wkst weakest chainsplit 1: " << ww1chainsplitinv
                                                                  << endl
        << "------- INV wkst weakest chainsplit 2: " << ww2chainsplitinv
                                                                  << endl

        << endl;

   cout << "Percentage of instances solved: "
        << (double) (feacounter+infcounter)/numinst << endl;
   cout << "Running time: " << runtime1 << " seconds" << endl;
   cout << "Number of instances checked: " << resultcounter
        << " Failed checks among these: " << problemcounter;
   cout << endl << endl << endl;

   // Other statistics
   cout << " Number of elsecases: " << numberofelsecases;
   cout << " Number of elsecasesINV: " << numberofelsecasesinv;
   cout << endl << endl;

   for(i=0; i<=numsc; ++i)
     cout << lengthchain[i] << " times " << i << " scores" << endl;
   for(i=0; i<=numsc; ++i)
   {
     cout << lengthchaininv[i] << " times "
          << i << " scores in INV case" << endl;
   }
   // cout << "Running time without generation of instances: "
   //      << runtime2 << " seconds" << endl;
   for(i=1; i<nocomp; ++i)
   {
     cout << distofcyclesstart[i] << " times " << i
          << " cycles originally, afterwards "
          << distofcyclesleft[i] << " times." << endl;

   }
   for(i=1; i<nocomp; ++i)
   {
     cout << distofcyclesstartinv[i] << " times " << i
          << " cycles originally, afterwards "
          << distofcyclesleftinv[i] << " times in INV case." << endl;

   }

   cout << prob << " problematic cases" << endl;
   cout << probinv << " problematic cases in INV case" << endl;

// ***************************************************************************
// End of function main
// ***************************************************************************
  cin.get();
  return 0;
}
```

```
// ********************************************************************************
// ************************* END OF PROGRAMME *************************************
// ********************************************************************************
```