A Dual Network Exterior Point Simplex-type Algorithm for the Minimum Cost Network Flow Problem

Georgios Geranis

PhD Thesis

Supervisors:
Samaras Nikolaos (Principal)
Roumeliotis Emmanuel
Papistas Athanasios

Department of Applied Informatics
University of Macedonia
Thessaloniki

December 2012

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. This work could not have been carried out without the help of Dr. Konstantinos Paparrizos who guided me through the dissertation process as my supervisor. The whole work is dedicated to his memory.

I am also grateful to Dr Nikolaos Samaras for the great help he offered with his importand advices and suggestions. Many thanks also to Dr Emmanuel Rumeliotis and Dr Athanasios Papistas.

I also feel very thankful for the precious help that Dr Angelo Sifaleras offered to me throughout my work. Many special thanks also to PhD candidates Nikolaos Ploskas and Themistoklis Glavelis and Dr Eleni Rosiou and Dr Athanasios Baloukas for our collaboration.

Finally, I would like to express my gratitude to all my family and friends for their support all these years.

Geranis Georgios

List of publications

- G. Geranis, K. Paparrizos, A. Sifaleras, A dual exterior point simplex type algorithm for the minimum cost network flow problem, *Yugoslav Journal of Operations Research* 19(1) (2009) pp. 157-170.

- G. Geranis, K. Paparrizos, A. Sifaleras, A Dual Exterior Point Simplex Type Algorithm, *8th Balcan Conference On Operational Research*, Belgrade, Zlatibor, September 14-17 (2007)

- G. Geranis, K. Paparrizos, A. Sifaleras, Computational experience with a dual network exterior point simplex algorithm for the minimum cost network flow problem, *20th National Conference of the Hellenic Operational Research Society*, Spetses, June 19-21 (2008)

- G. Geranis, K. Paparrizos, A. Sifaleras, On a dual network exterior point simplex algorithm for the minimum cost network flow problem and its empirical behavior, *21st Conference of the European Chapter on Combinatorial Optimization (ECCO XXI)*, May 29-31, Dubrovnik, Croatia (2008) .

- G. Geranis, K. Paparrizos, A. Sifaleras, On the Computational Behavior of a Dual Network Exterior Point Simplex Algorithm for the Minimum Cost Network Flow Problem, *Proceedings of the International Network Optimization Conference (INOC 2009)*, 26-29 April, Pisa, Italy, (2009).

- G. Geranis, K. Paparrizos, A. Sifaleras, Use of Dynamic Trees for the

Implementation of the Dual Network Exterior Point Simplex Algorithm, *1st International Symposium and 10th Balcan Conference on Operational Research (BALCOR 2011)*, September 22-25, Thessaloniki, Greece (2011).

- G. Geranis, K. Paparrizos, A. Sifaleras, On a Dual Network Exterior Point Simplex Type Algorithm and its Computational Behavior, *RAIRO-Oper. Res. 46* (2012) pp. 211 - 234.

Abstract

The Minimum Cost Network Flow Problem (MCNFP) refers to a wide category of network flow problems and it is an important research area of Network Optimization. A Dual Network Exterior Point Simplex Algorithm (DNEPSA) for the MCNFP is presented here. The algorithm belongs to the special category of Exterior Point Simplex-Type algorithms. Similarly to the classical Dual Network Simplex-type Algorithm (DNSA), DNEPSA starts with a dual feasible tree-solution and after a number of iterations, it produces a solution that is both primal and dual feasible, i.e. it is optimal. However, contrary to DNSA, the algorithm does not always maintain a dual feasible solution throughout all its iterations. Instead, it produces tree-solutions that can be infeasible for the dual problem and at the same time infeasible for the primal problem. The theoretical proof of correctness and the implementation details of DNEPSA are also presented. A detailed comparative computational study of DNEPSA against DNSA on sparse and dense random problem instances is presented and it is followed by the statistical analysis of the experimental results showing the effectiveness of DNEPSA compared to DNSA in terms of cpu time and iterations. The implementation of DNEPSA by using dynamic trees is also demonstrated and the algorithm's amotized computational complexity per pivot is estimated.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 The Minimum Cost Network Flow Problem

Network Optimization is a research area that is part of the wider research area of Linear Optimization. It refers to those linear problems that can be modeled by using some special data structures like graphs and networks. One of the most important network optimization problems is the Minimum Cost Network FLow Problem, MCNFP for short. The MCNFP problem, as described in [77], [2] and [14], is the problem of finding a minimum cost flow of product units, through a number of supply nodes (sources), demand nodes (sinks) and transshipment nodes. Other common problems, such as the shortest path problem, the transportation problem, the assignment problem, etc., are special cases of the MCNFP problem, as it is described in section 1.4. The MCNFP problem appears very frequently in different sectors of technol-

ogy, like Informatics, Telecommunications, Transportation, etc. Numerous real life problems can be solved by applying network flow models, as it is demonstrated in [3] and [45].

The MCNFP problem can be easily transformed into a linear programming problem and there exist well-known general linear programming algorithms that can be applied in order to find an optimal solution. Such algorithms though, do not take advantage of the special features met in the MCNFP. Therefore, other special Simplex-type algorithms for the MCNFP problem have been developed, such as the well-known Primal Network Simplex Algorithm and the Dual Network Simplex Algorithm. There are also other non Simplex-type algorithms that can be used for solving the same problem, as presented in [67] and [30].

1.2 Elements of Graph Theory

Some introductory concepts from Graph Theory used throughout this thesis will be presented here. A *graph* G is an abstract representation of a set of objects, called *vertices* or *nodes*, where some pairs of the vertices are connected by links, called *arcs*. Let N and A be the set of nodes and arcs respectively, that consist a graph denoted as G=(N,A). The set of arcs A contains pairs of the form (i,j), showing that there exists a link from node i to node j. In a *directed graph*, the arc (i,j) is different from the arc (j,i). On the other hand, in a *non-directed graph* a pair of nodes (i,j) is considered equivalent to the pair (j,i) and is called an *edge*. Figures 1.1 and 1.2 show a directed and a non directed graph respectively. The nodes of a graph can be

numbered by using integer values from 1 to n, where n=$|N|$ is the number of nodes in the graph. The number of nodes is also called the *order* of the graph. In Figure 1.1, each arc (i,j) is represented by an arrow from node i to node j. Node i is the *tail* of the arc (i,j) and node j is the *head* of the arc. The nodes i and j connected by an arc (i,j) are said to be *adjacent* to each other or *neighbours*. The head and the tail of an arc are also called *endpoints* or *ends* of the arc.

The number of arcs m=$|A|$ in a graph G=(N,A) is the *size* of the graph. For the graph in Figure 1.1, we have n=$|N|$=6 and m=$|A|$=11, where it is N=$\{1, 2, 3, 4, 5, 6\}$ and A={(1,2), (1,4), (2,5), (3,1), (3,4), (3,6), (4,3), (4,5), (5,2), (5,6), (6,3)}. For a non directed arc, the edges are presented as simple lines that connect two nodes. The edge (i,j) connects node i with node j and it is equivalent to (j,i). In Figure 1.2 it is n=$|N|$=6 and m=$|A|$=8, where it is N=$\{1, 2, 3, 4, 5, 6\}$ and A={(1,2), (1,3), (1,4), (2,5), (3,4), (3,6), (4,5), (5,6)}. A non-directed graph can be considered as a special case of a directed graph where for each edge (i,j) we have two arcs: an arc (i,j) connecting node i to node j and an arc (j,i) connecting node j to node i.

Two or more arcs having the same tail i and the same head j, where $i \neq j$, are called *parallel arcs*. An arc which starts and ends on the same vertex is called a *loop*. In Figure 1.3, a graph that contains parallel arcs and loops is shown. There are two parallel arcs from node 2 to node 1 and two other parallel arcs from node 4 to node 5. There is also a loop on node 4. We assume here that the graphs we have are directed without any parallel arcs or loops.

A graph G'=(N',A') is a *subgraph* of a graph G=(N,A), when G' is a

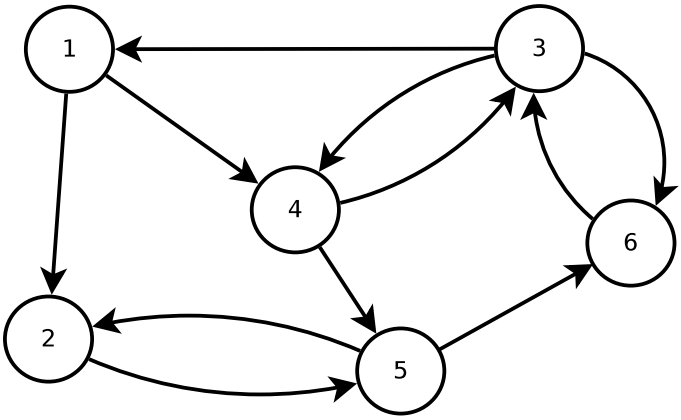

Figure 1.1: A directed graph

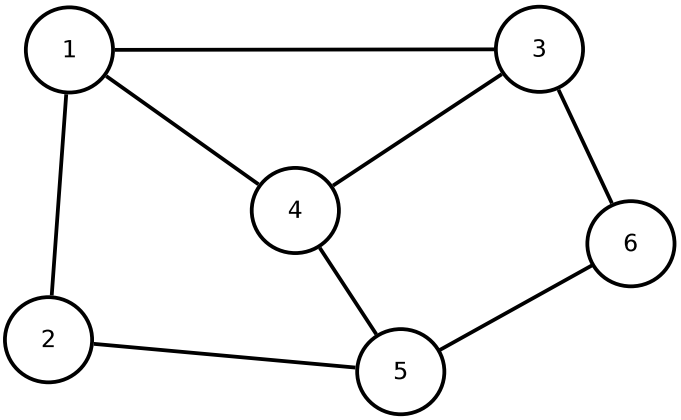

Figure 1.2: A non directed graph

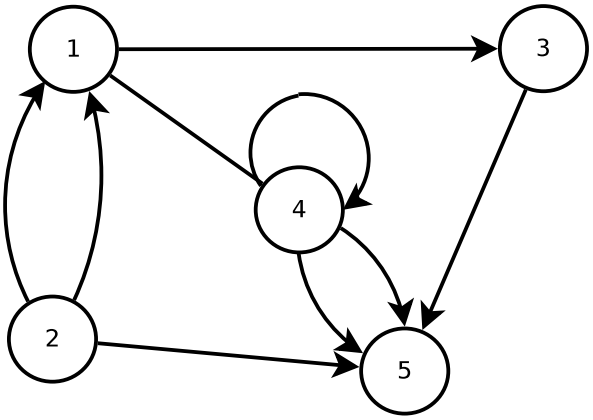

Figure 1.3: A graph containing parallel arcs and loops

graph whose vertex set is a subset of that of G ($N' \subseteq N$) and whose set of arcs connects only nodes of N' and it is a subset of that of G ($A' \subseteq A$). In the other direction, a graph G' is a *supergraph* of a graph G if G is a subgraph of G'.

A graph can be extended by assigning a weight to each arc or edge of the graph, called a *weighted graph*. A directed graph with weighted arcs is also called a *network*.

The *degree* of a node is the number of arcs or edges that connect to it. More particularly, the number of the arcs that start from a node is called the *out-degree* of the node, while the number of the arcs that end to a node is called the *in-degree* of the node. For example, in Figure 1.1, the out-degree of node 1 is equal to 2 and its in-degree equals 1. A node of degree 0 is called *isolated* and a node of degree 1 is a *leaf*.

A *chain* of a graph G is an alternating sequence of vertices and edges of the form $x_0, e_1, x_1, e_2, ..., e_k, x_k$, beginning and ending with vertices in which each edge is incident with the two vertices immediately preceding and following

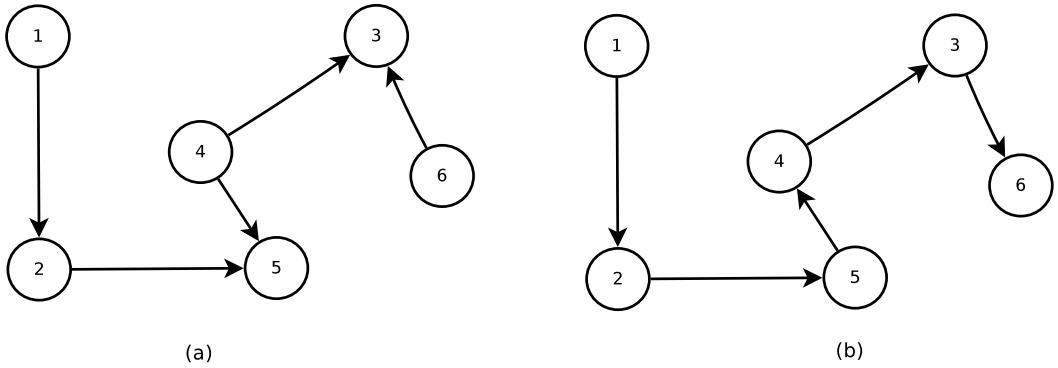

Figure 1.4: Directed and not directed paths

it. The chain above joins vertices x_0 and x_k and may also be denoted by $x_0, x_1, ..., x_k$. A chain is called *closed*, if it is $x_0 = x_k$ and it is called *open* otherwise. The *length* of a chain is the number of edges in it.

A chain is called a *path* when every vertex and every arc in the chain appears only once. An arc (i,j) in a path from vertex x_0 to vertex x_k is called a *forward* arc when on the way from vertex x_0 to vertex x_k we meet node i before we meet node j. Otherwise it is called a *backward* arc. We call *directed path* a path where all the arcs are forward arcs or all of them are backward arcs. In Figure 1.4(b) the path 1-2-5-4-3-6 is directed while in Figure 1.4(a) it is not directed. The arc (2,5) in Figure 1.4(a) is a forward arc for the path 1-2-5-4-3-6 while the arc (4,5) is a backward arc for the same path.

If a path is closed, it is called a *cycle*. If a closed path is directed then we have a *directed cycle*. A directed cycle is also called a *circuit*. A graph is *acyclic* when it contains no directed cycles. In Figure 1.5(a), we can see a non-directed cycle, while in Figure 1.5(b) we have a directed cycle. If two arcs (i,j) and (k,l) in a cycle are both forward or they are both backward, then we say they have the same orientation and we will denote it as $(i, j) \uparrow\uparrow (k, l)$.

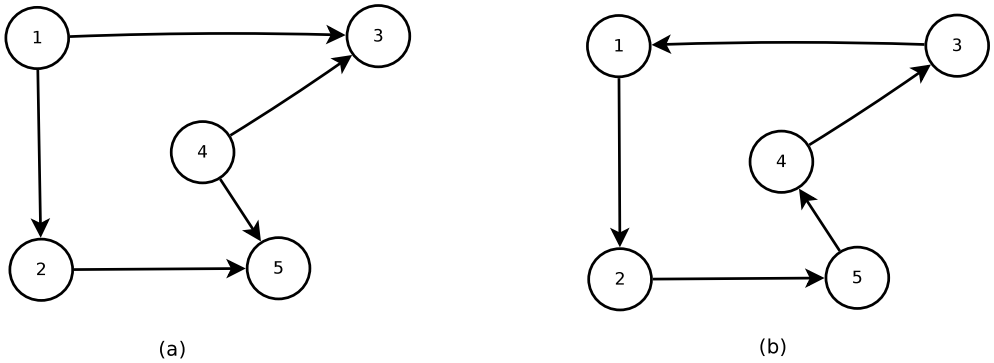

Figure 1.5: Directed and not directed cycles

Otherwise, if one of them is a forward arc and the other is a backward arc, we denote it as $(i, j) \uparrow\downarrow (k, l)$. In Figure 1.5(a), the arcs (1,2) and (4,3) have the same orientation, i.e. $(1, 2) \uparrow\uparrow (4, 3)$, while for the arcs (1,2) and (4,5) we have $(1, 2) \uparrow\downarrow (4, 5)$.

Two vertices are said to be *connected*, if there exists a path that starts from the first vertex and ends to the other. If all the vertices in a graph are connected to each other, then we have a *connected graph*. Otherwise, it is a *disconnected graph*. If for every pair of vertices (i,j) in a graph there exists a directed path from vertex i to vertex j then the graph is a *strongly connected graph*. A graph that is not connected can be divided into connected *components* which are disjoint connected subgraphs of the original graph.

There are two standard ways to represent a graph G=(N,A). The first way is the representation of the graph as a collection of *adjacency lists*. The second way is its representation as an *adjacency matrix*. Either way is applicable to both directed and undirected graphs. The adjacency list representation is very often preferred because it provides a compact way to represent sparse graphs (those for which $|A|$ is much less than $|N|^2$). However, the adjacency

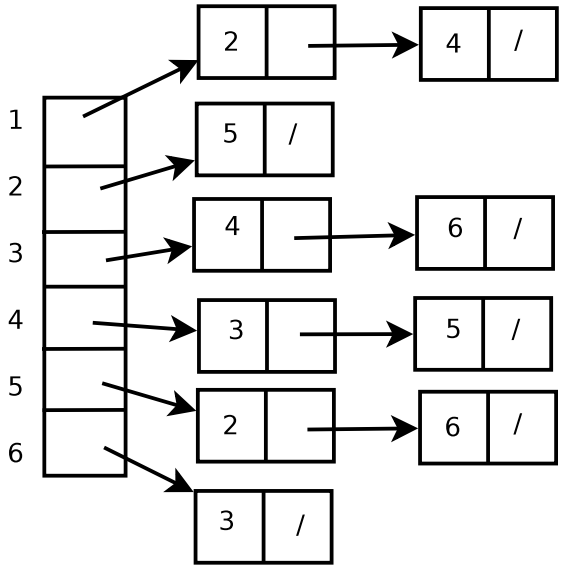

Figure 1.6: Adjacency list representation

matrix representation may be preferred in case the graph is dense ($|A|$ is close to $|N|^2$).

The adjacency list representation of a graph G=(N,A) consists of an array Adj of $|N|$ lists, one for each vertex in N. For each node $i \in N$, the adjacency list Adj[i] contains all the nodes j such that, there is an arc $(i, j) \in A$, i.e. the list Adj[i] consists of all the vertices that are adjacent to i. The vertices in each adjacency list are typically stored in an arbitrary order. Figure 1.6 shows an adjacency list representation of the directed graph in Figure 1.1. The sum of the lengths of all the adjacency lists is $|A|$ and the amount of memory required is $\Theta(|N| + |A|)$. Adjacency lists can be easily adapted to represent weighted graphs as well.

A disadvantage of the adjacency list representation is that there is no quick way to determine if a given arc (i,j) is present in the graph. In order to do that, the adjacency list Adj[i] has to be searched in order to determine if

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	1	0	1
4	0	0	1	0	1	0
5	0	1	0	0	0	1
6	0	0	1	0	0	0

Figure 1.7: Adjacency matrix representation

j is present in the list. This disadvantage can be remedied by the adjacency matrix representation of a graph. The adjacency matrix representation of a graph G=(N,A) consists of a $|V| \times |V|$ matrix A such that, it is $a_{ij} = 1$ if $(i, j) \in A$ and $a_{ij} = 0$, otherwise. Figure 1.7 shows the adjacency matrix representation of the directed graph in Figure 1.1. The adjacency matrix of a graph requires $\Theta(|N|^2)$ memory, independent of the number of arcs in the graph.

A different matrix representation for a graph is the *incidence matrix*. The incident matrix for a directed graph G=(N,A) contains one row for every node of the graph and one column for every arc of the graph. If the set of arcs A contains an arc (i,j) then, in the column that corresponds to that arc, the incident matrix contains 1 in the i-th row and -1 in the j-th row. The incident matrix for the directed graph in Figure 1.1 is shown in Figure 1.8.

A *tree* is a connected graph without cycles. A *forest* is a vertex-disjoint union of trees. A tree with n vertices contains n-1 arcs (edges) and, if $n \geq 2$, it contains at least two leaves. For every pair of vertices in a tree there exists

$$\begin{array}{c} \begin{matrix} (1,2) & (1,4) & (2,5) & (3,1) & (3,4) & (3,6) & (4,3) & (4,5) & (5,2) & (5,6) & (6,3) \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & -1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & -1 \end{pmatrix} \end{array}$$

Figure 1.8: Incident matrix representation

exactly one simple path connecting the one to the other. If we delete an arc from a given tree T then, two separate subtrees are produced. On the other hand, if we add a new arc to a given tree then, a graph is produced that contains a unique cycle. Figure 1.9 shows some examples of trees.

A node r of a tree T can be chosen to be the tree's *root*. In that case, we have a *rooted tree*. All the nodes of a rooted tree can be placed in different levels, depending on the length of the path that connects them to the root. The root of the tree belongs to level 0 of the tree and all the other nodes are placed in levels numbered as 1,2,3,... etc. The level where a node is placed is the *depth* of the node. The depth of a tree is the maximum of the depths of its nodes. A node i of a tree T is the *parent* or *immediate predecessor* of a node j of T when i belongs in level k, j belongs in level k+1 and (i,j) or (j,i) is an arc of T. In that case, j is a *child* or *immediate successor* of i. All the nodes of a rooted tree, except of the tree's root, have exactly one parent and zero or more children. The root of a tree is the only node that has no parent.

If in the tree of Figure 1.9(d), node 1 is the root, then we have the tree

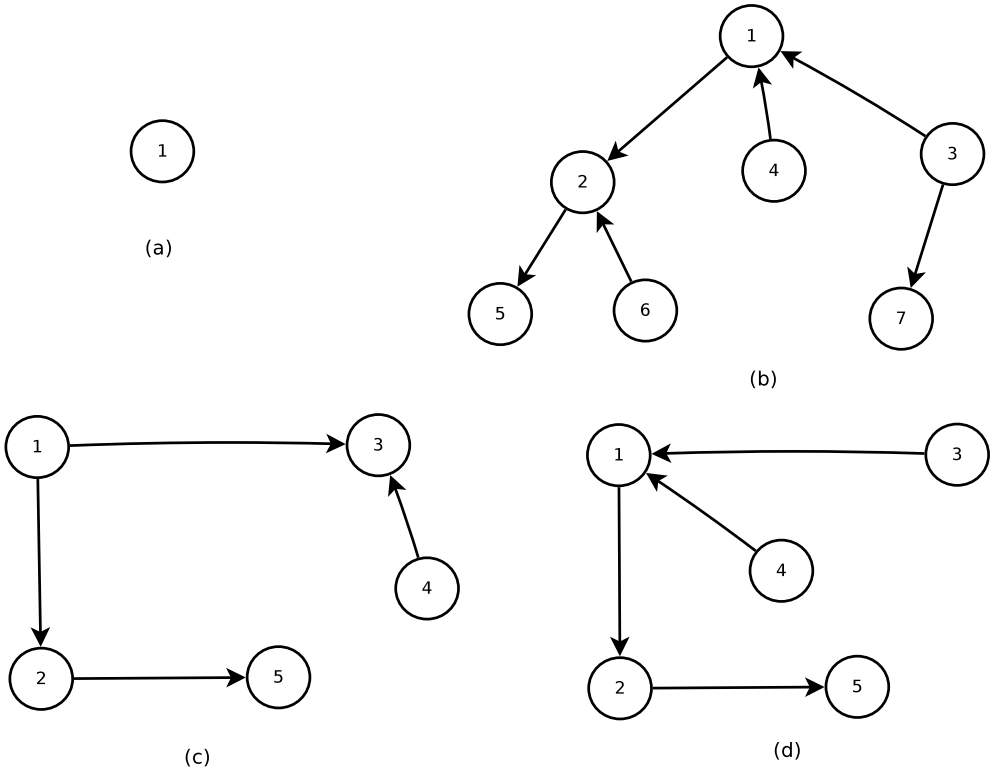

Figure 1.9: Examples of trees

levels as shown on figure 1.10(a). Nodes 2, 4 and 3 are placed in level 1 and node 5 is placed in level 2. The depth of the tree is 2. If, on the other hand, node 4 is the root then, we have the levels as shown in figure 1.10(b). In that case we take a tree of depth 3.

All the Simplex-type algorithms for the solution of the Minimum Cost Network Flow Problem (MCNFP) start from an initial tree and they try, iteration by iteration, to find out a tree that gives an optimal solution. In order to move from a tree-solution to the next tree-solution, in every iteration, the algorithm has to select a leaving and an entering arc so that a new tree is produced. The algorithm's performance, as we'll see in the following chapters, depends a lot on the data structures used to store the original

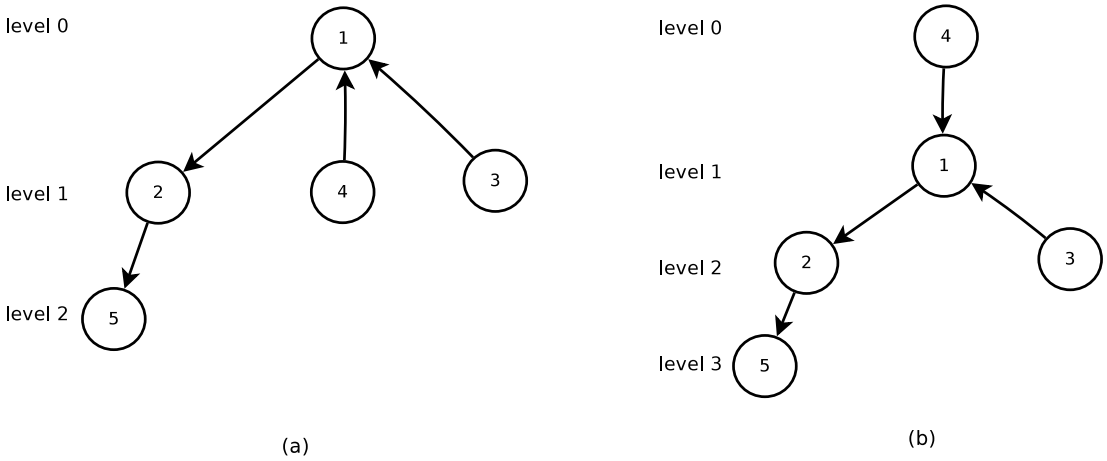

Figure 1.10: Rooted trees

network and the tree-solutions produced in every iteration.

1.3 Problem Statement and Notation

Let $G = (N, A)$ be a network that consists of a finite set of nodes N and a finite set of directed arcs A. The MCNFP problem on G is the problem of finding a flow from a set of supply nodes of G, through the arcs of G, to a set of demand nodes of G, at minimum total cost. Let n and m be the number of nodes and arcs, respectively. For each node $i \in N$, there is an associated variable b_i representing the available supply or demand on that node. If a node i is a *supply node* or *source*, then it is $b_i > 0$. On the other hand, if node i is a *demand node* or *sink*, then it is $b_i < 0$. Finally, a node i is a *transshipment node* in the case it is $b_i = 0$. The total supply in the network, has to be equal to the total demand, i.e., it is $\sum_{i \in N} b_i = 0$. Such a network is called a *balanced network*.

For every arc $(i, j) \in A$ we have a flow x_{ij} that shows the current amount

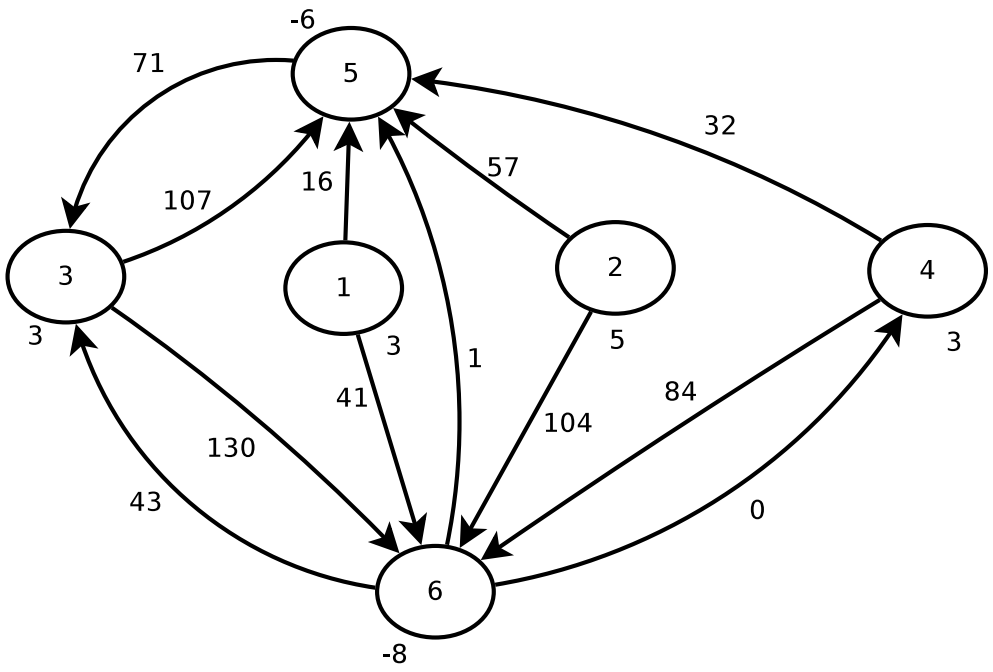

Figure 1.11: The Minimum Cost Network Flow Problem

of product units transferred from node i to node j. The value of x_{ij} can be zero or even negative. In case it is $x_{ij} < 0$ it is like having a flow from node j to node i instead of having a flow from node i to node j. For every arc $(i, j) \in A$, there is also an associated value $c_{ij} \geq 0$ that represents the cost when one unit of product flows from node i to node j. In Figure 1.11 a network of 6 nodes is given. For every node of the network, the associated supply appears next to the node. For example, node 1 is a source node that supplies 3 product units and node 5 is a sink node that demands 6 product units. Next to each arc appears the cost per product unit for the flow on that arc. The total supply, offered by nodes 1, 2, 3 and 4, is equal to 14 and this sum is the same as the total demand, needed by nodes 5 and 6.

The total cost for a flow through the network is called the *objective value*

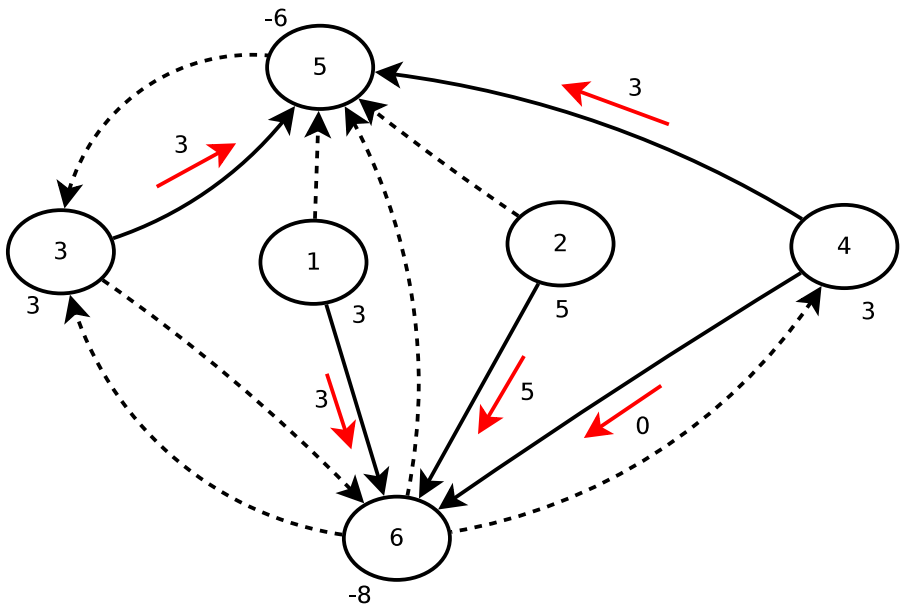

Figure 1.12: A tree-solution for the Minimum Cost Network Flow Problem

for the flow and it is equal to $z = \sum_{(i,j)\in A} c_{ij}x_{ij}$. The question in a MCNFP problem is to find a flow that minimizes that total cost. For every flow x_{ij} on an arc (i,j) there can be a lower and an upper bound, denoted as l_{ij} and u_{ij} respectively. This gives us additional constraints of the form $l_{ij} \leq x_{ij} \leq u_{ij}$, that have to hold for every arc $(i,j) \in A$. 1 In our case, we consider that it is $l_{ij} = 0$ and $u_{ij} = +\infty, \forall (i,j) \in A$. That is, we'll deal only with the *uncapacitated MCNFP* problem. Figure 1.12 shows a flow for the network in Figure 1.11. For every solid line arc (i,j) in the figure, there is a flow x_{ij} from node i to node j. For all dotted line arcs there is no flow through them. The solid line arcs form a tree-solution that satisfies all the product demands in the network and its total cost is equal to $z = 3 \times 107 + 3 \times 32 + 3 \times 41 + 5 \times 104 + 0 \times 84 = 1060$. This is not an optimal solution since there exist other tree-solutions of less total cost.

For every node $i \in N$, it has to be:

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i \tag{1.1}$$

because for every node in the network, the outgoing flow must be equal to the incoming flow plus the node's supply. Thefore, the mathematical formulation for the MCNFP problem is as follows:

$$minimize \quad z = \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{1.2}$$

subject to the constraints below:

$$\begin{aligned} &\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i, \forall i \in N \\ &x_{ij} \geq 0, \forall (i,j) \in A \end{aligned} \tag{1.3}$$

For a balanced network, the total supply has to be equal to the total demand, i.e. it is $\sum_{i\in N} b_i = 0$. Therefore, by using Relation (1.3), it comes out that:

$$\sum_{i\in N} \Big(\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} \Big) = 0 \tag{1.4}$$

That means that the constraints in Relation (1.3) are linearly dependent so, we can arbitrarily drop out one of them.

In matrix notation format, the problem can be expressed as follows:

$$\begin{aligned} &minimize \quad z = c^T x \\ &s.t. \quad Ax = b \\ &x \geq 0 \end{aligned} \tag{1.5}$$

where $A \in \Re^{n \times m}$, $x \in \Re^m$, $c \in \Re^m$ and $b \in \Re^n$. Notation c^T denotes the transpose of vector c, containing the arc costs per product unit. Matrix A is the incident matrix for network G and vector b contains the node supplies/demands. For the MCNFP problem in Figure 1.11 the incident matrix A is:

$$x^T = [x_{15},\ x_{16},\ x_{25},\ x_{26},\ x_{35},\ x_{36},\ x_{45},\ x_{46},\ x_{53},\ x_{63},\ x_{64},\ x_{65}]$$

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & 0 \\ -1 & 0 & -1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & -1 & 0 & -1 & 0 & -1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Above the columns of matrix A, we have the flows of the corresponding arcs, as they appear in the vector of flows x. For the vector c of costs for Figure 1.11 we have:

$$c^T = [\,16,\ 41,\ 57,\ 104,\ 107,\ 130,\ 32,\ 84,\ 71,\ 43,\ 0,\ 1\,]$$

and for the vector b of supplies/demands it is:

$$b^T = [\,3,\ 5,\ 3,\ 3,\ -6,\ -8\,]$$

The MCNFP problem, as it can be seen in Relations 1.2 and 1.5, is a special case of the linear problem and can be solved by using standard Linear Programming (LP) techniques. The MCNFP problem though, has some special properties, not met in other LP problems. The incident matrix A in every column contains a unique value equal to 1 and a unique value equal to -1. All the other values are equal to 0. Moreover, as it is shown in Relation (1.3), all the constraints form equalities instead of inequalities. These characteristics allow the development of algorithms for the MCNFP problem that work faster than the standard Linear Programming algorithms.

For every linear problem, there exists a unique linear problem which is called its *dual problem*. The original problem is called the *primal problem*. That is, all the linear problems, and therefore all the MCNFP problems, come in primal/dual pairs. The dual of the dual problem is the original primal problem. The dual problem for the MCNFP problem uses a set of *dual variables* w_i, one for every node i of the network, i.e. we have one dual variable for every primal constraint. The dual problem also uses a number of *reduced cost* variables s_{ij}, one for every directed arc $(i, j) \in A$. The dual MCNFP problem is a maximization problem and in matrix notation format it is described as follows:

$$\begin{aligned} &\text{maximize } \ z = b^T w \\ &s.t. \ A^T w + I_m s = c \\ &s \geq 0 \end{aligned} \tag{1.6}$$

where $A \in \Re^{n \times m}$, $c \in \Re^m$, $w \in \Re^n$, $s \in \Re^m$, $b \in \Re^n$ and I_m is the unit matrix of size m. For the dual variable values w_i it is

$$w_i - w_j = c_{ij}, \forall (i,j) \in T \tag{1.7}$$

Since we have one dual variable for each primal constraint and the primal constraints are linearly dependent, as it is stated in Relation (1.4), we can arbitrarily choose one dual variable w_r and give any value we want to it. Then, it is very easy to compute the values for the rest of the dual variables. Node r, that corresponds to the dual selected variable w_r, is considered to be the *root* of the basic tree-solution.

In an optimization problem, a *slack variable* is a variable that is added to an inequality constraint to transform it to an equality. The reduced cost variables are actually the slack variables for the dual problem (*dual slacks*), as is can been seen in Relation (1.6), and their values can be computed by the following relation:

$$s_{ij} = c_{ij} - w_i + w_j, \forall (i,j) \notin T \tag{1.8}$$

If for a solution x of the primal problem, it is $x_{ij} \geq 0, \forall (i,j) \in T$, then that solution x is called a *primal feasible* solution. If, on the other hand, for a tree-solution w of the dual problem, it is $s_{ij} \geq 0 \forall (i,j) \notin T$ then it is called

a *dual feasible* solution.

The solution of the dual problem provides an upper bound to the solution of the primal problem. That is, we always have:

$$\sum_{(i,j)\in A} c_{ij}x_{ij} \leq \sum_{i=1}^{n} b_i w_i \tag{1.9}$$

or in matrix notation form, it is:

$$c^T x \leq b^T w \tag{1.10}$$

Relations (1.9) and (1.10) are the result of a theorem called the *weak duality theorem*. A consequence of this theorem is that, if we can find a primal feasible solution x* and a dual feasible solution w*, such that $c^T x^* = b^T w^*$, then both solutions x* and w* are optimal solutions for the primal and dual problem respectively.

A second very useful theorem, called the *strong duality theorem*, states that if the primal problem has an optimal solution x*, then the dual problem has also an optimal solution w* and vice versa. For the optimal solutions x* and w*, it is $c^T x^* = b^T w^*$.

There is not always an optimal solution. A linear problem is said to be *primal infeasible*, if no solutions exist that satisfy all the constraints in Relation (1.3) of the primal problem. Similarly, a problem is called *dual infeasible* if there is no solution that satisfies the constraints in Relation (1.6) of the dual problem. A linear program can also be *unbounded*, i.e. the objective value is possible to go to $-\infty$ for a minimization problem or to $+\infty$ for a maximization problem. The weak duality theorem tells us that, if the

primal problem is unbounded then its dual problem is infeasible. Likewise, if the dual is unbounded, then the primal must be infeasible. However, it is possible for both the dual and the primal to be infeasible

A primal Simplex-type algorithm tries to find an optimal solution for the primal problem, while a dual Simplex-type algorithm tries to find an optimal solution for the dual problem. The two solutions, as stated by the duality theorems, correspond to the same objective value and, by solving a linear problem, its dual problem is also solved. Some classic Simple-type algorithms for the MCNFP problem are presented in section 1.5.

1.4 Specializations of the MCNFP problem

The MCNFP problem formulation is very broad and it can be used to model a number of various important network problems. Some problems that can be solved as a specialization of the MCNFP problem are described in this section.

1.4.1 The Assignment problem

In the assignment problem, we have a graph $G = (N, A)$ where the set of nodes N is partitioned into two disjoint subsets N_1 and N_2 of equal size. That is $N = N_1 \cup N_2$ and $N_1 \cap N_2 = \emptyset$. Additionally, for all the arcs $(i, j) \in A$, it is $i \in N_1$ and $j \in N_2$. The problem is to assign to each node in N_1, a node of N_2, at the minimum possible cost. The cost of assigning a node $i \in N_1$ to a node $j \in N_2$ is denoted as c_{ij}. We set the parameters of the MCNFP problem, as it is shown below:

$b_i = 1, \forall i \in N_1$

$b_i = -1, \forall i \in N_2$

$l_{ij} = 0, \forall (i,j) \in A$

$u_{ij} = 1, \forall (i,j) \in A$

The formulation of the problem is:

$$minimize \quad z = \sum_{(i,j)\in A} c_{ij} x_{ij}$$

subject to the following constraints:

$$\sum_{(i,j)\in A} x_{ij} = 1, \forall i \in N_1$$
$$\sum_{(i,j)\in A} x_{ij} = -1, \forall j \in N_2$$
$$0 \leq x_{ij} \leq 1, \forall (i,j) \in A$$

1.4.2 The Transportation problem

In this problem, the set of nodes N is again partitioned into two subsets N_1 and N_2 so that, $N = N_1 \cup N_2$, $N_1 \cap N_2 = \emptyset$ and $\forall (i,j) \in A$, $i \in N_1$, $j \in N_2$. All the nodes in N_1 are supply nodes (sources) and all the nodes in N_2 are demand nodes (sinks). The problem is to find the flow of least cost from the supply nodes of N_1 to the sink nodes in N_2. We denote as b_i the supply/demand at node i, x_{ij} the flow from source node i to sink node j and c_{ij} the cost of shipping one product unit from node i to node j. There are no upper bounds on flows.

The formulation of the problem is as follows:

$$minimize \quad z = \sum_{(i,j)\in A} c_{ij}x_{ij}$$

subject to the constraints:

$$\sum_{(i,j)\in A} x_{ij} = b_i, \forall i \in N_1$$
$$\sum_{(i,j)\in A} x_{ij} = -b_j, \forall j \in N_2$$
$$x_{ij} \geq 0, \forall (i,j) \in A$$

1.4.3 The single-source Shortest Path problem

The single-source shortest path problem is the problem of finding in a network, the directed paths of shortest length from a given node (source) to all other nodes of the network. We assume, without loss of generality, that node 1 is the source node. We can set the parameters of the MCNFP problem, as follows:

$b_1 = n - 1$

$b_i = 1, \forall i \in N$

c_{ij} = length of arc (i,j)

$l_{ij} = 0$

$u_{ij} = n - 1, \forall (i,j) \in A$

Therefore, the formulation of the problem is:

$$minimize \quad z = \sum_{(i,j)\in A} c_{ij}x_{ij}$$

subject to the constraints below:

$$\sum_{(1,j)\in A} x_{1j} - \sum_{(j,1)\in A} x_{j1} = n-1,$$
$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = -1, \forall i \in N, i > 1$$
$$0 \leq x_{ij} \leq n-1, \forall (i,j) \in A$$

1.4.4 The Maximum Flow problem

The maximum flow problem is the problem to send the maximum possible flow from a source node to a specified sink node. Every arc (i, j) has its own capacity Cap_{ij} and cannot pass flow greater than this capacity. Without loss of generality, let's assume that node 1 is the source node and node n is the sink node. We can add a new artificial arc that connects the sink node to the source node and we set the parameters of the MCNFP problem, as follows:

$c_{n1} = 1$

$l_{n1} = 0$

$u_{n1} = \infty$

$b_i = 0, \forall i \in N$

$u_{ij} = Cap_{ij}, \forall (i,j) \in A, (i,j) \neq (n,1)$

$l_{ij} = 0, \forall (i,j) \in A$

$u_{ij} = n-1, \forall (i,j) \in A$

Therefore, the formulation of the problem is:

$$minimize \quad -x_{n1}$$

subject to the constraints below:

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = 0,\ \forall i \in N$$

$$0 \leq x_{ij} \leq u_{ij},\ \forall (i,j) \in A$$

$$x_{n1} \geq 0$$

1.5 Simplex-type algorithms for the Minimum Cost Network Flow Problem

In Linear Programming, the feasibility region, i.e. the set of all feasible solutions $x \geq 0$, forms a polyhedron. Simplex-type algorithms use solutions that correspond to the vertices of the polyhedron of the feasibility region, called *basic solutions*. In a Network Simplex-type algorithm for the MCNFP on a graph G=(N,A), a basic solution, as first described in [28], corresponds to a spanning tree of G, called a *basic tree*. An arc that belongs to a basic tree T is called a *basic arc*. The flow for all the non-basic arcs is always equal to 0. If the flow for all the basic arcs in T, denoted as x(T), is not negative, i.e. $x(T) \geq 0$, then the tree T is called *primal feasible* tree.

A Network Simplex-type algorithm for the MCNFP problem starts from an initial tree-solution T and computes vectors x, w and s that correspond to the values of flows, dual variables and reduced cost variables respectively. If it is $s_{ij} \geq 0,\ \forall (i,j) \notin T$, then the solution is dual feasible. If the algorithm manages to find a solution being both primal and dual feasible then it is an optimal solution and the algorithm stops.

Primal Network Simplex-type algorithms start from a primal feasible

tree-solution and they move, at every iteration, to a new primal feasible tree-solution, until they find an optimal solution. On the other hand, Dual Network Simplex-type algorithms start from a dual feasible tree-solution and they reach to an optimal solution by moving inside the dual feasibility region. Exterior Point Simplex-type algorithms start from a primal or dual feasible tree-solution, but they have the possibility to move outside the primal and dual feasibility region. That is, they use tree-solutions that are neither primal nor dual feasible, before they finally find a tree-solution that is both primal and dual feasible, i.e. it is optimal.

Every iteration substitutes the current basic tree T with a new tree T', until it finds a tree that corresponds to an optimal solution. The successive tree T' is derived from T after the addition into T of an *entering arc* (g,h) and the deletion from it of a *leaving arc* (k,l). Therefore, it is $T' = T \cup (g,h) \setminus (k,l)$. The succession from a tree T to the next tree T' is called a *pivot*. That way, a Simplex-type algorithm follows a sequence of successive trees $T^1, T^2, ..., T^n$, where T^1 is the initial tree-solution that the algorithm uses as a starting point. An initial tree-solution T^1 can always be found except of the case that the problem is infeasible. A final optimal solution T^n can always be found except of the cases where the MCNFP problem is either infeasible or unbounded.

A tree-solution found in an iteration is not necessarily better than the tree-solution found in the previous iteration. For some iterations *degenerate pivots* may occur. A pivot is degenerate when after the pivot is applied, the objective value for the new tree-solution is the same as the objective value of the old tree-solution. If degenerate pivots are applied one after the other,

problems like *cycling* or *stalling* may occur. Such problems are discussed in more detail in section 4.1.1.

Section 1.5.1 presents the classic Primal Network Simplex-type algorithm for the MCNFP problem and section 1.5.2 presents the corresponding dual Simplex-type algorithm. Section 1.5.3 describes briefly a primal Exterior Point Simplex-type algorithm for the MCNFP and finally, in section 1.5.4, some state of the art algorithms are presented, together with some interesting software implementations.

1.5.1 The Primal Network Simplex-type Algorithm for the MCNFP problem

The Primal Network Simplex-type algorithm, PNSA for short, was first presented in 1951 by Danzing in [27]. PNSA maintains a primal feasible tree-solution at every iteration. The algorithm iterates towards an optimal solution by exchanging basic with non-basic arcs and adjusting the flows accordingly. The algorithm's steps, for the uncapacitated MCNFP problem on a network G=(N,a), are as follows:

Step 0: Initialization

A transformation has to be performed to the problem so that we find an initial primal feasible tree-solution. A new artificial node, labeled n+1, is added to the graph. The supply for the new node is $b_{n+1} = 0$. Additionally n artificial arcs joining the artificial node to all other nodes $i \in N$ are also added to the graph. If node $i \in N$ is a supply or transshipment node, i.e. $b_i \geq 0$, then an artificial arc (i,n+1) is added and its cost per product unit

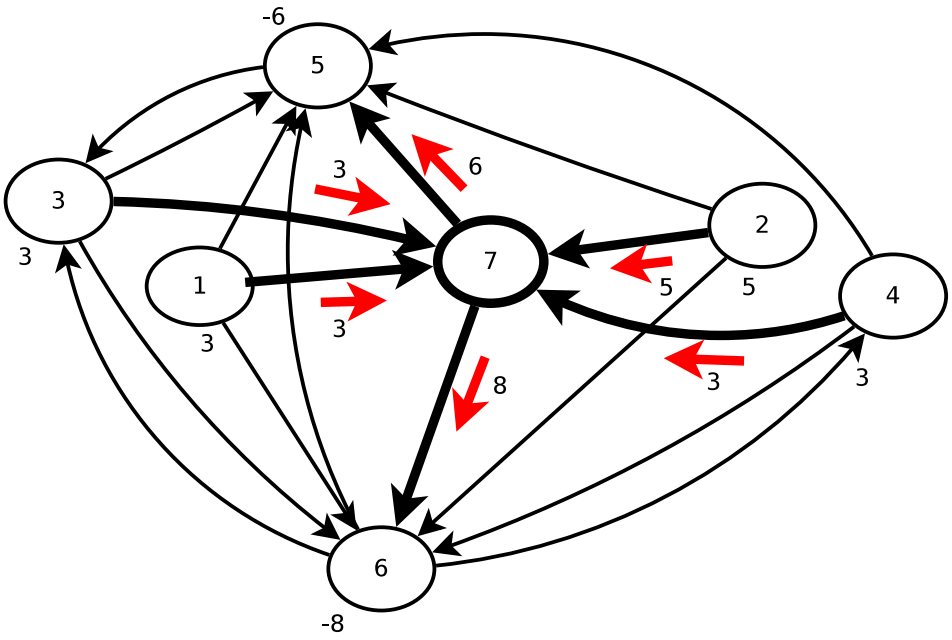

Figure 1.13: The big M method

is set equal to M, where M is a very big value. That is, we have $c_{i,n+1} = M$. If node $i \in N$ is a demand node, i.e. $b_i < 0$, then an artificial arc (n+1,i) is added and its cost per product unit is again set equal to M, i.e. $c_{n+1,i} = M$.

The method described above is named the *big M method* and it produces a new extended graph G'. In that graph G', it is very easy to find an initial primal feasible tree-solution T^1 (basic tree) by just including into it all the artificial arcs of G'. For an artificial arc (n+1,i) of T^1 ($i \in N$), we consider a flow $x_{n+1,i} = -b_i > 0$. On the other hand, for an artificial arc (i,n+1) of T^1 ($i \in N$), we consider a flow $x_{i,n+1} = b_i \geq 0$. For all other non-basic arcs $(i, j) \notin T^1$ we have $x_{ij} = 0$. The flow x produced that way is primal feasible, since it is $\sum_{i \in N} b_i = 0$ and $x \geq 0$. Figure 1.13 shows the extended graph G' produced by the big M method when it is applied on the graph of Figure 1.11. Node 7 is the new artificial node added and the arcs in bold line: (1,7), (2,7),

(3,7), (4,7), (7,5) and (7,6) are the artificial arcs needed for the flow to pass from the supply nodes to the demand nodes. The algorithm then processes the new graph G'. If it comes to an optimal solution and none of the artificial arcs has a positive flow, then an optimal solution for the original network has been found. If, on the other hand, there exist artificial arcs having possitive flow, then the original problem is infeasible.

A question that arises here is how large the value of M should be. We can think of M having a big enough value, but if we want to give to it a certain value, then we can assign to it any value that is greater than $(n-1)CU+1$, where C and U are computed by the relations: $C = max\{c_{ij}, (i,j) \in A\}$ and $U = \sum_{i \in N} b_i$.

After computing the initial flow x, PNSA has to compute the values of the dual variables w_i from Relation (1.7), and the reduced costs s_{ij} from Relation (1.8). In order to do that, it is enough to set any dual variable to an arbitrary value, e.g. $w_1 = 0$, and then solve some simple linear equations to compute the values of the other dual variables and all the reduced costs.

Step 1: Test of optimality

A non-basic arc has to be selected to enter the basic tree. The algorithm chooses an arc (g,h) that violates the dual feasibility conditions, i.e. an arc having $s_{gh} < 0$. If no such admissible arc exists, then the current solution is optimal, for the extended graph, since it is both primal and dual feasible. In that case, if the original problem is feasible, the final tree-solution contains no artificial arcs and an optimal solution for the original graph has been found. So, the algorithm stops. Otherwise, the algorithm continues with the next step.

Step 2: Select the entering arc

There are various rules for choosing the right *entering arc* but mostly known is *Dantzig's rule* that chooses an arc (g,h) having the minimum reduced cost value, i.e.:

$$s_{gh} = min\{s_{ij} : (i,j) \notin T \, and \, s_{ij} < 0\} \tag{1.11}$$

Step 3: Determine the leaving arc

After adding the non-basic arc (g,h) into the basic tree T, then a unique cycle C is created. Let C^+ denote the set of arcs (i,j) of C having the same orientation as (g,h), i.e. $(i,j) \uparrow\uparrow (g,h)$, and C^- denote the set of arcs (i,j) of C having orientation opposite to (g,h), i.e. $(i,j) \uparrow\uparrow (g,h)$. That is:

$$\begin{aligned} C^+ &= \{(i,j) : (i,j) \in C \, and \, (i,j) \uparrow\uparrow (g,h)\} \\ C^- &= \{(i,j) : (i,j) \in C \, and \, (i,j) \uparrow\downarrow (g,h)\} \end{aligned} \tag{1.12}$$

The entering arc (g,h) is a linear combination of the basic arcs $(i,j) \in T$, after multiplying each basic arc (i,j) by a coefficient h_{ij}. For a basic arc (i,j) that not belongs into cycle C, i.e. $(i,j) \notin C$, it is $h_{ij} = 0$. If a basic arc (i,j) belongs into C^+, then it is $h_{ij} = -1$. Otherwise, if $(i,j) \in C^-$, then it is $h_{ij} = -1$. Therefore, we have:

$$h_{ij} = \begin{cases} 0 & , if \, (i,j) \notin C \\ -1 & , if \, (i,j) \in C^+ \\ +1 & , if \, (i,j) \in C^- \end{cases} \tag{1.13}$$

In order to eliminate the cycle C, created after adding the entering arc

(g,h) into the basic tree-solution T, one of the basic arcs must leave the basic tree T. The *leaving arc* (k,l) is the arc of minimum flow that belongs into C^-. That is, for the leaving arc (k,l) we have:

$$x_{kl} = min\{x_{ij} : (i,j) \in C^-\} \tag{1.14}$$

If it is $C^- = \emptyset$ then the problem is unbounded.

Step 4: Pivot

A new basic tree-solution $T*$ is produced by adding into the current tree-solution T the entering arc (g,h) and removing the leaving arc (k,l) from it. That is, the new tree is $T* = T \cup \{(g,h)\} \setminus \{(k,l)\}$. Vectors x, w and s has to be updated for the new tree and after that, the algorithm repeats from Step 1.

For arcs $(i,j) \in C^+$ their updated flow is decreased by the amount of the flow x_{kl} on the leaving arc (k,l), whereas for arcs $(i,j) \in C^-$ their updated flow is increased by x_{kl}. If we denote $x_{ij}^{(t)}$ the flow of arc (i,j) in the current iteration t, then it is:

$$x_{ij}^{(t+1)} = \begin{cases} x_{ij}^{(t)} - x_{kl}^{(t)} & , if\ (i,j) \in C^+ \\ x_{ij}^{(t)} + x_{kl}^{(t)} & , if\ (i,j) \in C^- \\ x_{ij}^{(t)} & , otherwise \end{cases} \tag{1.15}$$

The next issue is how to update the values for the dual variables. When the leaving arc (k,l) is removed from the basic tree (without the addition of the entering arc), two subtrees are produced. Let T' be the subtree that contains the root r of the tree (the node that corresponds to the dual vari-

able w_r to whom an arbitrary value was given) and T'' be the subtree not containing r. Figure 1.14 shows the two subtrees produced when the leaving arc (k,l) is removed from the basic tree T. Recalling that the dual variables are calculated starting with the root node r and working up the basic tree T, it is clear that the dual variables for the nodes of subtree T' remain unchanged, whereas those for the nodes of T'' change by the same amount. If the entering arc (g,h) crosses from subtree T' to subtree T'', then all dual variables on T'' are increased by s_{gh}. Otherwise, all dual variables on T'' are decreased by s_{gh}. Therefore, the values of the dual variables are updated as shown below:

$$w_i^{(t+1)} = \begin{cases} w_i^{(t)} + s_{gh}^{(t)} & , if\, i \in T'',\, g \in T' \, and \, h \in T'' \\ w_i^{(t)} - s_{gh}^{(t)} & , if\, i \in T'',\, g \in T'' \, and \, h \in T' \\ w_i^{(t)} & , otherwise \end{cases} \tag{1.16}$$

Finally, we must update the reduced costs variables (dual slacks). The only reduced cost variables that change are those that span across the two subtrees T' and T''. For these nodes, the dual variable of either the head or the tail of the arc changes. For the arcs that bridge the two subtrees in the same direction as the entering arc (g,h), their dual slcks are decremented by s_{gh}. On the other hand, for the arcs bridging the two subtrees in the opposite direction, their dual slacks are incremented by s_{gh}. That is:

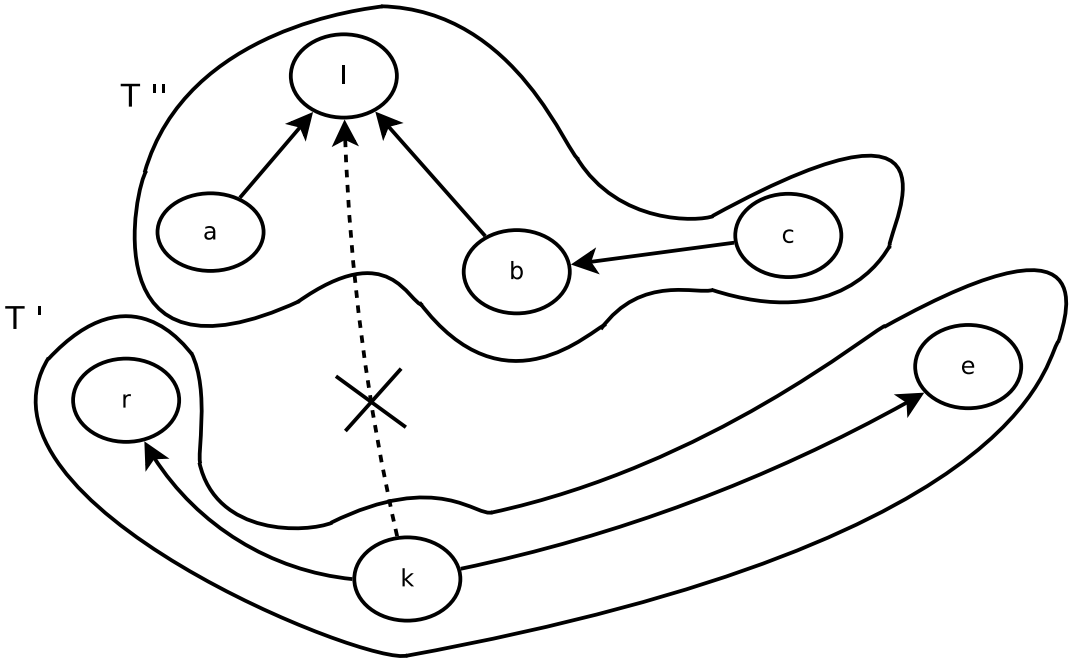

Figure 1.14: Subtrees produced from the basic tree after removing the leaving arc

$$s_{ij}^{(t+1)} = \begin{cases} s_{ij}^{(t)} - s_{gh}^{(t)} & , if\,(i,g \in T' \,and\, j,h \in T'')\, or\, (i,g \in T'' \,and\, j,h \in T') \\ s_{ij}^{(t)} + s_{gh}^{(t)} & , if\,(i,h \in T' \,and\, j,g \in T'')\, or\, (i,h \in T'' \,and\, j,g \in T') \\ s_{ij}^{(t)} & , otherwise \end{cases} \tag{1.17}$$

All the steps for the Primal Network Simplex-type Algorithm for the MCNFP problem are given in Algorithm 1.

Algorithm 1 The Primal Network Simplex-type Algorithm

Require: $G = (N, A), b, c$

procedure PNSA(G)

Step 0 (Initialization)

Compute flows x (big M method) and vectors w and s by using Relations (1.7) and (1.8) respectively.

Step 1 (Test of optimality)

if $s_{ij} \geq 0, \forall (i,j) \notin T$ **then**

if there are no artificial arcs in T **then**

STOP. An optimal solution is found.

else

STOP. The problem is infeasible.

end if

end if

Step 2 (Select the entering arc)

Choose the entering arc (g, h) by using Relation (1.11).

Step 3 (Determine the leaving arc)

Find sets C^+ and C^- using Relation (1.12).

if $C^- = \emptyset$ **then**

STOP. The problem is unbounded.

else

Choose the leaving arc (k, l) by using Relation (1.14).

end if

Step 4 (Pivoting)

Set $T = T \cup \{(g,h)\} \setminus \{(k,l)\}$.

Update vectors x, s, and w.

Go to Step 1.

end procedure

1.5.2 The Dual Network Simplex-type Algorithm for the MCNFP problem

The Dual Network Simplex-type algorithm, DNSA for short, starts from a dual feasible initial tree-solution T^1. In every iteration, the algorithm maintains a dual feasible basic tree and tries to find a tree-solution that is both dual and primal feasibly, i.e. it is optimal. Therefore, DNSA moves into the dual feasible region instead of the primal feasible region where PNSA moves into. Unlike PNSA, DNSA first selects the leaving arc and after that it chooses the entering arc. The algorithm's steps are described below:

Step 0: Initialization

First of all, an initial dual feasible tree-solution has to be constructed. This can be done by using the technique described in section 4.3.

Step 1: Test of optimality

The algorithm first has to select a leaving arc (k,l) that breaks the solution's primal feasibility. If no such arc exists, i.e. if for all the basic arcs (i,j) it is $x_{ij} \geq 0$, then the current solution is optimal. In that case, the algorithm stops. Otherwise, it continues to Step 2.

Step 2: Select the leaving arc

The algorithm selects the basic arc, having the minimum flow, to leave the basic tree-solution. So, for the leaving arc (k, l), we have:

$$x_{kl} = min\{x_{ij} : (i, j) \in T \, and \, x_{ij} < 0\} \tag{1.18}$$

Step 3: Determine the entering arc

When the leaving arc (k,l) is removed from the basic tree T, then two

disjoint subtrees T' and T" are produced. The entering arc (g,h) the algorithm chooses, is one that bridges the two subtrees in the opposite direction to the leaving arc (k,l). In other words, in the cycle C created when (g,h) is added to the basic tree T, the entering arc (g,h) has the same orientation as the leaving arc (k,l). That is, (k,l) must belong into set C^+, as it is defined in Relation (1.12). The leaving arc (k,l) must also have the smallest reduced cost amongst all the non-basic arcs (i,j) in C^+, i.e. it is:

$$s_{gh} = min\{s_{ij} : (i,j) \notin T \, and \, (i,j) \in C^+\} \tag{1.19}$$

There is no case the primal problem is unbounded. If the primal problem was unbounded, then the dual problem would be infeasible. In that case it would not be possible to find an initial tree during the initialization step.

Step 4: Pivot

A new basic tree is produced by removing the leaving arc (k,l) and adding the entering arc (g,h). That is, the new tree is $T \setminus \{(k,l)\} \cup \{(g,h)\}$. Vectors x, w and s has to be updated for the new tree and the algorithm repeats from Step 1.

The steps for the Dual Network Simplex-type Algorithm for the MCNFP problem are described in Algorithm 2.

1.5.3 The Primal Network Exterior Point Simplex-type Algorithm for the MCNFP problem

The Primal Network Exterior Point Simplex-type Algorithm for the MCNFP, PNEPSA for short, is a Simplex-type algorithm that, like PNSA, starts from

Algorithm 2 The Dual Network Simplex-type Algorithm

Require: $G = (N, A), b, c$

procedure DNSA(G)
Step 0 (Initialization)
    Find initial dual feasible tree and compute vectors w and s by using Relations (1.7) and (1.8) respectively.
Step 1 (Test of optimality)
    if $x_{ij} \geq 0, \forall (i,j) \in T$ **then**
        STOP. An optimal solution is found.
    end if
Step 2 (Select the leaving arc)
    Choose the leaving arc (k, l) by using Relation (1.18).
Step 3 (Determine the entering arc)
    Choose the entering arc (g, h) by using Relation (1.19).
Step 4 (Pivoting)
    Set $T = T \setminus \{(k,l)\} \cup \{(g,h)\}$.
    Update vectors x, s, and w.
    Go to Step 1.
end procedure

a primal feasible solution and after a number of iterations, it finds an optimal solution for the MCNFP problem. Unlike PNSA, it can go through tree-solutions that are not primal feasible. That is, it has the capability to move outside the primal feasible region. PNEPSA is described in detail in [84] and [78]. The algorithm's steps are as follows:

Step 0: Initialization

First the algorithm has to build an initial primal feasible tree T to start from. This can be done by using the big M method, described in section 1.5.1, that produced an extended network, as it is shown in Figure 1.13. The set of non-basic arcs is divided into two subset, a set P containing those arcs of negative reduced cost value and a set Q containing the rest of the arcs, as it is seen below:

$$P = \{(i,j) : (i,j) \notin T \, and \, s_{ij} < 0\}$$
$$Q = \{(i,j) : (i,j) \notin T \, and \, s_{ij} \geq 0\} \tag{1.20}$$

If a non-basic arc $(i,j) \notin T$ is added into the basic tree T, then a cycle C is created. For every arc (i,j), let h_{ij} be the vector of orientations in C of all the basic arcs of T compared to the orientation of (i,j). If an arc (u,v) in C has the same orientation as (i,j), then it is $h_{ij}(u,v) = -1$, otherwise we have $h_{ij}(u,v) = +1$. For all other arcs $(u,v) \notin C$, it is $h_{ij}(u,v) = 0$. Therefore, we have:

$$h_{ij}(u,v) = \begin{cases} 0 & , if \, (u,v) \notin C \\ -1 & , if \, (u,v) \in C \, and \, (u,v) \uparrow\uparrow (i,j) \\ +1 & , if \, (u,v) \in C \, and \, (u,v) \uparrow\downarrow (i,j) \end{cases} \tag{1.21}$$

The vectors of orientations h_{ij} are used for the computation of a direction vector d, that allows the algorithm to keep in touch with the primal feasible region. Vector d, for every arc (i,j), contains a value computed by the following formula:

$$d_{ij} = \begin{cases} -\sum\limits_{(i,j) \in P} h_{ij} & , if \, (i,j) \in T \\ 1 & , if \, (i,j) \in P \\ 0 & , if \, (i,j) \in Q \end{cases} \tag{1.22}$$

Step 1: Test of optimality

If $P = \emptyset$ then the current tree solution is optimal and the algorithm stops. If $P \neq \emptyset$ and $d(i,j) \geq 0, \forall (i,j) \in T$, then the problem is unbounded and the algorithm stops again. If non of the above is true, the algorithm continues

with the next step.

Step 2: Select the leaving arc

Vector d is used for the determination of the leaving arc (k,l), as it is shown in the following formula:

$$\frac{x_{kl}}{d_{kl}} = min\{\frac{x_{ij}}{d_{ij}} : (i,j) \in T \, and \, d_{ij} < 0\} \tag{1.23}$$

Step 3: Determine the entering arc

In order to determine the entering arc (g,h), the algorithm first computes two values θ_1 and θ_2, that give two candidate entering arcs (p_1, p_2) and (q_1, q_2), by using the following formula:

$$\begin{aligned} \theta_1 &= -s_{p_1 p_2} = min\{-s_{ij} : (i,j) \in P \, and \, h_{ij}(k,l) = 1\} \\ \theta_2 &= s_{q_1 q_2} = min\{s_{ij} : (i,j) \in Q \, and \, h_{ij}(k,l) = -1\} \end{aligned} \tag{1.24}$$

If it is $\theta_1 \leq \theta_2$, then we say we have a *type A iteration* and the algorithm chooses $(g,h) = (p_1, p_2)$ as the entering arc. Otherwise, we have a *type B iteration* and $(g,h) = (q_1, q_2)$ is the entering arc.

Step 4: Pivot

A new basic tree-solution is produced by removing the leaving arc (k,l) from the basic tree-solution and adding the entering arc (g,h) into it. That is, the new tree is $T \setminus \{(k,l)\} \cup \{(g,h)\}$. The leaving arc (k,l) is added into set Q. If it is $\theta_1 \leq \theta_2$ then it becomes $P = P \setminus \{(g,h)\}$, otherwise it becomes $Q = Q \setminus \{(g,h)\}$ Vectors x, w and s are updated and the algorithm repeats from Step 1.

The steps for the Primal Network Exterior Point Simplex-type Algorithm for the MCNFP problem are shown in Algorithm 3.

1.5.4 State of the art algorithms for the MCNFP problem

In complexity theory, an algorithm is said to be of *polynomial time* if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm. A *strongly polynomial-time algorithm* is one whose running time is bounded polynomially by a function only of the inherent dimensions of the problem and independent of the sizes of the numerical data. An algorithm for a network problem is strongly polynomial if its running time depends only on the number of nodes and arcs of the network, and not on the size of the costs or capacities. An algorithm that runs in polynomial time but is not strongly polynomial, is said to run in *weakly polynomial time*.

The first weakly polynomial time algorithm for the MCNFP problem was presented in 1972 by Edmonds and Karp and it used scaling techniques (see [29]) to solve the MCNFP problem. *Scaling* techniques work as follows: Given a network problem, it divides all capacities or supplies by two (or both). Then this scaled-down problem is solved recursively (or it could be solved iteratively). By doubling the solution, we get we take a near-optimal solution that can be transformed to an optimal solution of the original problem. A cost scaling technique is described in [83] and an algorithm using capacity scaling is presented in [68] by Orlin. A double scaling technique, i.e. a technique scaling both capacity and cost, was developed in [1].

Algorithm 3 The Primal Network Exterior Point Simplex-type Algorithm

Require: $G = (N, A), b, c$

procedure DNSA(G)

Step 0 (Initialization)

Find initial primal feasible tree (big M method)

Compute vectors w, s and d by using Relations (1.7), (1.8) and (1.22).

Step 1 (Test of optimality)

if $P = \emptyset$ **then**

STOP. An optimal solution is found.

else

if $d(i, j) \geq 0, \forall (i, j) \in T$ **then**

STOP. The problem is unbounded.

end if

end if

Step 2 (Select the leaving arc)

Choose the leaving arc (k, l) by using Relation (1.23).

Step 3 (Determine the entering arc)

Find θ_1 and θ_2 by using Relation (1.24).

if $\theta_1 \leq \theta_2$ **then**

the leaving arc is $(g, h) = (p_1, p_2)$.

else

the leaving arc is $(g, h) = (q_1, q_2)$.

end if

Step 4 (Pivoting)

Set $T = T \setminus \{(k, l)\} \cup \{(g, h)\}$.

Set $Q = Q \cup \{(k, l)\}$.

if $\theta_1 \leq \theta_2$ **then**

Set $P = P \setminus \{(g, h)\}$

else

Set $Q = Q \setminus \{(g, h)\}$

end if

Update vectors x, s, and w.

Go to Step 1.

end procedure

Edmonds and Karp in [29] stated that a challenging problem was to give a method for the minimum cost flow problem having a bound of computation, which is polynomial in the number of nodes and is independent of both costs and capacities. That is, they expressed the importance of the development of a strongly polynomial algorithm.

The first strongly polynomial algorithm was presented in 1985 by Tardos in [88]. She showed how to solve the MCNFP problem by solving $m = |E|$ distinct problems such that, in each problem it is $logC \leq 2log|V|$, where C is the maximum absolute cost value, given that all costs are integer values (otherwise it is ∞). She also showed how to extend her own technique to provide algorithms for all linear programs in which the constraint matrix coefficients are small. Other strongly polynomial algorithms were also presented in [35] and [30]. Orlin in 1997 (see [70]) presented a polynomial-time primal Simplex-type algorithm for the MCNFP problem that needs a number of $O(min\{nmlognC, nm^2logn\})$ iterations. Polynomial-time dual Simplex-type algorithms for the MCNFP problem have been described in [69] and [6].

Different approaches in the development of algorithms for the MCNFP problem include the cycle cancelling algorithm, the minimum mean cycle-cancelling method, the out-of-kilter algorithm and the steepest edge method, presented in [87], [46], [33] and [25] respectively. Moreover, new scaling techniques were developed like, the repeated capacity scaling technique in [47], the enhanced capacity scaling technique in [80] and the triple scaling method in [46].

Interior point methods were also used for the solution of he MCNFP

problem. Interior point methods (also referred as *barrier methods*) form a certain class of algorithms to solve linear (and non-linear) optimization problems. Contrary to the Simplex-type methods, they reach an optimal solution by traversing the interior of the (primal or dual) feasible region. Karmarkar in 1984 presented for the first time a polynomial interior point algorithm (see [57]). Plenty other similar interior point methods were described since then.

Miscellaneous network optimization codes for the MCNFP problem have been implemented. One of the most important implementations is *RELAX-IV* (see [13]) which is an evolvement of RELAX software that implements the particularly effective (in practice) relaxation method, presented in [12]. RELAX-IV combines the RELAX code with an initialization process, based on a shortest path algorithm. Other software suits were also developed, like *NETFLO* in [58], *cs2* that uses scaling techniques (see [48]), RNET in [50] and commercial CPLEX with its NETOPT solver. There are various comparative studies (see [16]) on the performance of these (and other) software implementations.

1.6 Innovation, objectives and structure of the Thesis

The Dual Network Exterior Point Simplex-type Algorithm for the MCNFP problem (DNEPSA for short) is the first Exteriorior-point Dual algorithm developed for the solution of the MCNFP problem. Only a Primal Exterior-point Simplex-type algorithm was presented in the previous years for the

MCNFP problem and DNEPSA shows superior computational results compared to the primal algorithm. Hence, the development of the algorithm introduces an innovation to the field of MCNFP algorithms since it is the only algorithm of its kind.

The main objective of this thesis is to give a detailed presentation of DNEPSA together with the proof of correctness of the algorithm. Illustrative examples and implementation details will be also given. A comparative computational study, against other classic Simplex-type algorithms, will try to indicate the importance of exterior point algorithms. Moreover, some aspects of DNEPSA's time complexity will be examined, first from a statistical point of view and later from a theoretical point of view, when dynamic trees data structures are used.

In chapter 1, an introduction to the Graph Theory is presented, together with the definitions that will be used during the rest of the thesis. The Minimum Cost Network Flow Problem (MCNFP) is defined and a reference to related work is given. Furthermore, some classic algorithms for the MCNFP problem are briefly described.

Chapter 2 gives an analytical presentation of DNEPSA. The steps of the algorithm are described in detail and additionally, a method for the quick update of the algorithm's variables is shown (section 2.3). Two illustrative examples demonstrates the way the algorithm works in practice. In chapter 3 a sequence of theorems prove the correct behaviour of the algorithms.

Implementation details for the algorithm, that concerns the methods and the data structures it uses, are given chapter 4. Some computational results about the algorithm's performance and a comparative study against the clas-

sic Dual Network Simplex-type Algorithm are presented in section 4.4. A statistical analysis is carried out in section 4.5. Finally, section 4.6 calculates the empirical complexity of the algorithm by using the computational results.

Chapter 5 describes an implementation of DNEPSA by using dynamic trees. The theoretical time complexity per iteration is also found. The conclusions of this thesis and the future work that have to been done are presented finally in chapter 5.

Chapter 2

The Dual Network Exterior Point Simplex-type Algorithm

2.1 Introduction

The Dual Network Exterior Point Simplex-type Algorithm, DNEPSA for short, was first presented by Georgios Geranis in [37]. As its name implies, is a Simplex-type algorithm that, starts from a dual feasible basic tree-solution T and tries to find an optimal solution by moving outside the dual (and primal) feasibility region. In other words, DNEPSA starts from a dual solution and reaches an optimal solution by following a route consisting of solutions that do not always belong to the feasible area of the dual problem. This is the main difference between DNEPSA and the other existing dual network simplex-type algorithms, like DNSA described in section 1.5.2. Furthermore, DNEPSA, contrary to the classical DNSA, first selects the entering arc and afterwards it selects the leaving arc. Finally, in DNEPSA the entering and

the leaving arc are selected by using different rules than the rules used in DNSA.

DNEPSA also works differently compared to PNEPSA, described in section 1.5.3. PNEPSA starts from a primal feasible solution and it works in a different way, while DNEPSA starts from a dual feasible solution. DNEPSA also shows much better performance compared to DNSA and NEPSA, as it is shown in section 4.4. The steps of the algorithm are described in detail in the following section.

2.2 Algorithm Description

The algorithm performs itteratively a number of steps until it satisfies the optimality condition, or it decides that the problem is infeasible. The steps of the algorithm are described below:

Step 0: Initialization

DNEPSA starts from an initial dual feasible basic tree-solution. Various methods may be used in order to find the starting tree. An algorithm that can construct a dual feasible tree-solution for the generalized network problem (and also for pure networks) was first described in [39]. An improved version of the same algorithm, that gives a dual feasible solution that is closer to an optimal solution, is presented in [53] and it is described in more detail in section 4.3.

The initial tree-solution T is dual feasible. That is, for the non-basic arcs $(i, j) \notin T$, it is $x_{ij} = 0$ and $s_{ij} \geq 0$, while for the basic arcs $(i, j) \in T$ it is $s_{ij} = 0$. The values of the dual variables w_i, $1 \leq i \leq n$, can be easily

computed from relation:

$$w_i - w_j = c_{ij},\ \forall (i,j) \in T \tag{2.1}$$

In Relation (2.1) we have $n-1$ equations and n variables, so we can choose one of the dual variables, e.g. w_1, and set it equal to an arbitrary value, e.g. 0. Then, it is easy to compute the values for the rest of the dual variables by just solving some first-order equations.

In order to compute the values for the reduced cost variables s_{ij} for all the non-basic arcs (i,j), we can use the relation:

$$s_{ij} = c_{ij} - w_i + w_j,\ \forall (i,j) \notin T \tag{2.2}$$

while it is $s_{ij} = 0$ for all the basic arcs. Next, the algorithm creates a set, named I_-, that contains the basic arcs (i,j) having negative flow, i.e. $x_{ij} < 0$. That is, I_- contains those arcs that violate the primal feasibility of the current tree-solution. The rest of the arcs, having $x_{ij} \geq 0$, belong in a set, named I_+. Therefore, it is:

$$I_- = \{(i,j) \in T : x_{ij} < 0\} \tag{2.3}$$

and

$$I_+ = \{(i,j) \in T : x_{ij} \geq 0\} \tag{2.4}$$

If a non-basic arc (i,j) is added into the basic tree T, then a cycle C_{ij} is created. Let h_{ij} be the vector of orientations in C_{ij} of all the basic arcs of

T relative to the orientation of the entering arc (i, j). If an arc (u, v) in C_{ij} has the same orientation as (i, j), then it is $h_{ij}(u, v) = -1$, otherwise it is $h_{ij}(u, v) = +1$. For an arc (u, v) not belonging into C_{ij}, we have $h_{ij}(u, v) = 0$. Let Cij^{+} denote the set of arcs (u,v) in C_{ij} having the same orientation as (i,j), i.e. $(i, j) \uparrow\uparrow (u, v)$ and let C_{ij}^{-} denote the set of arcs (u,v) of C_{ij} having orientation opposite to (i,j), i.e. $(i, j) \uparrow\downarrow (u, v)$. Then, it is:

$$\begin{aligned} C_{ij}^{+} &= \{(u, v) : (u, v) \in C_{ij} \, and \, (i, j) \uparrow\uparrow (u, v)\} \\ C_{ij}^{-} &= \{(u, v) : (u, v) \in C_{ij} \, and \, (i, j) \uparrow\downarrow (u, v)\} \end{aligned} \tag{2.5}$$

Therefore, for the vector h_{ij} we have:

$$h_{ij}(u, v) = \begin{cases} 0 & , if \, (u, v) \notin C_{ij} \\ -1 & , if \, (u, v) \in C_{ij}^{+} \\ +1 & , if \, (u, v) \in C_{ij}^{-} \end{cases} \tag{2.6}$$

In every iteration, DNEPSA finds out a new basic tree-solution which is probably neither primal nor dual feasible. The algorithm maintains a direction vector d pointing to the feasible region of the dual problem, so that it keeps in touch with it. Vector d is computed by using the following formula:

$$d_{ij} = \begin{cases} 1 & , if \, (i, j) \in I_{-} \\ 0 & , if \, (i, j) \in I_{+} \\ \sum\limits_{(u,v) \in I_{-}} h_{uv} & , if \, (i, j) \notin T \end{cases} \tag{2.7}$$

Step 1: Test of optimality

If the set I_{-} is empty, i.e. $I_{-} = \emptyset$, that means that there are no basic

arcs breaking the primal feasibility solution. In that case the current tree-solution is both primal and dual feasible and therefore, it is optimal. So, the algorithm stops. Otherwise, it is $I_{-} \neq \emptyset$ and DNEPSA creates a set named J_{-}, defined as shown below:

$$J_{-} = \{(i,j) \notin T : s_{ij} \geq 0 \; and \; d_{ij} < 0\} \tag{2.8}$$

If it is $J_{-} = \emptyset$, then the MCNFP problem is infeasible and the algorithm stops again. Otherwise, it continues with the next step.

Step 2: Selection of the entering arc

DNEPSA uses set J_{-} in order to compute the minimal ratio by using the following formula:

$$\alpha = \frac{s_{gh}}{-d_{gh}} = \min\{\frac{s_{ij}}{-d_{ij}} : (i,j) \in J_{-}\} \tag{2.9}$$

This ratio α, is used in order to choose the *entering arc* (g,h). The entering arc is the non-basic arc that minimizes the ratio shown in Relation (2.9). After (g,h) is added into the basic tree T, a cycle C is created. When a product unit flows through C, the value of the objective function is changed by the following amount:

$$\Delta z = \sum_{(i,j) \in C} t_{ij} c_{ij}$$

where t_{ij} equals 1 if the arcs (i,j) and (g,h) have the same orientation in cycle C, i.e. $(i,j) \uparrow\uparrow (g,h)$. Otherwise, in case $(i,j) \uparrow\downarrow (g,h)$, it is $t_{ij} = -1$. By using Relation (2.1) for $(i,j) \in T$, we have:

$$\Delta z = \sum_{(i,j)\in C} t_{ij} c_{ij} = c_{gh} - w_g + w_h$$

and by applying Relation (2.2) we take:

$$\Delta z = s_{gh} \tag{2.10}$$

Relation (2.10) implies that the change in the objective value equals the reduced cost value for the entering arc.

Step 3: Selection of the leaving arc

In order to find the *leaving arc* (k, l), DNEPSA calculates the values θ_1 and θ_2, as shown in the following formula:

$$\begin{aligned} \theta_1 &= -x_{k_1 l_1} = \min\{-x_{ij} : (i,j) \in I_- \ and \ \ (i,j) \uparrow\uparrow (g,h)\} \\ \theta_2 &= \ x_{k_2 l_2} = \min\{ \ x_{ij} : (i,j) \in I_+ \ and \ \ (i,j) \uparrow\downarrow (g,h)\} \end{aligned} \tag{2.11}$$

These values give two candidate arcs (k_1, l_1) and (k_2, l_2) and the algorithm chooses the one that will leave the basic tree by comparing the values of θ_1 and θ_2.

If $\theta_1 \leq \theta_2$, then the arc (k_1, l_1) is the leaving arc. In that case, we say we have a *type A iteration*. An arc of negative flow $x_{kl} = -\theta_1$ is leaving and an arc with flow $x_{gh} = \theta_1$ is entering the basic tree-solution. For the leaving arc (k, l), the subtree containing node k is denoted by T^+, while the other subtree, containing l, is denoted by T^-, as it is seen in Figure 2.1.

If, on the other hand, it is $\theta_1 > \theta_2$, then the arc (k_2, l_2) is the leaving arc. In that case, we say we have a *type B iteration*. An arc of positive

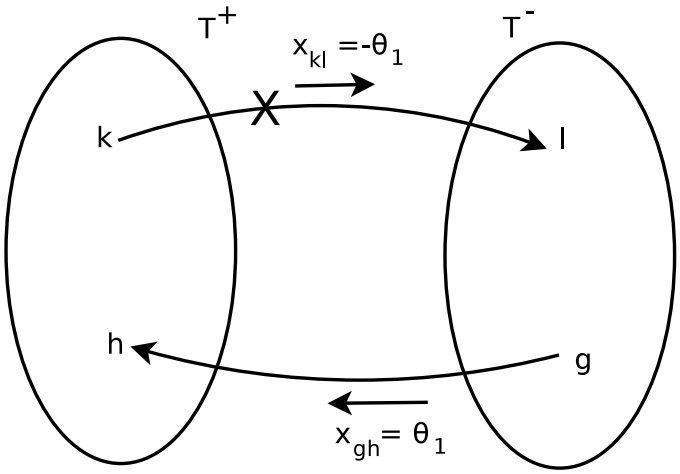

Figure 2.1: Type A iteration

flow $x_{kl} = \theta_2$ is leaving and an arc with flow $x_{gh} = \theta_2$ is entering the basic solution, as it can be seen in Figure 2.2.

Step 4: Pivoting

After selecting the entering and the leaving arc, the algorithm comes to a new tree-solution, closer to an optimal solution. For the new tree solution it is $T = T \setminus (k, l) \cup (g, h)$. The algorithm has to update vectors x, w and s for the new tree-solution and sets I_- and I_+. After that, it will repeat the same process from step 1. The update of the vectors and sets can be done as it is described in section 2.3.

The formal description of DNEPSA, in pseudocode format, is shown in Algorithm 4.

2.3 Update of variables and sets

It is not necessary for the algorithm in every iteration to compute the values of variables x_{ij}, s_{ij}, and d_{ij} or to create sets I_- and I_+ from scratch. The

Algorithm 4 The Dual Network Exerior Point Simplex-type Algorithm

Require: $G = (N, A), b, c, T$

procedure DNEPSA(G)
Step 0 (Initialization)
    Compute x, w, and s, using Relations (1.2), (2.1), and (2.2).
    Find sets I_- and I_+, using Relations (2.3) and (2.4).
    Compute vector d, using Relation (2.7).
Step 1 (Test of optimality)
    while $I_- \neq \emptyset$ **do**
        Find set J_-, using Relation (2.8).
        if $J_- = \emptyset$ **then**
            STOP. The problem 1.2 is infeasible.
        else
Step 2 (Selection of the entering arc)
            Compute α, using Relation (2.9).
            Select the entering arc (g, h).
Step 3 (Selection of the leaving arc)
            Compute θ_1, θ_2, using Relations (2.11).
            Select the leaving arc (k, l).
Step 4 (Pivoting)
            Set $T = T \setminus (k, l) \cup (g, h)$.
            Update vectors x, s, and d and sets I_- and I_+.
            if $\theta_1 \leq \theta_2$ **then**
                Set $I_- = I_- \setminus (k, l)$ and $I_+ = I_+ \cup (g, h)$
            else
                Set $I_+ = I_+ \cup (g, h) \setminus (k, l)$
            end if
        end if
    end while
    STOP. The current tree-solution is optimal.
end procedure

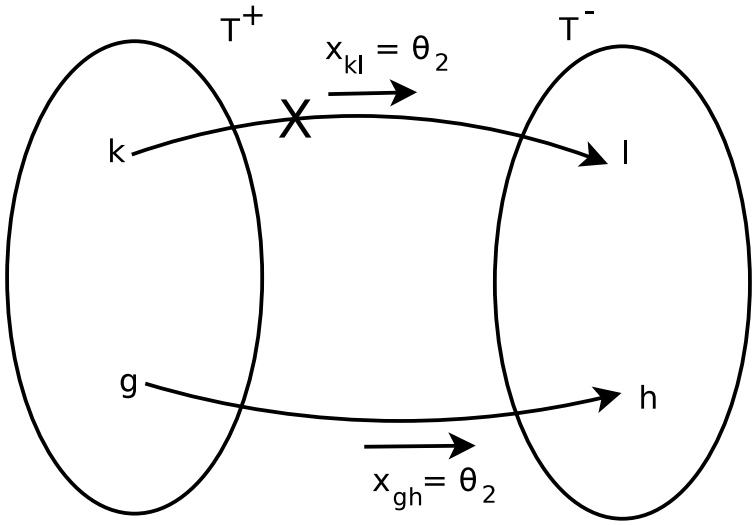

Figure 2.2: Type B iteration

Table 2.1: Cases for the basic arcs $(i, j) \in T$

Case 1	$(i, j) \notin C^{(t)}$
Case 2	(i,j)=(g,h) and type A iteration
Case 3	(i,j)=(g,h) and type B iteration
Case 4	(type A iteration and $(i, j) \uparrow\uparrow (g, h)$) or (type B iteration and $(i, j) \uparrow\downarrow (g, h)$)
Case 5	(type A iteration and $(i, j) \uparrow\downarrow (g, h)$) or (type B iteration and $(i, j) \uparrow\uparrow (g, h)$)

variables and sets used by DNEPSA can be efficiently updated from iteration to iteration. Let notation $x_{ij}^{(t)}$ mean the flow on arc (i, j) during iteration t of the algorithm. A similar notation is used for variables s_{ij} and d_{ij}. After adding the entering arc (g,h) into the basic tree T during iteration t, a unique cycle, denoted by $C^{(t)}$, is created. For the basic arcs $(i, j) \in T$ we can have the 5 different cases, shown in Table 2.1.

In iteration $t + 1$, for every basic arc (i, j), the flow $x_{ij}^{(t+1)}$ depends on the flow of the arc in the previous iteration $x_{ij}^{(t)}$ and the flow of the leaving arc $x_{kl}^{(t)}$, as it is shown in Table 2.2.

Table 2.2: Update of flows $x_{ij}, \forall (i,j) \in T$

Cases	$x_{ij}^{(t+1)}$
Case 1	$x_{ij}^{(t)}$
Case 2	$-x_{kl}^{(t)} = \theta_1$
Case 3	$x_{kl}^{(t)} = \theta_2$
Case 4	$x_{ij}^{(t)} - x_{kl}^{(t)}$
Case 5	$x_{ij}^{(t)} + x_{kl}^{(t)}$

Table 2.3: Cases for the non-basic arcs $(i,j) \notin T$

Case 1	$i,j \in T^+$ or $i,j \in T^-$
Case 2	$i \in T^+, j \in T^-$ and type A iteration
Case 3	$i \in T^+, j \in T^-$ and type B iteration
Case 4	$i \in T^-, j \in T^+$ and type A iteration
Case 5	$i \in T^-, j \in T^+$ and type B iteration

For the non basic arcs (i,j) we can also have different cases, depending on the type of iteration and the arc's position, as it is shown in Table 2.3.

In iteration $t+1$, the reduced costs $s_{ij}^{(t+1)}$ for the non basic arcs (i,j) depend on the reduced costs in the previous iteration $s_{ij}^{(t)}$ and the reduced cost for the entering arc $s_{gh}^{(t)}$. In Table 2.4, we can see how the s_{ij} values are updated, for the cases of Table 2.3.

In a similar way, Table 2.5 shows how the d_{ij} values are updated after each iteration, for the cases in Table 2.3.

Table 2.4: Update of reduced cost values $s_{ij}, \forall (i,j) \notin T$

Cases	$s_{ij}^{(t+1)}$
Case 1	$s_{ij}^{(t)}$
Case 2	$s_{ij}^{(t)} + s_{gh}^{(t)}$
Case 3	$s_{ij}^{(t)} - s_{gh}^{(t)}$
Case 4	$s_{ij}^{(t)} - s_{gh}^{(t)}$
Case 5	$s_{ij}^{(t)} + s_{gh}^{(t)}$

Table 2.5: Update of direction values $d_{ij}, \forall (i,j) \notin T$

Cases	$d_{ij}^{(t+1)}$
Case 1	$d_{ij}^{(t)}$
Case 2	$d_{ij}^{(t)} + d_{gh}^{(t)}$
Case 3	$d_{ij}^{(t)} - d_{gh}^{(t)}$
Case 4	$d_{ij}^{(t)} - d_{gh}^{(t)}$
Case 5	$d_{ij}^{(t)} + d_{gh}^{(t)}$

The update of the sets $I\_$ and I_+ depends only on the type of the iteration. We have the following two cases:

- *Case 1*: For a type A iteration, both sets change according to the following formulas:

$$\begin{aligned} I_+ &= I_+ \cup \{(g,h)\} \\ I_- &= I_- - \{(k,l)\} \end{aligned} \tag{2.12}$$

- *Case 2*: For a type B iteration, only set I_+ changes. This is done as it is shown below:

$$I_+ = I_+ \cup \{(g,h)\} - \{(k,l)\} \tag{2.13}$$

After the update of the vector variables and the sets is done, DNEPSA repeats the same process, from step 1, until it finds an optimal solution. In section 2.4 an illustrative example demonstrates how the algorithm works in practice.

2.4 Illustrative examples

In this section, two illustrative examples will be given showing how DNEPSA works in practice. For the first newtwork, there exists an optimal solution and DNEPSA comes to it after the iterations shown below. The second problem is infeasible and DNEPSA detects it after a couple of iterations, as it is shown in subsection 2.4.1.

First, a step by step example for the application of DNEPSA to a MCNFP problem is presented. The algorithm will be applied to the network G = (N, A) of Figure 1.11. Network G consists of 6 nodes and 12 arcs. Next to each node, there is a value showing supply for that node (negative values mean demands). For every arc in A, the cost per product unit flow is also shown. When DNEPSA is applied to the MCNFP problem, it finds an optimal solution after three iterations, as it is shown below:

Iteration 1

Step 0 (Initialization)

In order to start, the algorithm needs an initial dual feasible basic tree-solution. Figure 2.3 shows a dual feasible tree-solution that can be used by DNEPSA as a starting point. Such an initial solution can be obtained by using existing techniques, like the technique described in section 4.3.

The tree-solution T shown in Figure 2.3 is a dual feasible tree, since for the reduced cost variables it is $s_{ij} \geq 0, \forall (i,j) \in T$. This can be easily verified by using Relation (2.1), which takes the following forms:

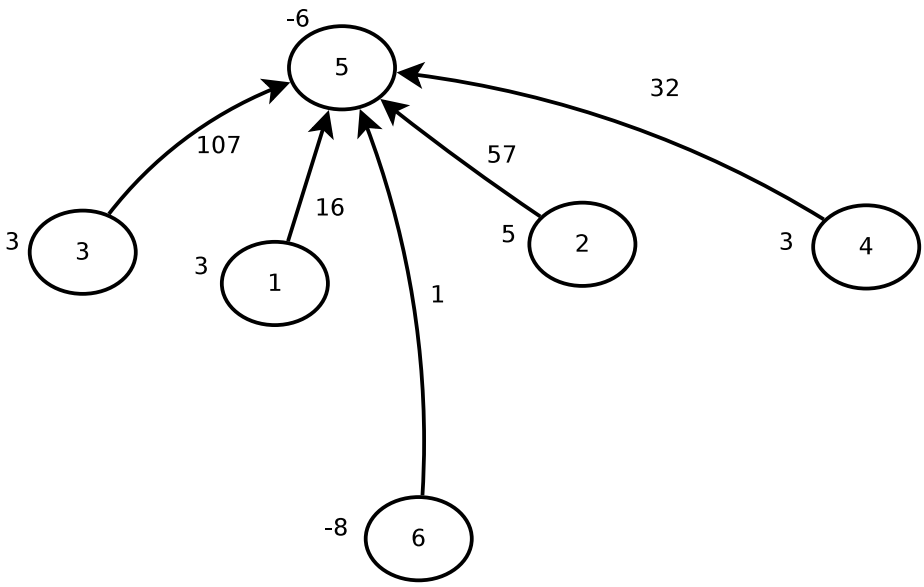

Figure 2.3: Initial dual feasible tree-solution

$$\begin{aligned} w_5 - w_1 &= 16 \\ w_5 - w_2 &= 57 \\ w_5 - w_3 &= 107 \\ w_5 - w_4 &= 32 \\ w_5 - w_6 &= 1 \end{aligned} \tag{2.14}$$

By setting arbitrarily the value of w_1 equal to 0, from Relation (2.14), we can very easily find $w_5 = 16, w_2 = -41, w_3 = -91, w_4 = -16$ and $w_6 = 15$.

For the computation of the reduced cost values for all non-basic arcs $(i, j) \notin T$, after using Relation (2.2), we have:

$$s_{16} = c_{16} + w_1 - w_6 = 41 + 0 - 15 = 26$$
$$s_{26} = c_{26} + w_2 - w_6 = 104 + (-41) - 15 = 48$$
$$s_{36} = c_{36} + w_3 - w_6 = 130 + (-91) - 15 = 24$$
$$s_{46} = c_{46} + w_4 - w_6 = 84 + (-16) - 15 = 53$$
$$s_{53} = c_{53} + w_5 - w_3 = 71 + 16 - (-91) = 178$$
$$s_{64} = c_{64} + w_6 - w_4 = 0 + 15 - (-16) = 31$$
$$s_{63} = c_{63} + w_6 - w_3 = 43 + 15 - (-91) = 149$$

For the basic arcs $(i, j) \in T$, it is $s_{ij} = 0$.

For all nodes $i \in N$ of the graph, the outgoing flow has to be equal to the incoming flow plus the supply of the node. Therefore, it is:

$$\sum_{(i,k)\in N} x_{ik} - \sum_{(j,i)\in N} x_{ji} = b_i$$

and thus we have:

$$node1 : x_{15} = 3$$
$$node2 : x_{25} = 5$$
$$node3 : x_{35} = 3$$
$$node4 : x_{45} = 3$$
$$node5 : -x_{15} - x_{25} - x_{35} - x_{45} - x_{65} = -6$$
$$node6 : x_{65} = -8$$

By solving the above equations we can easily compute the flows x for the initial tree-solution T: $x_{15} = 3, x_{25} = 5, x_{35} = 3, x_{45} = 3$ and $x_{65} = -8$. This is not a feasible tree, since there exist negative flows (it is $x_{65} < 0$).

We have $I_- = \{(6,5)\}$, because it is $x_{65} = -8 < 0$. The rest of the arcs form set $I_+ = \{(1,5),(2,5),(3,5),(4,5)\}$.

If we add the non-basic arc (1,6) into the basic tree, a cycle C is created. In this cycle, arc (1,6) has the same orientation as (6,5), which is the only arc of set I_-. So, $d_{16} = -1$. By checking in a similar way all the non-basic arcs, by using Relation (2.7), it comes out that it is: $d_{26} = -1, d_{36} = -1$ and $d_{46} = -1$, since the arcs (2,6), (3,6) and (4,6) have the same orientation in the cycle C created, as arc $(6,5) \in I_-$. On the other hand, it is $d_{53} = 0, d_{64} = 1$ and $d_{63} = 1$.

Step 1 (Test of optimality)

It is $I_- \neq \emptyset$, so the current tree-solution is not optimal. From Relation (2.8), the algorithm creates set $J_- = \{(1,6),(2,6),(3,6),(4,6)\}$. It is $J_- \neq \emptyset$, so DNEPSA continues to step 2. If it was $J_- = \emptyset$, then the problem would be infeasible.

Step 2 (Selection of the entering arc)

From Relation (2.9), the algorithm computes ratio α:

$$\alpha = \frac{s_{36}}{-d_{36}} = \frac{24}{1} = min\{\frac{s_{16}}{-d_{16}}, \frac{s_{26}}{-d_{26}}, \frac{s_{36}}{-d_{36}}, \frac{s_{46}}{-d_{46}}\} = min\{\frac{26}{1}, \frac{48}{1}, \frac{24}{1}, \frac{53}{1}\}$$

So, arc (3,6) is the entering arc.

Step 3 (Selection of the leaving arc)

After adding the entering arc (3,6) into the basic tree-solution T, a cycle C is created, as it is shown in Figure 2.4. That cycle contains only two basic arcs: (3,6) and (6,5). Arc (6,5) belongs into set I_- and it has the same

orientation as the entering arc (3,6). So, it is $\theta_1 = -x_{65} = 8$. Arc (3,5), on the other hand, belongs into set I_+ and it does not have the same orientation as the entering arc (3,6). So, it is $\theta_2 = x_{35} = 3$. We have $\theta_1 > \theta_2$ which means that we have a type B iteration and arc (3,5) is the leaving arc.

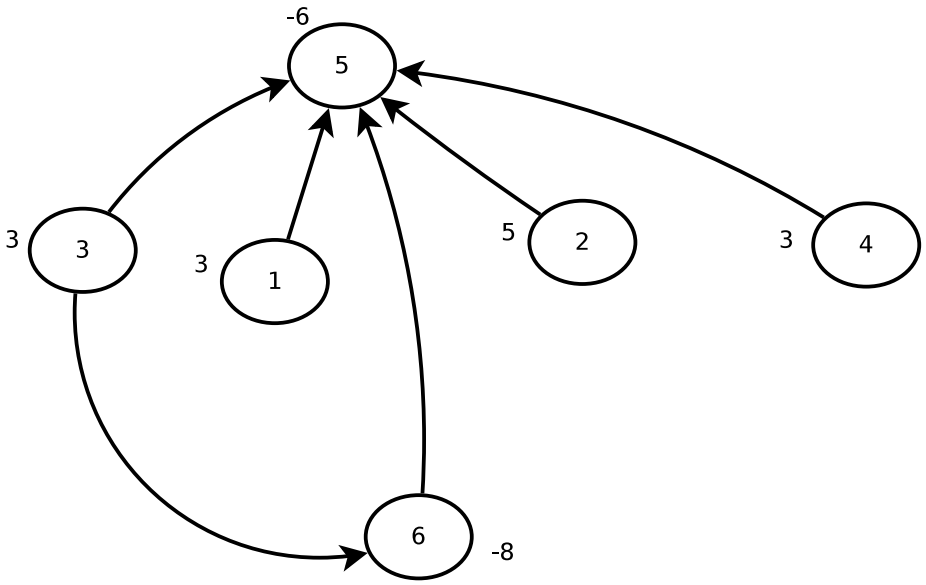

Figure 2.4: Cycle created when adding the entering arc into the basic tree

Step 4 (Pivot)

After removing the leaving arc from the basic tree-solution T and adding the entering arc into T, a new tree, shown in Figure 2.5, is produced. Vectors x, s and d has to be updated according to Tables (2.2), (2.4) and (2.5). The flow for the entering arc is $x_{36} = \theta_2 = 3$ and for arc $(6, 5) \in C$, it becomes $x_{65} = -8 + 3 = -5$. The rest of the flows remain unchanged.

When the leaving arc (3,5) is removed from the original tree T, then subtrees T^+ and T^-, shown in Figure 2.6, are created. Only the reduced costs for the arcs that cross from one subtree to the other are changed, the leaving arc included. Those arcs, as Table 2.4 implies, change their

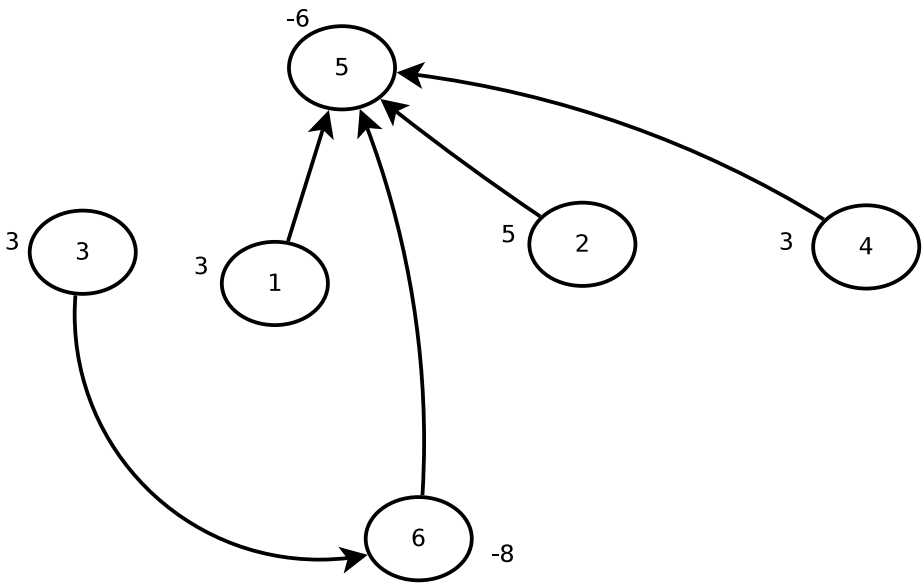

Figure 2.5: The basic tree after the first iteration

reduced cost value by adding or subtracting value $s_{36} = 24$ (depending on their orientation). So, we have $s_{35} = 0 - 24 = -24$, $s_{53} = 178 + 24 = 202$ and $s_{63} = 149 + 24 = 173$. In a similar way, as Table 2.5 implies, the values of d_{35}, d_{53} and d_{63} change by an amount of $d_{36} = -1$. So, it becomes: $d_{35} = 0 - (-1) = 1$, $d_{53} = 0 + (-1) = -1$ and $d_{63} = 1 + (-1) = 0$.

Since the algorithm performed a type B iteration, only set I_+ is changed, according to Relation (2.13), while set I_- remains unchanged. It is $I_+ = \{(1,5),(2,5),(3,5),(4,5)\} \cup \{(3,6)\} \setminus \{(3,5)\} = \{(1,5),(2,5),(4,5),(3,6)\}$.

The algorithm continues to the second iteration for the new basic tree-solution (Figure 2.5), starting from Step 1.

Iteration 2

Step 1 (Test of optimality)

It is $I_- \neq \emptyset$, so the current tree-solution is not optimal. From Relation (2.8), it comes out that $J_- = \{(1,6),(2,6),(4,6),(5,3)\}$. It is $J_- \neq \emptyset$, so

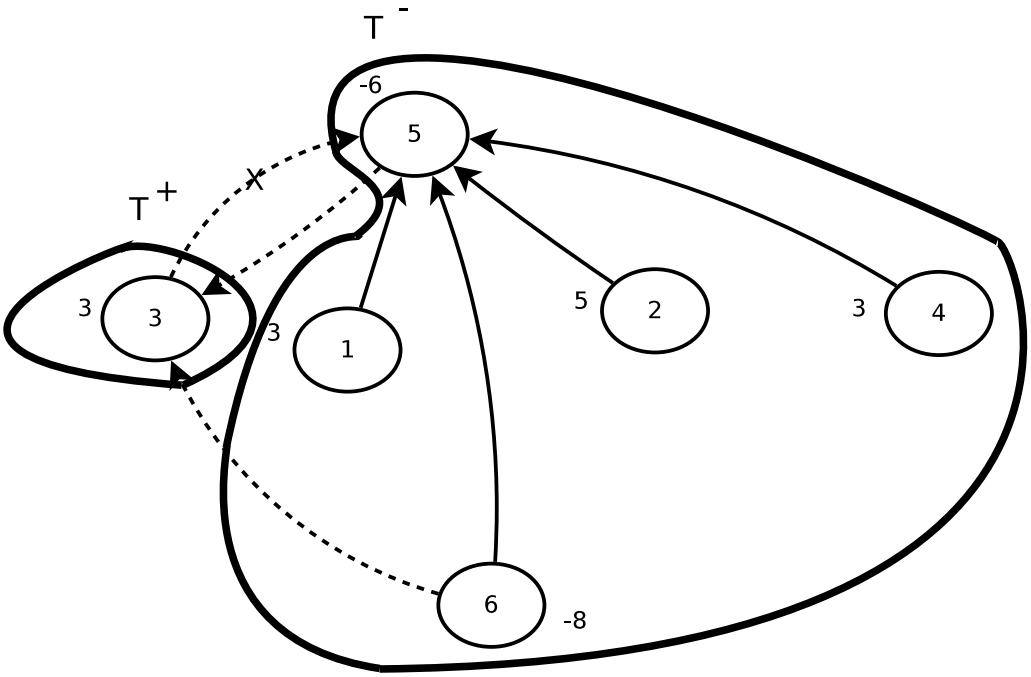

Figure 2.6: The subtrees created after the removal of the leaving arc

DNEPSA continues to step 2.

Step 2 (Selection of the entering arc)

From Relation (2.9), we find:

$$\alpha = \frac{s_{16}}{-d_{16}} = \frac{26}{1} = min\{\frac{26}{1}, \frac{48}{1}, \frac{53}{1}, \frac{202}{1}\}$$

So, arc (1,6) is the entering arc.

Step 3 (Selection of the leaving arc)

After adding the entering arc (1,6) into the basic tree T, a cycle C is created, as shown in Figure 2.7. Arc (6,5) belongs into set I_- and it has the same orientation as the entering arc (1,6). So, it is $\theta_1 = -x_{65} = 5$. Arc (1,5) belongs into set I_+ and has the opposite direction. So, it is $\theta_2 = x_{15} = 3$. Again we have $\theta_1 > \theta_2$, i.e a type B iteration. Arc (1,5) is the leaving arc.

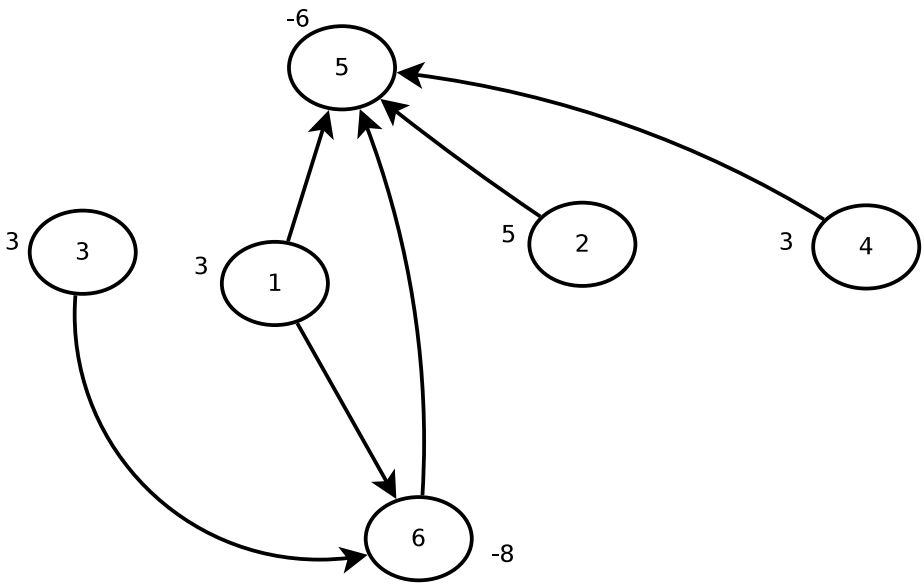

Figure 2.7: Cycle created for the 2nd iteration

Step 4 (Pivot)

After removing the leaving arc (1,5) from T and adding the entering arc (1,6) into it, the tree shown in Figure 2.8, is created. After updating the flows vector x we take: $x_{16} = 3, x_{25} = 5, x_{36} = 3, x_{45} = 3$ and $x_{65} = -2$. For the reduced cost variables it is: $s_{15} = -26, s_{26} = 48, s_{35} = -24, s_{46} = 53, s_{53} = 202, s_{64} = 31$ and $s_{63} = 173$. Again only set I_+ changes and it becomes $I_+ = \{(3,6), (1,6), (2,5), (4,5)\}$.

A primal feasible tree-solution has not been found yet, so the algorithm continues with its third iteration:

Iteration 3

Step 1 (Test of optimality)

It is $I_- \neq \emptyset$, so the current tree-solution is not optimal. From Relation (2.8), we find $J_- = \{(2,6), (4,6), (5,3)\}$. It is $J_- \neq \emptyset$, so DNEPSA continues to step 2.

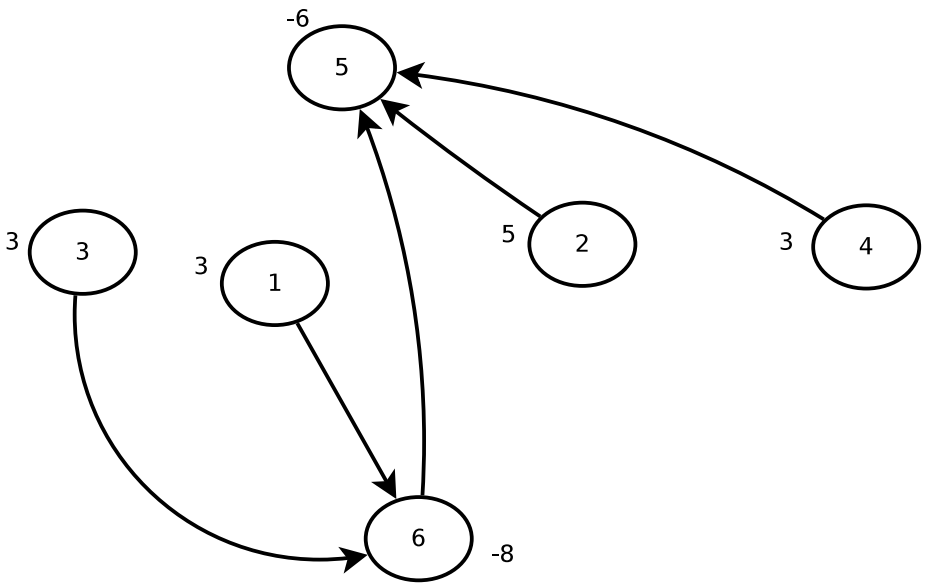

Figure 2.8: The basic tree after the second iteration

Step 2 (Selection of the entering arc)

From Relation (2.9), we find:

$$\alpha = \frac{s_{26}}{-d_{26}} = \frac{48}{1} = min\{\frac{48}{1}, \frac{53}{1}, \frac{202}{1}\}$$

Therefore, arc (2,6) is the entering arc.

Step 3 (Selection of the leaving arc)

After adding the entering arc (2,6) into the basic tree T, a cycle C is created, as shown in Figure 2.9. Arc (6,5) belongs into set I_- and it has the same orientation as the entering arc (1,6). So, it is $\theta_1 = -x_{65} = 2$. Arc (2,5) belongs into set I_+ and has the opposite direction. So, it is $\theta_2 = x_{25} = 5$. It is $\theta_1 \leq \theta_2$, i.e we have a type A iteration and arc (6,5) is leaving the basic tree-solution.

Step 4 (Pivot)

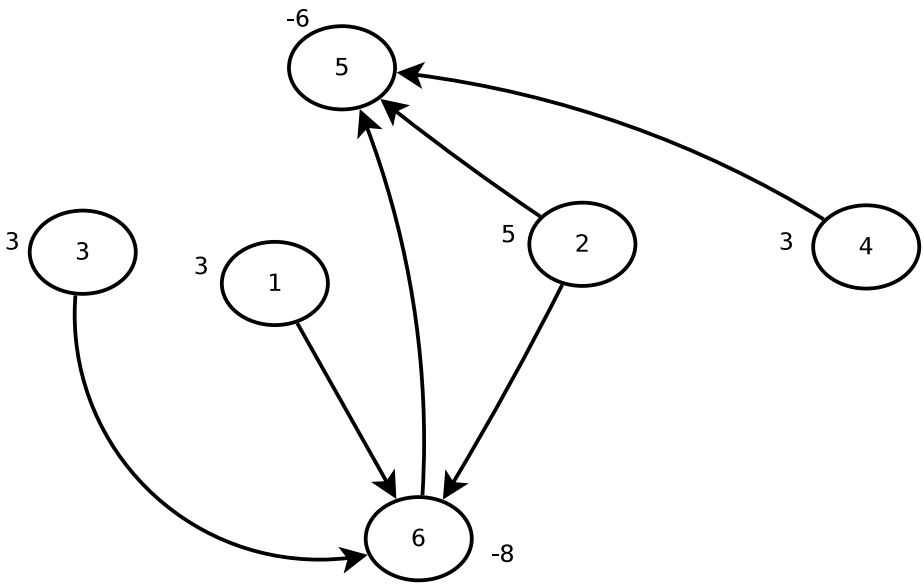

Figure 2.9: Cycle created for the 3rd iteration

After removing the leaving arc (6,5) from T and adding the entering arc (2,6) into it, the tree shown in Figure 2.10, is created. After updating the flows vector x, we take: $x_{16} = 3, x_{25} = 3, x_{36} = 3, x_{45} = 3$ and $x_{26} = 2$. We observe here that, the tree-solution found is primal feasible, that is $x_{ij} \geq 0, \forall (i,j) \in T$. As it is theoretically proved in chapter 3, when DNEPSA comes to a primal feasible tree-solution, that tree is also dual feasible and therefore, it is an optimal solution. Indeed, the new tree-solution is also dual feasible, since it is: $s_{15} = 22, s_{35} = 24, s_{46} = 5, s_{53} = 154, s_{64} = 79, s_{65} = 48$ and $s_{63} = 173$. Therefore, after three iterations, the algorithm has found an optimal solution. It is $I_{-} = \emptyset$, so the algorithm stops.

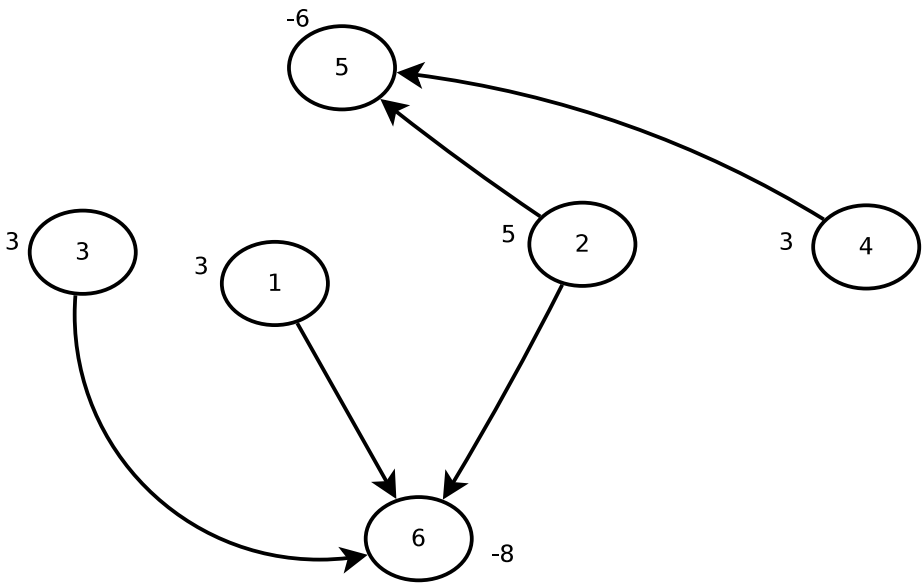

Figure 2.10: The optimal basic tree-solution found after the third iteration

2.4.1 An infeasible problem

The algorithm will now be applied to the network G = (N, A) of Figure 2.11 that consists of 5 nodes and 10 arcs. We have the following iterations:

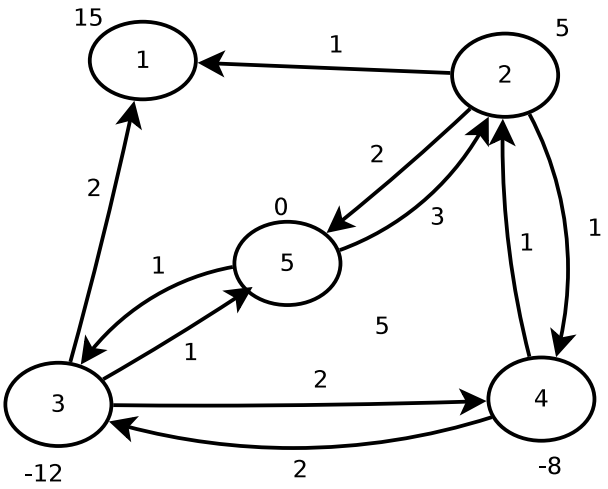

Figure 2.11: An infeasible problem

Iteration 1

Step 0 (Initialization)

Figure 2.12 shows a dual feasible tree-solution T that will be used by DNEPSA as a starting point. Again, such an initial solution can be obtained by using the technique described in section 4.3.

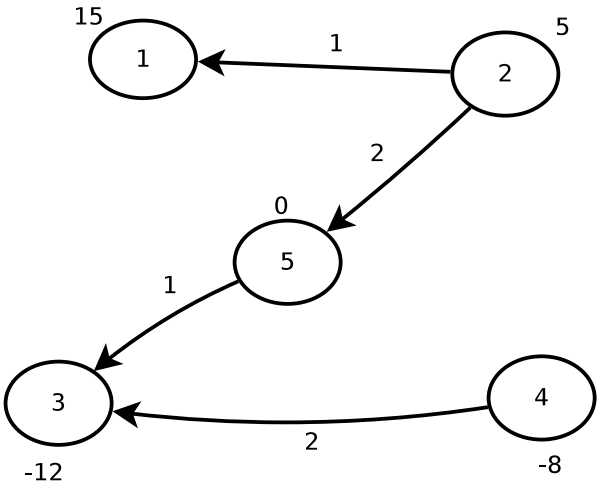

Figure 2.12: Initial dual feasible tree-solution for the infeasible problem

That tree-solution is a dual feasible tree, since for the reduced cost variables it is $s_{ij} \geq 0, \forall (i,j) \in T$. This can be easily verified by using Relation (2.1), which takes the following forms:

$$\begin{aligned} w_1 - w_2 &= 1 \\ w_5 - w_2 &= 2 \\ w_3 - w_5 &= 1 \\ w_3 - w_4 &= 2 \end{aligned} \tag{2.15}$$

By setting arbitrarily the value of w_1 equal to 0, from Relation (2.15), we can very easily find $w_2 = -1, w_3 = 2, w_4 = 0$ and $w_5 = 1$.

By using Relation (2.2), the reduced cost values for all non-basic arcs $(i, j) \notin T$ are:

$$s_{31} = c_{31} + w_3 - w_1 = 2 + 2 - 0 = 4$$
$$s_{24} = c_{24} + w_2 - w_4 = 1 + (-1) - 0 = 0$$
$$s_{42} = c_{42} + w_4 - w_2 = 1 + 0 - (-1) = 2$$
$$s_{52} = c_{52} + w_5 - w_2 = 3 + 1 - (-1) = 5$$
$$s_{34} = c_{34} + w_3 - w_4 = 2 + 2 - 0 = 4$$
$$s_{35} = c_{35} + w_3 - w_5 = 1 + 2 - 1 = 2$$

For the basic arcs $(i, j) \in T$, it is $s_{ij} = 0$.

For all nodes $i \in N$ of the graph, the outgoing flow has to be equal to the incoming flow plus the supply of the node. Therefore, it is:

$$\sum_{(i,k)\in N} x_{ik} - \sum_{(j,i)\in N} x_{ji} = b_i$$

and thus we have:

$$node1 : -x_{21} = 15$$
$$node2 : x_{21} + x_{25} = 5$$
$$node3 : -x_{53} - x_{43} = -12$$
$$node4 : x_{43} = -8$$
$$node5 : x_{53} - x_{25} = 0$$

By solving the above equations we can easily compute the flows x for the initial tree-solution T: $x_{21} = -15, x_{25} = 20, x_{43} = -8$ and $x_{53} = 20$. This is not a feasible tree, since there exist negative flows.

We have $I_- = \{(2, 1), (4, 3)\}$ and $I_+ = \{(2, 5), (5, 3)\}$.

By using Relation (2.7), we find: $d_{31} = 1, d_{24} = -1, d_{42} = 1, d_{52} = 0, d_{34} = -1$ and $d_{35} = 0$.

Step 1 (Test of optimality)

It is $I_- \neq \emptyset$, so the current tree-solution is not optimal. From Relation (2.8), the algorithm creates set $J_- = \{(2,4),(3,4)\}$. It is $J_- \neq \emptyset$, so DNEPSA continues to step 2.

Step 2 (Selection of the entering arc)

From Relation (2.9), the algorithm computes ratio α:

$$\alpha = \frac{s_{24}}{-d_{24}} = min\{\frac{s_{24}}{-d_{24}}, \frac{s_{34}}{-d_{34}}\} = min\{\frac{0}{1}, \frac{4}{1}\} = 0$$

So, arc (2,4) is the entering arc.

Step 3 (Selection of the leaving arc)

After adding the entering arc (2,4) into the basic tree-solution T, a cycle C is created, as it is shown in Figure 2.13. That cycle contains three basic arcs: (2,5) (5,3) and (4,3). Arc (4,3) belongs into set I_- and it has the same orientation as the entering arc (2,4). So, it is $\theta_1 = -x_{43} = 8$. Arcs (2,5) and (3,5), on the other hand, belong into set I_+ and they do not have the same orientation as the entering arc (2,4). So, it is $\theta_2 = min\{x_{25}, x_{35}\} = min\{20, 20\} = 20$. We have $\theta_1 < \theta_2$ which means that we have a type A iteration and arc (4,3) is the leaving arc.

Step 4 (Pivot)

After removing the leaving arc from the basic tree-solution T and adding the entering arc into it, a new tree, shown in Figure 2.14, is produced. Vectors x, s and d has to be updated according to Tables (2.2), (2.4) and (2.5). The

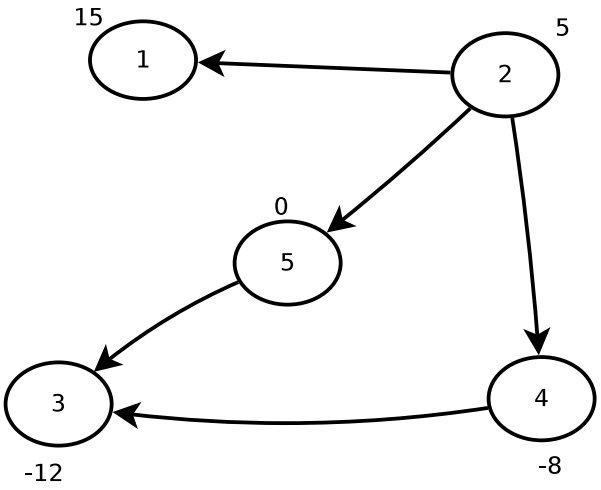

Figure 2.13: Cycle created when adding the entering arc into the basic tree

flow for the entering arc is $x_{24} = \theta_1 = 8$ and for the arcs $(2, 5)$ and $(5, 3) \in C$, it becomes $x_{25} = 20 - 8 = 12$ and $x_{53} = 20 - 8 = 12$. The rest of the flows remain unchanged.

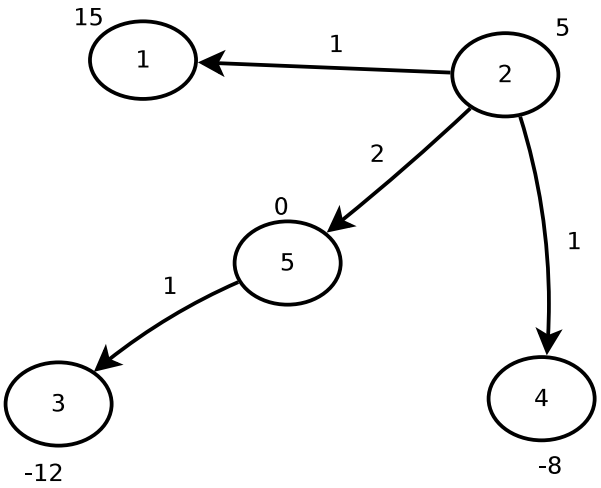

Figure 2.14: The basic tree after the first iteration

When the leaving arc (2,4) is removed from the original tree T, then subtrees T^+ and T^-, shown in Figure 2.14, are created. Only the reduced costs for the arcs that cross from one subtree to the other are changed, the leaving

arc included. Those arcs, as Table 2.4 implies, change their reduced cost value by adding or subtracting value of s_{24}, depending on their orientation. Since it is $s_{24} = 0$, the reduced cost values will remain unchanged. In a similar way, as Table 2.5 implies, the values of d_{34}, d_{43} and d_{42} change by an amount of $d_{24} = -1$. So, it becomes: $d_{34} = -1-(-1) = 0$, $d_{43} = 1+(-1) = 0$ and $d_{42} = 1 + (-1) = 0$.

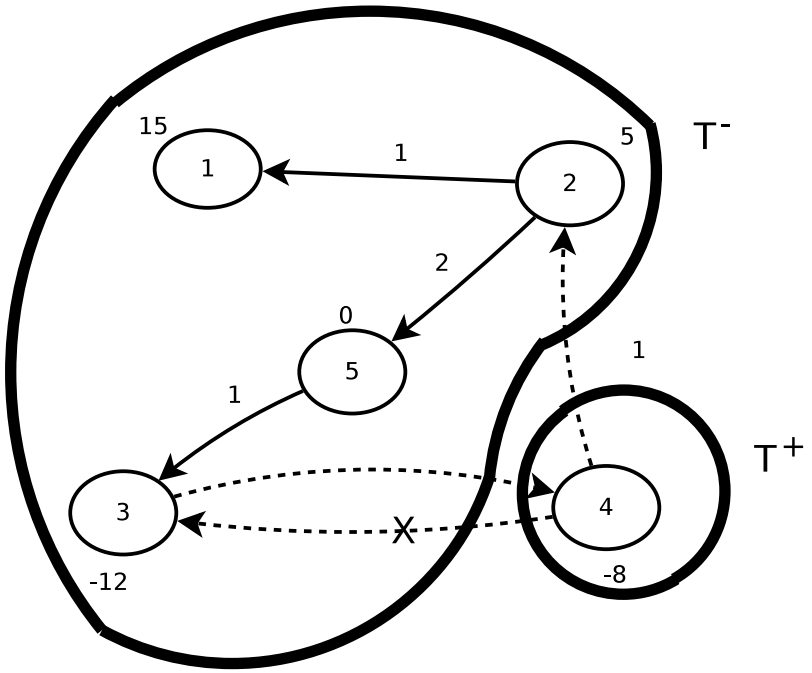

Figure 2.15: The subtrees created after the removal of the leaving arc

The algorithm performed a type A iteration, and it becomes $I_+ = I_+ \cup \{(2,4)\} = \{(2,5),(5,3),(2,4)\}$ and $I_- = I_- \setminus \{(4,3)\} = \{(2,1)\}$.

The algorithm continues to the second iteration for the new basic tree-solution (Figure 2.14), starting from Step 1.

Iteration 2

Step 1 (Test of optimality)

It is $I_{\_} \neq \emptyset$, so the current tree-solution is not optimal. From Relation (2.8), it comes out that $J_{\_} = \emptyset$. So, the algorithms comes into the conclusion that the problem is infeasible.

Chapter 3

Mathematical Proof of Correctness

3.1 Introduction

In this chapter, the analytical proof of correctness for the DNEPSA algorithm will be presented through a number of theorems. All the theorems are mathematically proved in section 3.2, under the assumption that the MCNFP problem is not degenerate. Although the assumption made, a practical method to avoid the bad results due to degeneracy (stalling and cycling), was applied during the implementation of DNEPSA. More precisely, when DNEPSA has to choose between two or more equally qualified arcs for the selection of the leaving or the entering arc, a technique is applied that is similar to the *Bland's rule* for the general linear programming problem. This method is described in more detail in section 4.1.1. In the next section the theorems needed to prove the correctness of the algorithm are given and

theoretically proved.

3.2 Theorems

The first theorem states that the value of the objective function z increases strictly from iteration to iteration (for a non degenerate pivot). This conclusion is used in Theorem 3.2, in order to prove that the algorithm terminates after a finite number of iterations.

Theorem 3.1. *The value of the objective function z increases strictly from iteration to iteration.*

Proof. Let $z^{(t)}$ be the value of the objective function in iteration t. We will prove that the increase in the objective value is equal to $\Delta z = z^{(t+1)} - z^{(t)} > 0$.

As it is shown in Relation (2.10), for the flow of a product unit, the objective function value changes by $\Delta z = s_{gh}$. For a type A iteration, as it is described in section 2.2, an arc (k,l) of negative flow is leaving the basic tree T. In that case, the flow on the entering arc (g, h) is equal to $x_{gh}^{(t+1)} = \theta_1 = -x_{kl}^{(t)} > 0$ (see Table 2.2). That is, the total change in the objective value is equal to $\Delta z = x_{gh}^{(t+1)} s_{gh} = \theta_1 s_{gh} > 0$ (it is $s_{gh} > 0$ since $(g, h) \in J_{-}$, see Relation (2.8)).

Similarly, for a type B iteration, an arc of positive flow is leaving the basic tree and the flow on the entering arc (g, h) is equal to $x_{gh}^{(t+1)} = \theta_2 = x_{kl} > 0$. The objective value is now changing by $\Delta z = \theta_2 s_{gh} > 0$. Therefore it is always $\Delta z > 0$. □

Theorem 3.2. *The algorithm terminates after a finite number of iterations.*

Proof. In Theorem 3.1 we proved that the value of the objective function strictly increases from iteration to iteration. That is, no tree-solution will be created twice. The number of trees that can be created for a given network is finite. Therefore, DNEPSA will perform a finite number of iterations before it terminates. □

In the next theorem, we prove that after each iteration, the set I_- contains the basic arcs (i, j) of negative flow while the set I_+ contains those of non-negative flow.

Theorem 3.3. *For all iterations, if* $(i, j) \in I_-$*, then it is* $x_{ij} < 0$ *and if* $(i, j) \in I_+$ *then it is* $x_{ij} \geq 0$.

Proof. The theorem will be proved by using mathematical induction. Let's assume we have a type A iteration. For the first iteration, because of their definition, I_- contains the arcs having negative flow, while I_+ contains the arcs of the tree of non negative flow. We assume that it is true for iteration t. We'll prove that it is also true for iteration $t + 1$.

The elements of I_- are updated according to Relation (2.12). Therefore, an arc (i,j) that belongs to I_- during the $(t + 1)^{th}$ iteration, it also belongs to I_- during the t^{th} iteration. Because of the assumption, it is $x_{ij}^{(t)} < 0$. We need to show that it is also $x_{ij}^{(t+1)} < 0$. By examining all the cases shown in Table 2.1 and the way the flows are updated in Table 2.2, we have

- *Case 1*: It is $x_{ij}^{(t+1)} = x_{ij}^{(t)} < 0$ because of the assumption.
- *Case 2*: Cannot hold because $(i, j) = (g, h) \in I_+$.
- *Case 3*: Cannot hold because we assumed type A iteration.

- *Case 4*: It is $\theta_1 = -x_{kl}^{(t)} < -x_{ij}^{(t)}$ because of Relation (2.11). Therefore, it is $x_{ij}^{(t+1)} = x_{ij}^{(t)} - x_{kl}^{(t)} < 0$.

- *Case 5*: It is $x_{ij}^{(t+1)} = x_{ij}^{(t)} + x_{kl}^{(t)} < 0$ since $x_{ij}^{(t)} < 0$ and $x_{kl}^{(t)} < 0$.

The contents of set I_+ are also updated according to Relation (2.12). Arc (g, h) is the only new arc in I_+. We have the following cases:

- *Case 1*: It is $x_{ij}^{(t+1)} = x_{ij}^{(t)} \geq 0$ because of the assumption.

- *Case 2*: It is $x_{ij}^{(t+1)} = x_{gh}^{(t+1)} = \theta_1 = -x_{kl} \geq 0$.

- *Case 3*: Cannot hold because we assumed a type A iteration.

- *Case 4*: It is $x_{ij}^{(t+1)} = x_{ij}^{(t)} - x_{kl}^{(t)} \geq 0$ because $x_{ij}^{(t)} \geq 0$ and $x_{kl}^{(t)} < 0$.

- *Case 5*: It is $x_{ij}^{(t+1)} = x_{ij}^{(t)} + x_{kl}^{(t)}$. It is also $\theta_2 \leq x_{ij}^{(t)}$ because of Relation (2.11) and $\theta_1 \leq \theta_2$ since we have a type A iteration. Therefore, it is $\theta_1 = -x_{kl}^{(t)} \leq x_{ij}^{(t)} \Rightarrow x_{ij}^{(t+1)} \geq 0$.

In a similar way, we can prove for a type B iteration that, I_- contains all the arcs having negative flow and I_+ contains those arcs of non-negative flow. □

The next theorem will give a property for all the non-basic arcs $(i, j) \notin T$ that have negative reduced cost value, i.e. for those arcs having $s_{ij} < 0$. This property will be very important for the proof of the theorems that will follow.

Theorem 3.4. *If for a non-basic arc $(i,j) \notin T$ it is $s_{ij} < 0$, then it is $d_{ij} > 0$ and we also have:*

$$\frac{s_{ij}}{-d_{ij}} < \alpha = \frac{s_{gh}}{-d_{gh}}$$

Proof. We'll first prove, by using mathematical induction, the first part of the theorem, i.e. that if it is $s_{ij} < 0$ then it is $d_{ij} > 0$. During the first iteration, we don't have arcs having negative reduced cost, since the algorithm starts from a dual feasible solution. Let $k+1$ be the first iteration for which we have $s_{ij}^{(k+1)} < 0$ for an arc (i,j), while it is $s_{ij}^{(k)} \geq 0$. According to Table 2.4, in order to have a negative value for the reduced cost value, it has to be $s_{ij}^{(k+1)} = s_{ij}^{(k)} - s_{gh}^{(k)}$.

So, we have:

$$s_{ij}^{(k+1)} < 0 \Rightarrow s_{ij}^{(k)} - s_{gh}^{(k)} < 0 \Rightarrow s_{ij}^{(k)} < s_{gh}^{(k)}. \tag{3.1}$$

If it is $d_{ij}^{(k)} \geq 0$, then it is obviously $d_{ij}^{(k+1)} = d_{ij}^{(k)} - d_{gh}^{(k)} > 0$, since it is $d_{gh}^{(k)} < 0$ (because $(g,h) \in J_{-}$, see section 2.2). If, on the other hand, it is $d_{ij}^{k} < 0$, then we have:

$$\alpha = \frac{s_{gh}^{(k)}}{-d_{gh}^{(k)}} \leq \frac{s_{ij}^{(k)}}{-d_{ij}^{(k)}} \Rightarrow \frac{-d_{ij}^{(k)}}{s_{ij}^{(k)}} \leq \frac{-d_{gh}^{(k)}}{s_{gh}^{(k)}}. \tag{3.2}$$

By multiplying equations (3.1) and (3.2) together, we take:

$$-d_{ij}^{(k)} < -d_{gh}^{(k)} \Rightarrow d_{ij}^{(k)} - d_{gh}^{(k)} > 0 \Rightarrow d_{ij}^{(k+1)} > 0$$

In Relation (3.2), we made silently the assumption that if $d_{ij}^{k} < 0$, then it can't be $s_{ij}^{k} = 0$. This can be proved as follows: Let's assume that in k

iteration it is $s_{ij}^k = 0$, while $s_{ij}^{(k-1)} > 0$. It has to be:

$$s_{ij}^k = s_{ij}^{(k-1)} - s_{gh}^{(k-1)} \Rightarrow s_{ij}^{(k-1)} - s_{gh}^{(k-1)} = 0 \Rightarrow s_{ij}^{(k-1)} = s_{gh}^{(k-1)} \tag{3.3}$$

It is also

$$d_{ij}^k = d_{ij}^{(k-1)} - d_{gh}^{(k-1)} \Rightarrow d_{ij}^{(k-1)} < d_{ij}^k$$

since it is $d_{gh}^{(k-1)} < 0$. So, we have the following two cases:

- *Case 1*: If it is $d_{ij}^{(k-1)} \geq 0$, then it is $d_{ij}^k > 0$.
- *Case 2*: If it is $d_{ij}^{(k-1)} < 0$, then

$$\alpha = \frac{s_{gh}^{(k-1)}}{-d_{gh}^{(k-1)}} \leq \frac{s_{ij}^{(k-1)}}{-d_{ij}^{(k-1)}}$$

and because of (3.3)

$$\frac{1}{-d_{gh}^{(k-1)}} \leq \frac{1}{-d_{ij}^{(k-1)}}$$

and since $d_{gh}^{(k-1)} d_{ij}^{(k-1)} > 0$

$$-d_{ij}^{(k-1)} \leq -d_{gh}^{(k-1)} \Rightarrow d_{ij}^{(k-1)} - d_{gh}^{(k-1)} \geq 0 \Rightarrow d_{ij}^{(k)} \geq 0$$

Therefore, if it comes to be $s_{ij}^{(k)} = 0$, it is $d_{ij}^k \geq 0$. So, it has been proved that, starting from a dual feasible solution, for the first iteration it comes to

be $s_{ij} < 0$, it is $d_{ij} > 0$, even if in the previous iteration it is $s_{ij} = 0$.

Assume now that the theorem holds for the t iteration . For the iteration $t+1$, according to the Tables 2.3 and 2.4, we have the following cases:

- *Case 1*: Obviously, for this case, the theorem holds since it is $d_{ij}^{(t+1)} = d_{ij}^{(t)}$ and $s_{ij}^{(t+1)} = s_{ij}^{(t)}$.

- *Cases 2 and 5*: It is:

$$s_{ij}^{(t+1)} = s_{ij}^{(t)} + s_{gh}^{(t)} < 0 \Rightarrow s_{gh}^{(t)} < -s_{ij}^{(t)}. \tag{3.4}$$

  Because of the assumption, it is:

$$\frac{s_{ij}^{(t)}}{-d_{ij}^{(t)}} < \alpha = \frac{s_{gh}^{(t)}}{-d_{gh}^{(t)}}$$

  and since it is $-d_{gh}^{(t)} d_{ij}^{(t)} > 0$, we have

$$d_{gh}^{(t)} s_{ij}^{(t)} < d_{ij}^{(t)} s_{gh}^{(t)} \tag{3.5}$$

  By multiplying (3.4) and (3.5) together, we have:

$$s_{gh}^{(t)} d_{gh}^{(t)} s_{ij}^{(t)} < -s_{ij}^{(t)} d_{ij}^{(t)} s_{gh}^{(t)}$$

  and by dividing with $-s_{ij}^{(t)} s_{gh}^{(t)} > 0$, we take

$$-d_{gh}^{(t)} < d_{ij}^{(t)} \Rightarrow d_{ij}^{(t)} + d_{gh}^{(t)} > 0 \Rightarrow d_{ij}^{(t+1)} > 0$$

- *Cases 3 and 4*: It is $d_{ij}^{(t+1)} = d_{ij}^{(t)} - d_{gh}^{(t)} > 0$, since we have $d_{ij}^{(t)} > 0$ and $d_{gh}^{(t)} < 0$.

We'll prove now the second part of the theorem by using again mathematical induction. We first prove that it is

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} < \alpha^{(t)}$$

and after that, we prove that it is

$$\alpha^{(t)} \leq \alpha^{(t+1)}$$

Let's assume that the above relations hold for iteration t and we will show that they also hold for iteration $t+1$. For case 1 of Table 2.3, it is obviously

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} < \alpha^{(t)}$$

For the rest of the cases it is:

$$\frac{s_{ij}^{(t)}}{-d_{ij}^{(t)}} < \frac{s_{gh}^{(t)}}{-d_{gh}^{(t)}}$$

and multiplying by $-d_{gh}^{(t)} d_{ij}^{(t)} > 0$, we have

$$d_{gh}^{(t)} s_{ij}^{(t)} < d_{ij}^{(t)} s_{gh}^{(t)} \Rightarrow d_{ij}^{(t)} > d_{gh}^{(t)} \frac{s_{ij}^{(t)}}{s_{gh}^{(t)}} \tag{3.6}$$

We also have:

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} = \frac{s_{ij}^{(t)} \pm s_{gh}^{(t)}}{-(d_{ij}^{(t)} \pm d_{gh}^{(t)})}. \tag{3.7}$$

From Relations (3.6) and (3.7) we take:

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} < \frac{s_{ij}^{(t)} \pm s_{gh}^{(t)}}{-(d_{gh}^{(t)} \frac{s_{ij}^{(t)}}{s_{gh}^{(t)}} \pm d_{gh}^{(t)})} = -\frac{s_{gh}^{(t)}(s_{ij}^{(t)} \pm s_{gh}^{(t)})}{d_{gh}^{(t)}(s_{ij}^{(t)} \pm s_{gh}^{(t)})} = -\frac{s_{gh}^{(t)}}{d_{gh}^{(t)}} = \alpha^{(t)}$$

We will now show that

$$\alpha^{(t)} \leq \alpha^{(t+1)}$$

Assume that this is true for iteration t. For the case 1 of Table 2.3, it is obviously true. For the rest of the cases, it is:

$$\frac{s_{ij}^{(t)}}{-d_{ij}^{(t)}} \geq \alpha^{(t)} = \frac{s_{gh}^{(t)}}{-d_{gh}^{(t)}}$$

and by multiplying with $-d_{ij}^{(t)} > 0$, we have:

$$s_{ij}^{(t)} \geq \frac{d_{ij}^{(t)} s_{gh}^{(t)}}{d_{gh}^{(t)}} \tag{3.8}$$

It is also:

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} = \frac{s_{ij}^{(t)} \pm s_{gh}^{(t)}}{-(d_{ij}^{(t)} \pm d_{gh}^{(t)})} \tag{3.9}$$

From Relations (3.8) and (3.9) we take:

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} \geq \frac{d_{ij}^{(t)} \frac{s_{gh}^{(t)}}{d_{gh}^{(t)}} \pm s_{gh}^{(t)}}{-(d_{ij}^{(t)} \pm d_{gh}^{(t)})} = \frac{s_{gh}^{(t)}(d_{ij}^{(t)} \pm d_{gh}^{(t)})}{-d_{gh}^{(t)}(d_{ij}^{(t)} \pm d_{gh}^{(t)})} = \frac{s_{gh}^{(t)}}{-d_{gh}^{(t)}} = \alpha^{(t)}$$

We have proved that, for every case, it is $\alpha^{(t)} \leq \alpha^{(t+1)}$ and therefore, it is

$$\frac{s_{ij}^{(t+1)}}{-d_{ij}^{(t+1)}} < \alpha^{(t+1)}$$

□

The algorithm keeps in touch with the dual feasible region by maintaining a direction vector d. The next theorem proves that it is $s_{ij} + \alpha\, d_{ij} \geq 0$, for every arc (i, j). Therefore, the vector $y = s + \alpha d$ always gives a dual feasible solution.

Theorem 3.5. *Solution $y = s + \alpha d$ is dual feasible during all iterations of the algorithm.*

Proof. If it is $s_{ij} < 0$, then according to Theorem 3.4, it is $d_{ij} > 0$ and

$$\frac{s_{ij}}{-d_{ij}} < \alpha$$

Therefore, it is

$$s_{ij} + \alpha d_{ij} > 0$$

If it is $s_{ij} \geq 0$ and $d_{ij} < 0$ then, because of Relation (2.9), it is

$$\frac{s_{ij}}{-d_{ij}} \geq \alpha$$

Therefore, it is again $s_{ij} + \alpha d_{ij} \geq 0$.

Finally, if it is $s_{ij} \geq 0$ and $d_{ij} \geq 0$, then it is obviously $s_{ij} + \alpha d_{ij} \geq 0$, since $\alpha > 0$. □

The next theorem examines the case where the algorithm decides that the problem is infeasible.

Theorem 3.6. *If it is $J_- = \emptyset$ and $I_- \neq \emptyset$, then the problem in infeasible.*

Proof. As it was shown in Theorem 3.5, $y = s + \alpha d$ is always dual feasible. Therefore, it satisfies the restrictions of the dual problem as it is described in matrix format in Relation (1.6). So, we have:

$$A^T w + I_m(s + \alpha d) = c \tag{3.10}$$

We denote as A_B^T the matrix formed by the rows of matrix A^T that correspond to the basic variables. Vectors c_B, s_B, and d_B are formed in a similar way. It is as follows:

$$A_B^T w + (s_B + \alpha d_B) = c_B \tag{3.11}$$

By multiplying both parts of equation (3.11) with $b^T(A_B^T)^{-1}$ we take:

$$b^T(A_B^T)^{-1}A_B^T w + b^T(A_B^T)^{-1}(s_B + \alpha d_B) = b^T(A_B^T)^{-1}c_B$$

that becomes:

$$b^T w = b^T(A_B^T)^{-1}c_B - b^T(A_B^T)^{-1}(s_B + \alpha d_B) \tag{3.12}$$

For the basic solution x_B it is:

$$A_B x_B = b \Rightarrow x_B^T = b^T(A_B^T)^{-1}$$

so, Relation (3.12) becomes

$$b^T w = x_B^T c_B - x_B^T (s_B + \alpha d_B) = x_B^T c_B - \alpha x_B^T d_B$$

and since $s_B = 0$, we have:

$$b^T w = x_B^T c_B - \alpha x_B^T d_B \tag{3.13}$$

If we denote by z and z' the value of the objective function of the primal and the dual problem respectively, then the Relation (3.13) takes the following form:

$$z' = z - \alpha x_B^T d_B. \tag{3.14}$$

We have $d_{ij} = 1$ for the negative flows and $d_{ij} = 0$ for the non-negative flows (Relation (2.7)). There is at least one negative flow because $I_- \neq \emptyset$ (otherwise the algorithm would have stopped). Therefore, it is $x_B d_B < 0$ and as it can be seen in Relation (3.14), the objective function of the dual problem is unbounded because it increases as far as the value of α increases. The fact that the dual problem is unbounded, means that the primal problem in infeasible. □

The last theorem proves that the algorithm has reached an optimal solution when $I_- = \emptyset$.

Theorem 3.7. *If $I_- = \emptyset$ then the current solution is optimal.*

Proof. It is obvious from Relation (2.7) that

$$-|I_-| \leq d_{ij} \leq |I_-|$$

where the notation $|I_-|$ means the cardinality of the set I_-. If $I_- = \emptyset$, then it is $|I_-| = 0$ and therefore we have $d_{ij} = 0, \forall (i,j) \in A$. According to Theorem 3.5, it is $y = s + \alpha d \geq 0$, so we have $s \geq 0$. At the same time, it is $x \geq 0$, since we have $I_- = \emptyset$. Therefore, the current tree-solution is both primal and dual feasible and thus we can conclude that it is optimal. $\square$

Chapter 4

Implementation and Computational Results

4.1 Implementation of DNEPSA

The Dual Network Exterior Point Simplex-type Algorithm was implemented and run for numerous Minimum Cost Network Flow Problem instances. It was tested on MCNFP instances made "by hand" and it was also tested on MCNFP instances generated by random network generators, like NETGEN and GRIDGEN (see [59] and [61]). A brief description of the NETGET network generator is given in section 4.1.2.

DNEPSA needs an initial dual feasible tree-solution to start from. The method presented by Hultz and Klingman and described in section 4.3 was used. Degeneracy problems (cycling abd stalling) were handled in the way described in section 4.1.1.

The data structures of the algorithm for storing graphs and trees, were

implemented by using the *Augmented Thread Index method* (ATI method), due to Glover et al. (see [41]). This method was selected because it allows the fast update of the basic tree after each iteration of the algorithm. It can also easily identify the cycle created after the addition of the entering arc and check the orientations of the arcs in that cycle. In section 4.2 the data structures used in the implementation of the algorithm are described.

4.1.1 Degeneracy

When running a Simplex-type algorithm, *degenerate pivots* may occur for two or more iterations. A pivot is called degenerate when, after the pivot is applied, the objective value for the new tree-solution is exactly the same as the objective value of the previous tree-solution. Degenerate pivots may occur when there are *degenerate basic tree-solutions*, i.e. tree-solutions containing at least one arc of zero flow. In that case, we say we have a *degenerate problem*.

Degenerate pivots may occur one after the other. In that case, a Simplex-type algorithm may lead to *cycling*. Cycling occurs when starting from a basic solution, after a number of iterations, the same basic solution is revisited. So, when cycling occurs, the algorithm's steps are repeated endlessly. The possibility of cycling was recognized shortly after the first presentation of the Simplex method. Some examples of cycling are presented in [52] and [10]. Although cycling was one of the first problems encountered in Linear Programming, it still remains one of the most basic problems (see [51] and [36]). Some pathological examples for Minimum Cost Network Flow prob-

lems, where the standard PNSA algorithms falls into cycling, are given in [93] and [94].

Cunningham showed that by using a specific type of basis, known as *strongly feasible basis*, cycling can be avoided (see [23] and [24]). A basic tree-solution, for the uncapacitated MCNFP problem, is said to be strongly feasible, if all the basic arcs with zero flow are pointing toward the root of the tree. So, it is enough to modify the rules for the selection of the leaving and the entering arc in order to maintain strongly feasible basic trees, as it is described in [15]. Despite the rules for the leaving and the entering arc, it is always possible to construct pathological problems where cycling occurs (see [95]). In practice, cycling is not happening very often except of cases of strongly degenerate problems. So, in many cases, it is not worthy the extra work needed to apply complicated rules in order to choose leaving and entering arcs during the iterations.

Another problem, that may appear when applying a Simplex-type algorithm, is *stalling*. Stalling in a Simplex-type algorithm is defined as an exponentially long sequence of consecutive degenerate pivots without cycling. Several anti-stalling pivot rules have been proposed by researchers, like the rules described in [24], [62] and [4].

The phenomenon of having, during an iteration, more than one equally qualified choices for the leaving and the entering arc, is usually called a *tie*. DNEPSA is implemented in such a way that it tries to resolve ties by giving a numbering (index) to every arc of the graph. So, the algorithm breaks the ties by always choosing the arc with the minimum index between all the equally qualified leaving or entering arcs. This method, used by DNEPSA

to handle degeneracy, resembles the method introduced by Bland in [15]. For the randomly generated problems where DNEPSA was widely tested, problems like cycling or stalling were not observed. It has not theoretically proved though, that such phenomena could not appear in some pathological cases.

4.1.2 The NETGEN network generator

The computational results presented in section 4.4, were collected after the competetive algorithms were run on a number of different MCNFP instances. These instances were randomly generated by using the NETGEN network generator that was presented in 1974 by Klingman et al (see [59]). *NETGEN* is a well-known network generator that can randomly produce instances for the Minimum Cost Network Flow Problem (MCNFP) and also instances of other problems like the transportation problem, the assignment problem etc. There are more network generators, like GRIDGEN, MESH and GRIDGRAPH, that alternatively can be used for the random generation of MCNFP instances.

NETGEN was selected because it is one of the most widely used. It was first developed in FORTRAN programming language but there exists a version of NETGEN in C programming language. Table 4.1 gives an example of the values the parameters of NETGEN can have. These parameters, in C programming language, are determined by a *define* preprocessor directive. On the right side of each directive there is a comment describing its operation.

The parameters NODES and DENSITY refer to the number of nodes and

arcs that the network must have. The parameters SOURCES and SINKS determine how many nodes will be source nodes and how many will be sink nodes. The rest of them will be the transshipment nodes that are determined by directives TSOURCES and TSINKS. The total supply for the source nodes is given by parameter SUPPLY. This supply is divided randomly into the source nodes of the network. In our example, the network will have 10 nodes and 30 arcs. There will be 4 source nodes, 4 sink nodes and 2 transshipment nodes. The total supply is 200.

The minimum and maximum costs for the arcs of the network are determined by the MINCOST and MAXCOST parameters. Directive HICOST gives the percentage of the arcs that will have a cost of MAXCOST. For an uncapacitated MCNFP problem, the parameter CAPACITATED must be equal to 0, while for a capacitated network it must be equal to 1 and the minimum and maximum capacity of the arc have to be determined. Finally, the value of SEED parameter is used as the seed for the random numbers generated. When the seed changes, different numbers are produced when the algorithm runs again. In other word for different seed a different instance of the MCNFP problem is generated.

If it is $SOURCES+SINKS = NODES$ and $TSOURCES = TSINKS = 0$, then an instance of the transportation problem is generated, while if it is $SOURCES = SINKS$ and $SUPPLY = SOURCES$ then an instance of the assignment problem is generated. It becomes very clear here that the transportation and the assignment problem are special cases of the more general Minimum Cost Network Flow Problem (MCNFP).

Table 4.1: Example of NETGEN parameters

#define	PROBLEM_PARMS	13	No of parameters (i.e. the number of lines below)
#define	NODES	10	No of nodes
#define	DENSITY	30	No of arcs
#define	SOURCES	4	number of sources
#define	SINKS	4	number of sinks
#define	MINCOST	0	minimum cost of arcs
#define	MAXCOST	30	maximum cost of arcs
#define	SUPPLY	200	total supply
#define	TSOURCES	1	transshipment sources
#define	TSINKS	1	transshipment sinks
#define	HICOST	10	percent of skeleton arcs given maximum cost
#define	CAPACITATED	0	percent of arcs to be capacitated
#define	MINCAP	0	minimum capacity for capacitated arcs
#define	MAXCAP	0	maximum capacity for capacitated arcs
#define	SEED	1001992789	seed for random graph generation

4.2 Data Structures

For the implementation of the algorithm the *Augmented Thread Index* method (ATI method), presented in [41], was used. The method was further improved in [42], [43] and [9] and it was finally referred as the *Extended Thread Index* method (XTI method). The efficiency of the method is examined in different works, like in [44]. The ATI method offers fast update of the basic tree and easy cycle detection when an arc is added into the basic tree.

For the storing of a rooted tree T, a vector p containing the parents for each node in the tree is used. In Figure 4.1 a rooted tree T, containing 17 nodes, is depicted (node 10 is the root of the tree). The parents vector p for

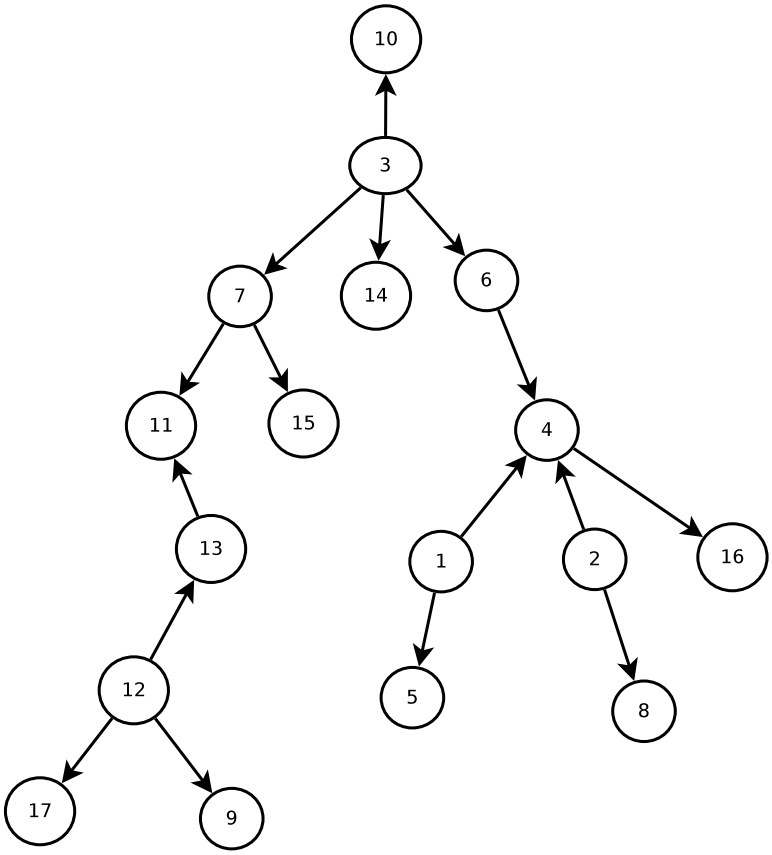

Figure 4.1: A rooted tree stored by using the ATI method.

this tree has the form:

$$p = [4, 4, 10, 6, 1, 3, 3, 2, 12, -1, 7, 13, 11, 3, 7, 4, 12]$$

For every node i of the tree, its parent in T is stored. In our example, it is $p(1) = 4$ because node 4 is the parent of node 1. Similarly, $p(2) = 4$ because node 4 is the parent of node 2 etc. For the root r of the tree T, the value -1 is stored, i.e. $p(r) = -1$. In our case, it is r=10 and therefore, it is $p(10) = -1$.

For a directed tree, it is not enough to store only the parent for each node. Another vector t has to be used in order to store the direction of each arc in

T. That vector t stores 1, if the arc (i,p(i)) belongs to T, i.e. if $(i, p(i)) \in T$ and it stores 0, if $(p(i), i) \in T$. For the root r of the tree, it is $t(r) = -1$, as it shown below:

$$t(i) = \begin{cases} 1 & , if\ (i, p(i)) \in T \\ 0 & , if\ (p(i), i) \in T \\ -1 & .if\ i = r \end{cases}$$

For the tree in Figure 4.1, we have:

$$t = [1, 1, 1, 0, 0, 0, 0, 0, 0, -1, 0, 1, 1, 0, 0, 0, 0]$$

Alternatively, the direction of each arc can be encoded in vector p, by having:

$$p(i) = \begin{cases} j & , if\ (i, p(i)) \in T \\ -j & , if\ (p(i), i) \in T \\ i & , if\ i = r \end{cases}$$

In that case, for the vector of parents p we have:

$$p = [4, 4, 10, -6, -1, -3, -3, -2, -12, 10, -7, 13, 11, -3, -7, -4, -12]$$

The preorder traversal of the nodes of the rooted tree are also stored in a vector named *pr*. For the tree of Figure 4.1, the vector pr has the following form:

$$pr = [10, 3, 7, 11, 13, 12, 17, 9, 15, 14, 6, 4, 1, 5, 2, 8, 16]$$

A vector *dep* that stores the depth of every node of the tree is also used. For

the tree in Figure 4.1, it is:

$$dep = [4, 4, 1, 3, 5, 2, 2, 5, 6, 0, 3, 5, 4, 2, 3, 4, 6]$$

The way these vectors are updated, when the algorithm performs a pivot, will be described now. When the leaving arc (k, l) is removed from the basic tree T, then two subtrees are produced. Let T^* denote the subtree that does not contain the root r of the tree. Only one of nodes k and l belongs to T^*. Let's denote x that note, i.e. it is either $x = k$ (if $k \in T^*$) or $x = l$ (if $l \in T^*$). Similarly, for the entering arc (g, h), only one of nodes g and h belongs to T^*. Let's denote y that note, i.e. it is either $y = g$ (if $g \in T^*$) or $y = h$ (if $h \in T^*$). Both x and y belong to T^*, so there is a path q in T^* that connects node x to node y. Let z denote the vector of nodes that contains the nodes of path q, starting from y and ending in x. If we assume that, for the tree of Figure 4.1, arc $(3, 7)$ is the leaving arc and arc $(1, 13)$ is the entering arc, then the subtree T^*, shown in Figure 4.2, is created and we have $x = 7$ and $y = 13$. Vector $z = [13, 11, 7]$ contains the nodes of the path that connects node x to node y. The arcs that belong in this path are drawn in thick line.

Figure 4.3 shows the new basic tree formed, after removing arc $(3, 7)$ from the tree of Figure 4.1 and adding arc $(1, 15)$ into it. For the new basic tree, the vector of parents p' has the following form:

$$p' = [4, 4, 10, -6, -1, -3, \mathbf{11}, -2, -12, 10, \mathbf{\text{-}13}, 13, \mathbf{\text{-}1}, -3, -7, -4, -12]$$

We can observe here that only the parents for the nodes that belong in path

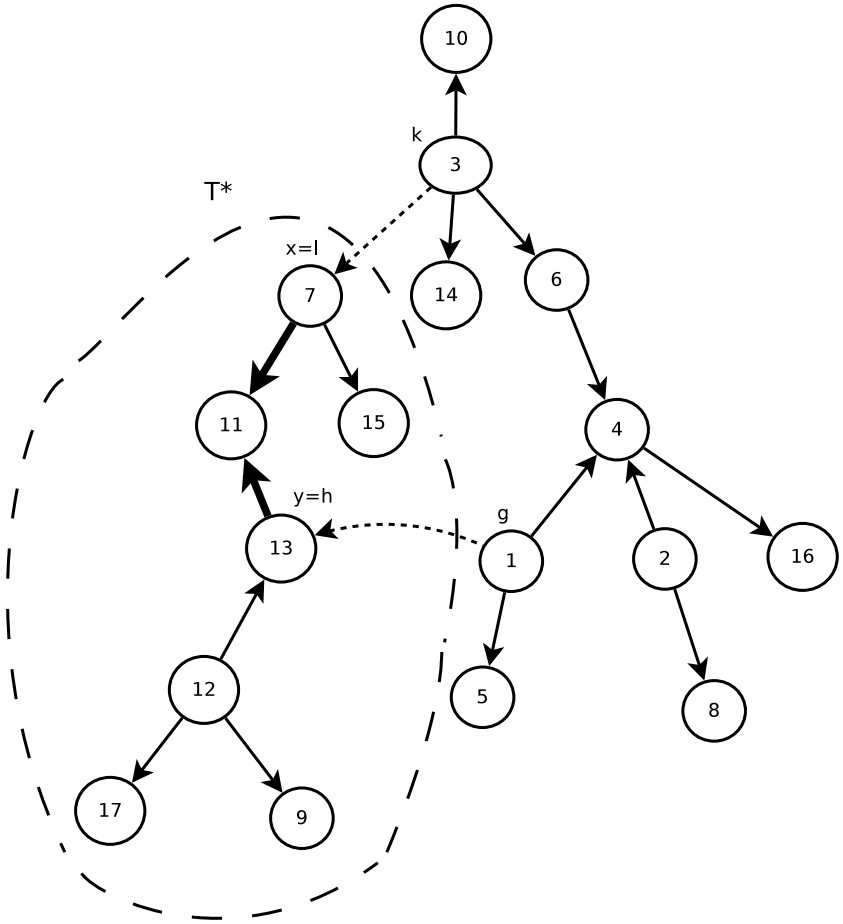

Figure 4.2: Update of the basic tree when using the ATI method.

q have been changed, i.e. in our example, only the parents for nodes 13, 11 and 7 changed (shown in bold in vector p' above). For a node i on path q, its new parent is the node j that precedes it in vector z. The sign for node i in vector p' is the opposite of the sign of node j in vector p. For the first node in z, the role of node j is played by the end of the entering arc (g, h) that is different than y and its sign is negative if it is $y = h$, otherwise it is positive. Therefore, we have $p'(i) = p(j)$, i.e. the parent of node 7 is node 11 and the parent of node 11 is node 13. It is $p'(7) = 11$ because it was $p(11) = -7$ and

$p'(11) = -13$ since $p(13) = 11$. Finally, since $(1, 13)$ is the entering arc, it is $p'(13) = -1$ (we have $y = h = 13$ and $g = 1$).

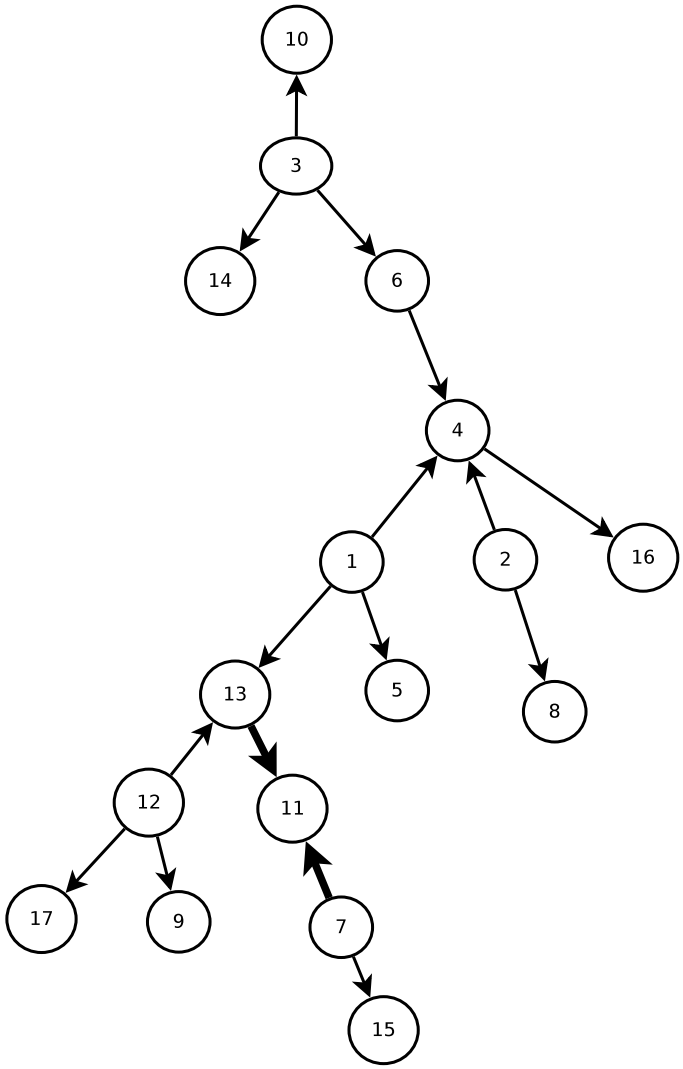

Figure 4.3: Update of the basic tree when using the ATI method.

4.3 Starting dual-feasible tree-solution

DNEPSA needs to find an initial dual feasible tree-solution to start its iterations from. A method developed by Hultz and Klingman and presented in [53], was used. The steps the method uses are shown in Algorithm 5. These steps were implemented by a function that DNEPSA calls when it starts running, in order to produce a starting dual feasible tree-solution. Of course, the same function was also called by the implementation of DNSA for the computational results demonstrated in Section 4.4.

This method needs a network $G = (N, A)$ as an input and produces a dual feasible tree T. In its general form, it supposes that, for every arc $(i, j) \in A$, there exists a value k_{ij} that denotes the attenuation factor affecting the amount of flow going from node i to node j. For the algorithms discussed here, it is enough to assume that $k_{ij} = 1, \forall (i, j) \in A$. It selects arbitrarily (step 1) a node t as the root of the tree and then continues by selecting an arc at every iteration. Therefore, it needs $|N| - 1$ iterations to finish, except of the case that the network is disconnected. In that case, it is not possible to produce a dual feasible solution and the algorithm stops earlier. We denote as p the current iteration, $T(p)$ the set of fixed (selected) nodes in iteration p, S the set of arcs starting from a fixed node and ending to a non-fixed node, B the set of arcs that have already been selected to be in the basic tree and q the node whose dual variable was last set.

Algorithm 5 Algorithm to find a starting dual feasible solution.

Require: $G = (N, A), b, c$

procedure StartingDualFeasibleTree(G)

(Step 1)

Select arbitrarily the starting node t.

Set dual variable $w_t = \sum_{(i,j)\in N} c_{ij} / \prod_{(i,j)\in N} k_{ij}$.

Set $T(1) = \{t\}$ and $B = \emptyset$.

Set $q = t$ and $p = 1$.

Set $w_i^1(0) = \infty, \forall i \notin T(1)$ and $w_i^2(0) = 0, \forall i \notin T(1)$

(Step 2)

Set $S = \{(i,j) \in N : i \in T(p), j \notin T(p)\}$.

if $S = \emptyset$ **then** go to Step 3

else

$\forall j \notin T(p)$ set

$$w_j^1(p) = \begin{cases} min\{w_j^1(p-1), (w_q + c_{qi})/k_{qj}\} & , if\ (q,i) \in N \\ w_j^1(p-1) & , if\ (q,i) \notin N \end{cases}$$

end if

(Step 3)

Set $R = \{(i,j) \in N : i \notin T(p), j \in T(p)\}$.

if $R = \emptyset$ **then** go to Step 4

else

$\forall j \notin T(p)$ set

$$w_j^2(p) = \begin{cases} max\{w_j^2(p-1), (k_{jq}w_q - c_{jq}\} & , if\ (i,q) \in N \\ w_j^2(p-1) & , if\ (i,q) \notin N \end{cases}$$

end if

(Continued)

Algorithm 5 (Continued)

(Step 4)

16: **if** $S \cup R = \emptyset$ and $\exists j \notin T(p)$ **then** STOP. The network is disconnected.

17: **else**

18: **if** $S \neq \emptyset$ **then**

19: Set $w_f = \min_{j \notin T(p)} \{w_j^1(p)\}$

20: Set $b_f^1 = b_f + \sum_{(f,j) \in N, j \notin T(p)} b_j / k_{fj} + \sum_{(j,f) \in N, j \notin T(p)} k_{jf} b_j$

21: **end if**

22: **if** $R \neq \emptyset$ **then**

23: Set $w_r = \max_{j \notin T(p)} \{w_j^2(p)\}$

24: Set $b_r^1 = b_r + \sum_{(r,j) N, j \notin T(p)} b_j / k_{rj} + \sum_{(j,r) \in N, j \notin T(p)} k_{jr} b_j$

25: **end if**

26: **end if**

27: **if** $w_f b_f^1 \geq w_r b_r^1$ **then**

28: Set $q = f$ and $T(p) = T(p) \cup \{q\}$ and $B = B \cup \{(p, f)\}$

29: **else**

30: Set $q = r$.

31: Set $T(p) = T(p) \cup \{q\}$.

32: Set $B = B \cup \{(r, p)\}$.

33: **end if**

34: **if** $T(p) = N$ **then** STOP. The algorithm has been completed.

35: **else**

36: Set $T(p+1) = T(p)$.

37: Set $p = p + 1$.

38: go to Step 2.

39: **end if**

end procedure

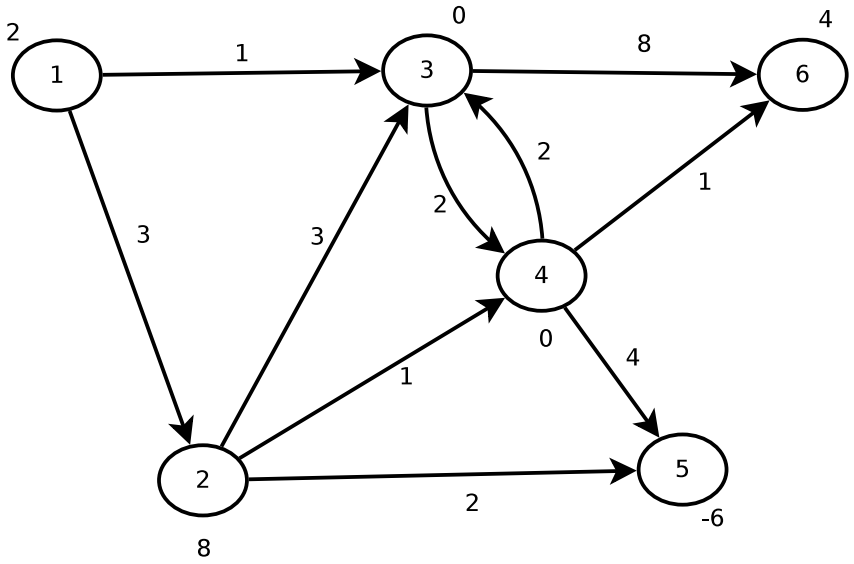

Figure 4.4: A network for the dual feasible tree algorithm.

The steps of the algorithm will be demonstrated by an example. Let's suppose we have the network depicted in Figure 4.4. Next to a node i is shown the supply/demand b_i for that node. Also, next to an arc (i, j), the cost c_{ij} for that arc is shown. The algorithm's steps for that network are described below.

Iteration 1

Step 1

Let's choose node 3 as the root of the tree, i.e. we set $t = 3$. Then it is

$$w_3 = \sum_{(i,j)\in A} c_{ij} = 27, \qquad T(1) = \{3\}$$

$$w_i^1(0) = \infty, \forall i \notin T(1), \quad w_i^2(0) = 0, \forall i \notin T(1)$$

$$B = \emptyset$$

Step 2

$$S = \{(3,4),(3,6)\}$$

$$w_4^1(1) = w_3 + c_{34} = 27 + 2 = 29$$

$$w_6^1(1) = w_3 + c_{36} = 27 + 8 = 35$$

Step 3

$$R = \{(1,3),(2,3),(4,3)\}$$

$$w_1^2(1) = w_3 - c_{13} = 27 - 1 = 26$$

$$w_2^2(1) = w_3 - c_{23} = 27 - 3 = 24$$

$$w_4^2(1) = w_3 - c_{43} = 27 - 2 = 25$$

Step 4

$$w_f = \min_i\{w_i^1(1)\} = 29\ for\ f = 4$$

$$b_4^1 = b_4^1 + (b_5 + b_6) + b_2 = 0 + 6 + 4 - 8 = 2$$

$$w_r = \max_i\{w_i^2(1)\} = 26,\ for\ r = 1$$

$$b_1^1 = b_1 + b_2 = -2 - 8 = -10$$

$$It\ is\ w_f b_f \geq w_r b_r, \Rightarrow q = 4$$

$$B = \{3,4\}$$

$$T = \{(3,4)\}$$

The algorithm needs 5 iterations in total (it is $|N| = 6$) in order to find a dual feasible tree-solution. If we proceed with all iterations, in the same way as it is described in the first iteration, then we find $T = \{(3,4),(4,6),(4,5),(2,5),(1,2)\}$, i.e. the dual feasible tree shown in Figure 4.5 is finally produced. It can be easily tested, by using Relations (2.1) and (2.2), that it is $s_{ij} \geq 0, \forall (i,j) \notin T$, so the tree the algorithm returns is indeed a dual feasible tree.

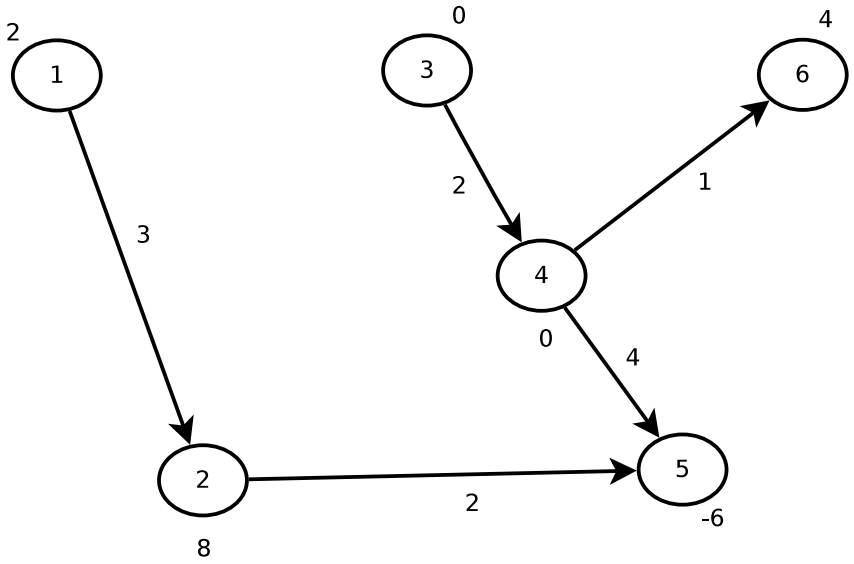

Figure 4.5: The dual feasible tree produced by the algorithm.

4.4 Computational results

In order to evaluate the performance of DNEPSA, an experimental comparison of DNEPSA against other standard Network Simplex-type Algorithm was performed. The comparison demonstrates the efficiency of Exterior-point algorithms on randomly generated MCNFP instances. In this section the results of these numerical tests are reported. DNEPSA was compared against the PNSA, DNSA and PNEPSA algorithms (presented briefly in sections 1.5.1, 1.5.2 and 1.5.3 respectively) for the MCNFP problem. The results of the comparisons against these three algorithms, were very good for DNEPSA in terms of the mean number of iterations needed and also, in terms of the mean time spent to solve the problem instances. The comparison though, is more "fair" against the Dual Network Simplex Algorithm (DNSA) than against PNSA and PNEPSA. The reason is that both DNSA and DNEPSA

need an initial dual feasible tree-solution to start their iterations. So, the two algorithms use the same starting point for the experimental tests. The method discussed in section 4.3 discovers a "good" starting point which is the same for both algorithms.

PNSA and PNEPSA, on the other hand, need a primal feasible tree-solution to start working. The big M method, described in section 1.5.1, was used in order to find the primal feasible solution the algorithms need. This method produces an extended graph (containing one artificial node and $|N|$ artificial arcs) and the algorithms work on this extended graph. The result is PNSA and PNEPSA need more work to finish, mainly because they start from tree-solution that is far away from an optimal solution. This is why the results of the comparison of DNEPSA against DNSA are only presented here.

The MCNFP problem instances used for the tests were created using the well-known NETGEN generator presented in section 4.1.2. The experiments were run on a desktop machine with an Intel Pentium 4 at 3.6 GHz processor, running Ubuntu 9.10 (Karmic Koala) version and having 3 GB memory (RAM DDR 2 400Mhz). The competitive programs were written in the C programming language, compiled by the gcc v4.2 compiler (the GNU Compiler Collection) with the "-O3" option used (fully optimized for speed).

The functions, used for the implementation of the algorithms, have been written following the same programming techniques for all algorithms, adjusted to the special characteristics of each one. The data structures described in section 4.2 were used for all algorithms. At every execution, the same dual feasible starting point was used for both DNEPSA and DNSA.

The time needed in order to find that initial solution, was included in the measurements.

Some preliminary computational results on random generated problems were published in [38]. This section presents a more detailed computational study, in order to estimate the efficiency of DNEPSA for problem instances of different densities. More specifically, five classes of instances were developed, one class for each of the following densities: 2%, 10%, 20%, 30%, and 40%. A graph's density D is estimated by the following formula:

$$D = \frac{|N|}{|A|(|A|-1)} \tag{4.1}$$

where $|N|$ denotes the number of nodes and $|A|$ denotes the number of arcs in the graph.

Each class consists of six problem categories, with varying dimensions. The number of the nodes in each class, starts from 200 nodes and goes up to 700 nodes (with a step equal to 100). The number of the arcs in a category depends on the density of the class of the instance and the number of nodes, as it is given by Relation (4.1). Moreover, in each category of the classes, ten network instances have been created, in order to compute the average number of iterations and also the average CPU time needed for the algorithms to finish. To conclude with, 5 different classes were used, with 6 problem categories for each class and 10 network instances for each category. Therefore 300 MCNFP network instances were randomly created by NETGEN and solved by using both algorithms. The comparative computational results and the normalized comparative computational results for DNEPSA

and DNSA are presented in Tables 4.2 and 4.3 respectively. Table 4.2 gives the average number of iterations and the average time in seconds that the two algorithms need to solve 10 randomly generated MCNFP instances. Table 4.3 gives the same information in a normalized format, so that it is clear in what percentage is DNEPSA faster than DNSA (again in terms of the number of iterations and the time spend).

A number of figures will be presented now to demonstrate the comparative computational results selected. All the figures have been created by using the gnuplot plotting program, version 4.2. Figure 4.6 demonstrates the average number of iterations needed for DNEPSA and DNSA to solve randomly generated MCNFP instances on networks of 2% density. Figure 4.7 gives a similar graph for the average CPU time (in seconds) needed for the same MCNFP instances. Figures 4.8 and 4.9 give similar graphs for the MCNFP problem instances on networks of 10% density. Finally, Figures 4.10 and 4.11, Figures 4.12 and 4.13 and Figures 4.14 and 4.15 give the graphs for network instances of density 20%, 30% and 40% respectively.

The comparative study of DNEPSA and DNSA algorithms shows that, for the instances considered, DNEPSA needs less CPU time and fewer number of iterations than DNSA does. A theoretical explanation of DNEPSA's superiority, is the fact that DNEPSA can cross over the infeasible region of the dual problem and return back to it to find an optimal solution. That fact may lead to an essential reduction on the number of iterations and the CPU time needed.

Table 4.2: Number of Iterations and CPU time for randomly generated instances.

		DNSA		DNEPSA	
Density	Nodes × Arcs	*niter*	*CPU*	*niter*	*CPU*
2%	200 × 796	296	0.85	289	0.71
	300 × 1794	550	4.02	516	3.31
	400 × 3192	969	11.10	845	8.90
	500 × 4990	1315	34.02	1143	26.50
	600 × 7188	1550	72.20	1237	51.60
	700 × 9786	2317	140.38	1784	95.20
10%	200 × 3980	383	1.43	350	1.01
	300 × 8970	853	7.70	770	5.01
	400 × 15960	1451	30.02	1277	18.30
	500 × 24950	2073	78.69	1811	46.81
	600 × 35940	2819	170.20	2231	97.10
	700 × 48930	4149	365.00	3101	188.21
20%	200 × 7960	389	2.34	359	1.71
	300 × 17940	884	11.50	786	8.02
	400 × 31920	1567	44.20	1298	29.60
	500 × 49900	2271	125.47	1841	82.90
	600 × 71880	3050	264.20	2291	155.40
	700 × 97860	4478	591.64	3202	323.23
30%	200 × 11940	416	3.13	364	2.16
	300 × 26910	954	16.50	803	11.20
	400 × 47880	1628	72.15	1331	47.32
	500 × 74850	2357	176.60	1848	103.63
	600 × 107820	3083	265.52	2366	155.91
	700 × 146790	4535	981.35	3278	497.33
40%	200 × 15920	454	4.59	377	3.06
	300 × 35880	1049	18.15	864	11.76
	400 × 63840	1791	79.37	1455	51.05
	500 × 99800	2512	199.26	1913	110.10
	600 × 143760	3398	312.25	2580	167.50
	700 × 195720	4943	1069.67	3473	552.08

Table 4.3: Normalized number of iterations and CPU time for randomly generated instances.

		DNSA		DNEPSA	
Density	Nodes × Arcs	*niter*	*CPU*	*niter*	*CPU*
2%	200 × 796	1.02	1.20	1	1
	300 × 1794	1.07	1.21	1	1
	400 × 3192	1.15	1.25	1	1
	500 × 4990	1.15	1.28	1	1
	600 × 7188	1.25	1.40	1	1
	700 × 9786	1.30	1.47	1	1
10%	200 × 3980	1.09	1.42	1	1
	300 × 8970	1.11	1.54	1	1
	400 × 15960	1.14	1.64	1	1
	500 × 24950	1.14	1.68	1	1
	600 × 35940	1.26	1.75	1	1
	700 × 48930	1.34	1.94	1	1
20%	200 × 7960	1.08	1.37	1	1
	300 × 17940	1.13	1.43	1	1
	400 × 31920	1.21	1.49	1	1
	500 × 49900	1.23	1.51	1	1
	600 × 71880	1.33	1.70	1	1
	700 × 97860	1.40	1.83	1	1
30%	200 × 11940	1.14	1.45	1	1
	300 × 26910	1.19	1.47	1	1
	400 × 47880	1.22	1.52	1	1
	500 × 74850	1.28	1.70	1	1
	600 × 107820	1.30	1.70	1	1
	700 × 146790	1.38	1.97	1	1
40%	200 × 15920	1.20	1.50	1	1
	300 × 35880	1.21	1.54	1	1
	400 × 63840	1.23	1.55	1	1
	500 × 99800	1.31	1.81	1	1
	600 × 143760	1.32	1.86	1	1
	700 × 195720	1.42	1.94	1	1

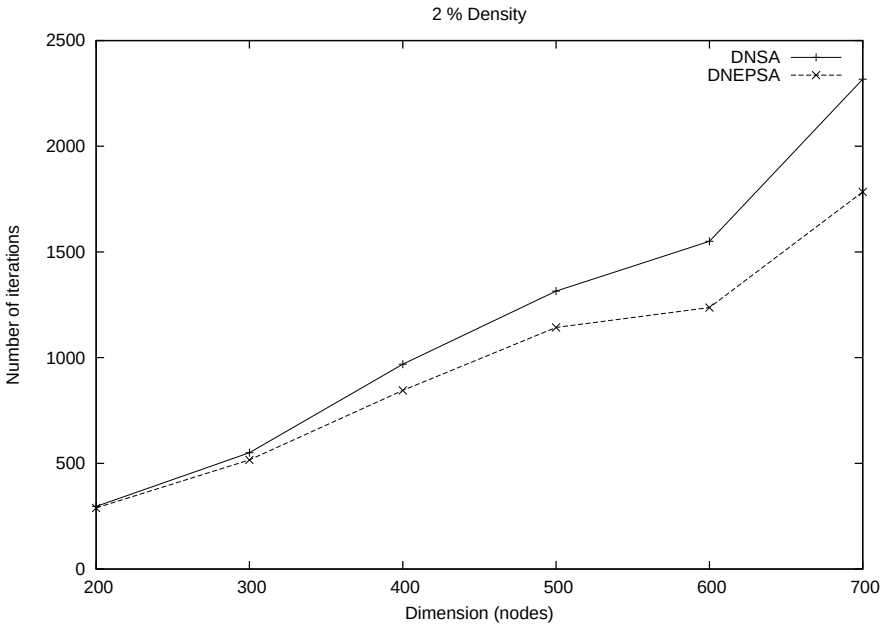

Figure 4.6: Average number of iterations for networks of density 2%.

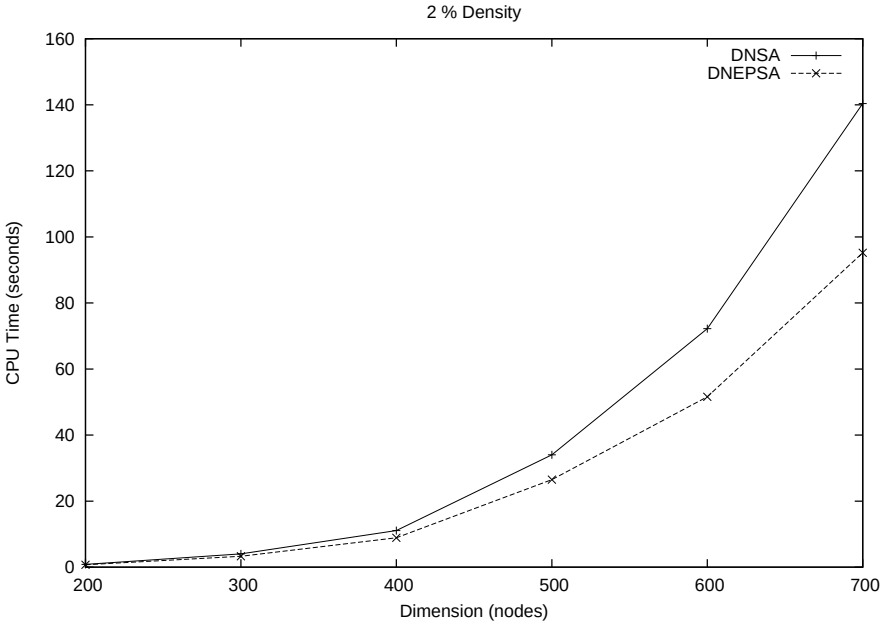

Figure 4.7: CPU time (in seconds) for networks of density 2%.

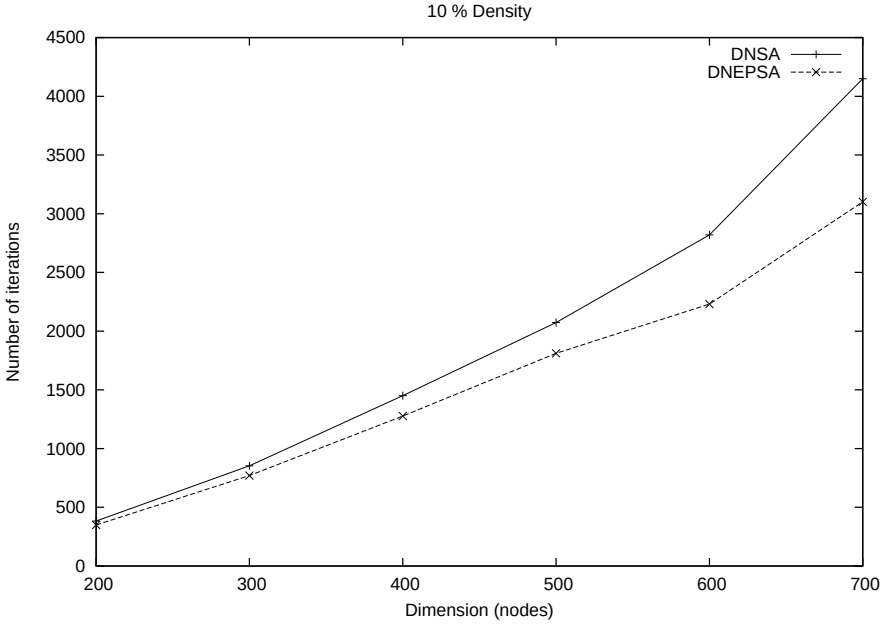

Figure 4.8: Average number of iterations for networks of density 10%.

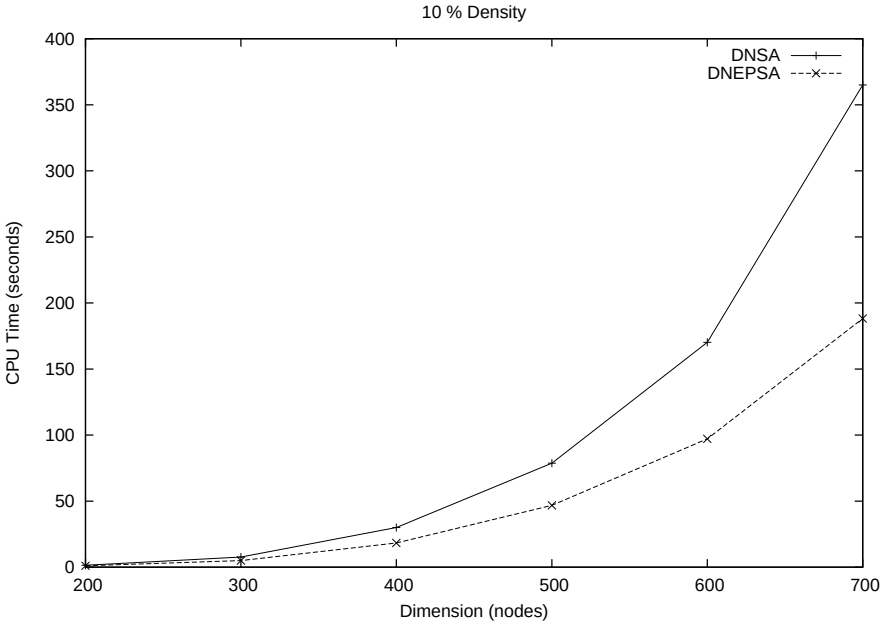

Figure 4.9: CPU time (in seconds) for networks of density 10%.

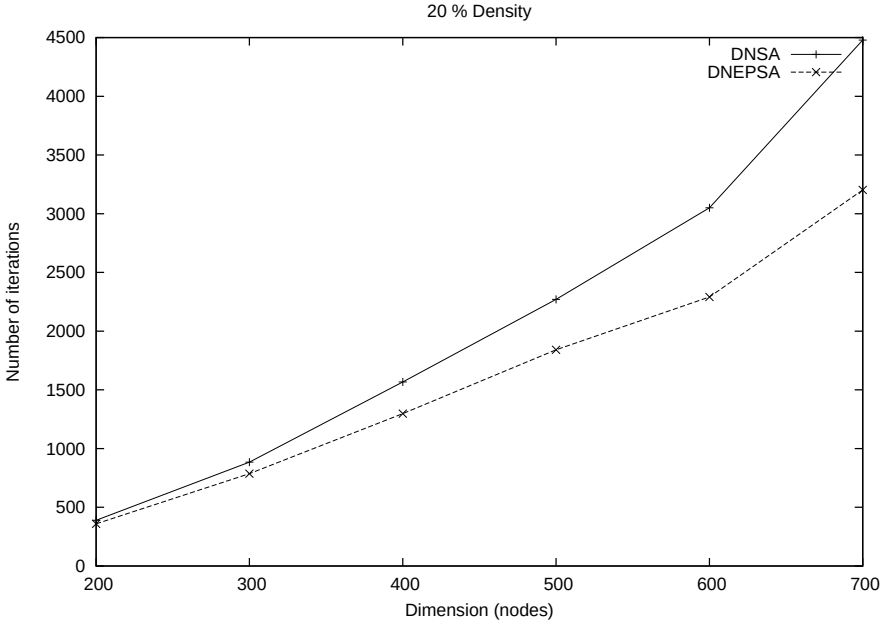

Figure 4.10: Average number of iterations for networks of density 20%.

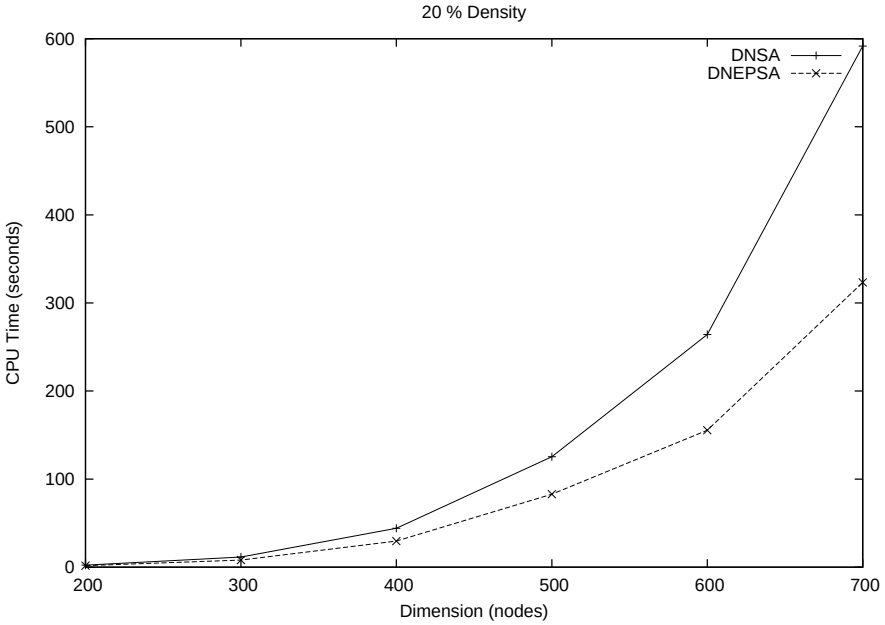

Figure 4.11: CPU time (in seconds) for networks of density 20%.

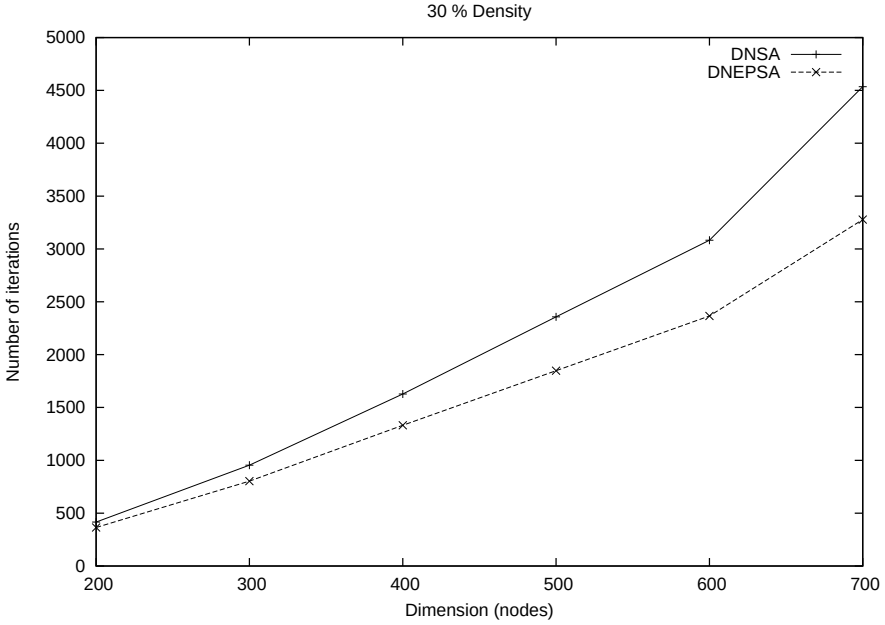

Figure 4.12: Average number of iterations for networks of density 30%.

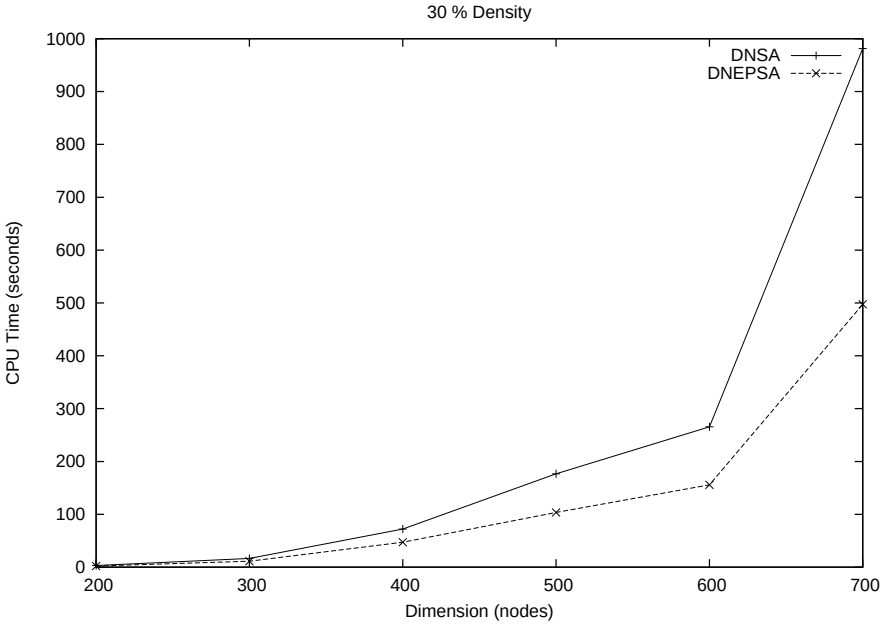

Figure 4.13: CPU time (in seconds) for networks of density 30%.

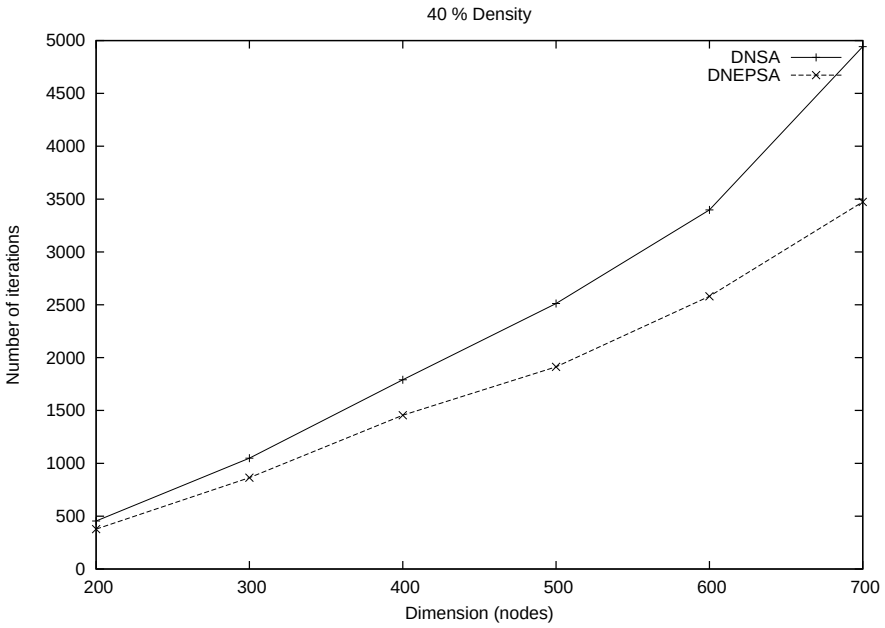

Figure 4.14: Average number of iterations for networks of density 40%.

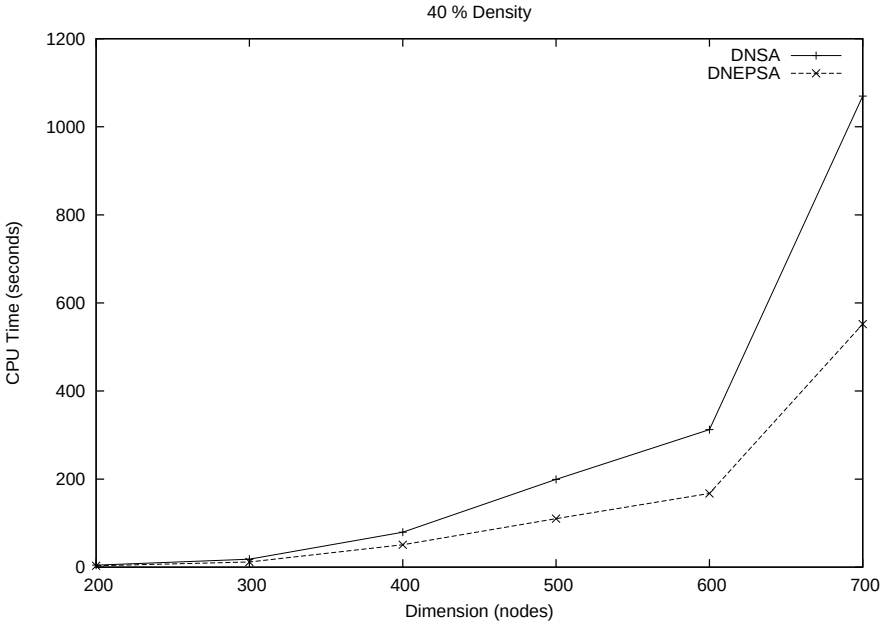

Figure 4.15: CPU time (in seconds) for networks of density 40%.

4.5 Statistical Analysis of the Performance of the algorithm

In order to have an insight into the performance evaluation of an algorithm, a statistical analysis is needed, as it is shown in [20], [64] and [66]. A systematic approach to the problem of inferring asymptotic bounds from experimental data is given in [65]. A study on how well an algorithm's mathematical complexity tally with the statistical measure is presented in [18]. For a list of properties of a statistical complexity bound as well as to understand what is the meaning of the design and analysis of computer experiments, [19] may be consulted.

The results of the statistical analysis presented in this section, were based on the IBM PASW Statistics package, version 19. The results are analysed in order to improve and strengthen the experimental results presented in the previous section. Typically, an increase or decrease in the running time leads to an increase or decrease in the variance respectively. This can be attributed to the fact that, the running times are bounded below by zero. Therefore, a logarithmic transformation of the running times usually is preferred.

Figures 4.16 and 4.17 depict two scatter-plots where a straight line with 45° slope appears for reference. Figure 4.16 is showing the number of iterations for DNEPSA and DNSA, while Figure 4.17 refers to CPU time needed on a double-logarithmic scale. Both plots exhibit a linear trend. The majority of the points lie below the 45° line, thus indicating that DNSA is generally slower than DNEPSA.

However, in order to draw a valid statistical conclusion about the differ-

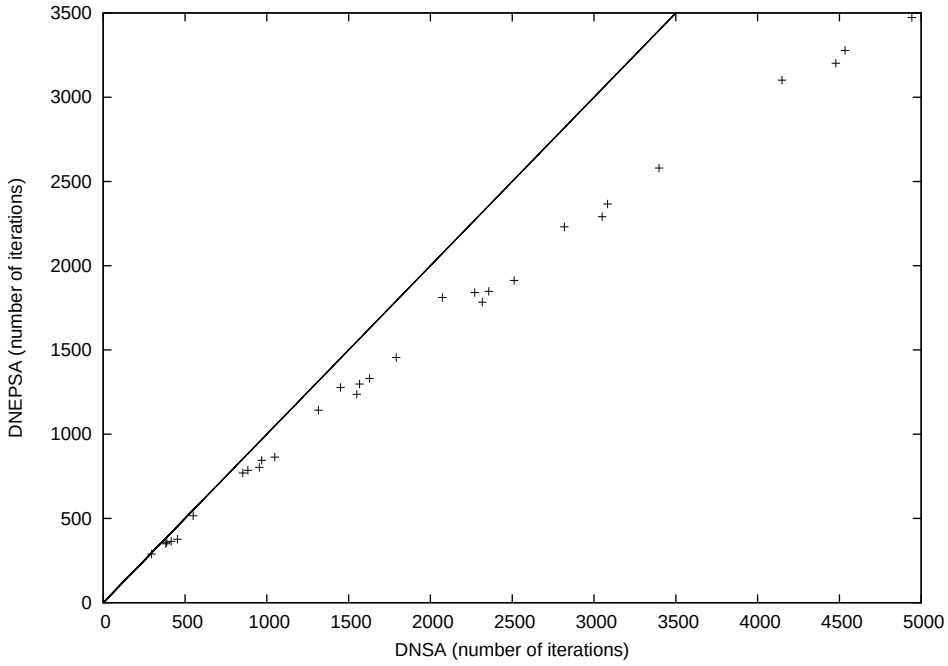

Figure 4.16: Scatterplot of DNEPSA vs DNSA (number of iterations).

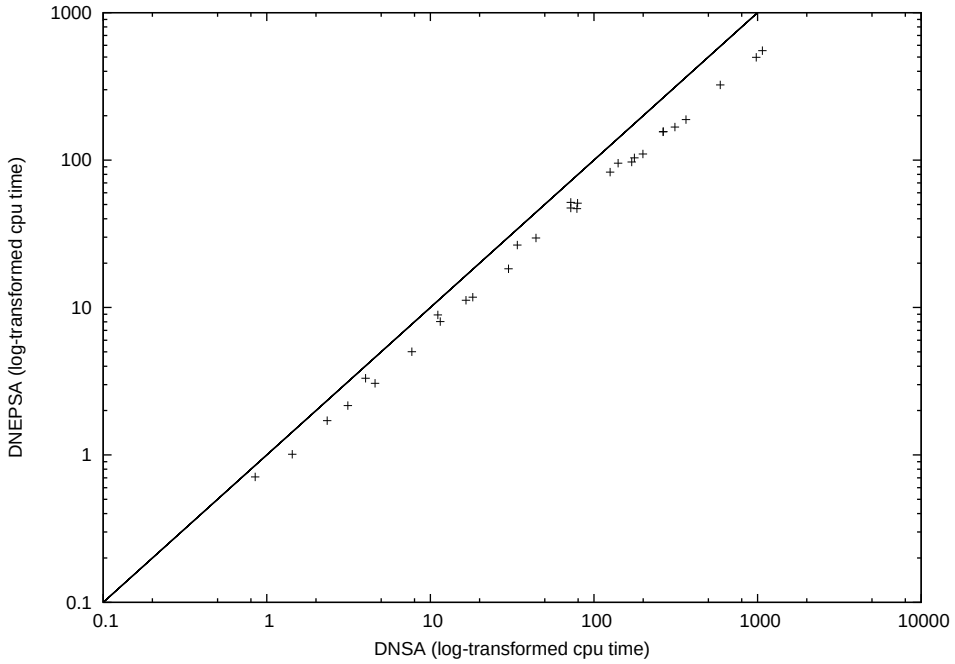

Figure 4.17: Scatterplot of DNEPSA vs DNSA (log-transformed CPU time).

ences between the solution times or the number of iterations, a hypothesis testing is needed. To accomplish this, a decision must be made regarding the use of parametric or non-parametric statistical hypothesis test. Let's denote by d_{niter} the vector of pairwise differences in the number of iterations between DNSA and DNEPSA and $niter_{DNSA}$ and $niter_{DNEPSA}$ the vectors of the number of iterations for the two algorithms respectively (the values are taken from the results in Table 4.2). So, it is:

$$d_{niter} = niter_{DNSA} - niter_{DNEPSA}$$

In a similar way, we denote by d_{cpu} the vector of pairwise differences in CPU time between DNSA and DNEPSA. Thus, it is

$$d_{cpu} = t_{DNSA} - t_{DNEPSA}$$

where t_{DNSA} and t_{DNEPSA} correspond to the vectors of DNSA and DNEPSA CPU time respectively (the values are taken from the results in Table 4.2).

By applying a one-sample Kolmogorov-Smirnov test to the sample of d_{niter}, we take a p-value equal to 0.260 which is greater than 0.05. Therefore, there is no reason to doubt the fact that the distribution of d_{niter} is normal and we can safely proceed to a paired-sample t-test. On the contrary, by applying again a one-sample Kolmogorov-Smirnov test to the sample of d_{cpu}, we take a p-value equal to 0.018 which is less than 0.05. Therefore, there is sufficient evidence to reject the normality assumption of the distribution of d_{cpu} and thus we should proceed to a Wilcoxon matched-pairs signed-ranks test.

In the first case, the paired-sample t-test is actually a test on the differences between the number of iterations for DNSA and DNEPSA. If we denote by M the population mean of pairwise differences, then $M = 0$ indicates that on randomly generated problem instances, the experimental performance of DNSA and DNEPSA is about the same. However, $M > 0$ implies that DNSA is likely to need more iterations, whereas $M < 0$ implies that DNEPSA needs more iterations. Since, we have no a priori reason to consider either algorithm is doing less iterations, we will test the hypothesis that

$$H_0 : M = 0$$

versus

$$H_1 : M \neq 0$$

The mean difference in number of iterations (Mean = 423.27, Standard Error = 74.90, N = 30) was significantly greater than zero, $t = 5.65$, two-tail $p < 0.05$, verifying the conclusion that the two algorithms perform differently. A 95% confidence interval about the mean difference in number of iterations is (270, 576), indicating that the mean difference in number of iterations lies between 270 and 576 iterations. Therefore, we reject the hypothesis H_0 and conclude that the algorithms perform differently. Since the values of the pairwise differences are all positive, we reach to the conclusion that DNEPSA needs a smaller number of iterations than DNSA.

In the second case, the Wilcoxon matched-pairs signed-ranks test is a distribution-free hypothesis test for the population median. The Wilcoxon

signed rank statistic W_+ is based on the sizes of the absolute values of the differences between observations of solution times. If we denote by M the population median of pairwise differences, then $M = 0$ indicates that on a randomly generated problem instances the experimental performance of DNSA and DNEPSA is about the same. Since we have no a priori reason to consider either algorithm is faster, we will test again the hypothesis that $H_0 : M = 0$ versus $H_1 : M \neq 0$

The median differences in solution times is significantly different from zero (it is $W_+ = 465$, $p < 0.05$) providing evidence that the two algorithms perform differently. Therefore, we reject hypothesis H_0 and conclude that the algorithms perform in a different way. Since the values of the pairwise differences are all positive, we conclude that DNEPSA performs faster than DNSA. It should be noted that the differences between the null hypothesis of the Wilcoxon matched-pairs signed-ranks test and the paired-sample t-test, is that the median difference between pairs of observations, instead the mean difference between pairs, is zero.

4.6 Empirical complexity of DNEPSA using statistical analysis

In this section, a statistical analysis based on the experimental results will be performed, in order to measure the empirical complexity of DNEPSA. Assume an algorithm with expected running time $T(n, m) = O(g(n, m))$, where $g(n, m)$ is a function with parameterized input length by n and m.

Then, the estimated function $O(g(n,m))$ is usually referred to as the empirical complexity of the algorithm (see [20]).

The empirical complexity of DNEPSA presented in this section, was based on a statistical analysis carried out by using the IBM PASW Statistics package, version 19. The research approach includes a stepwise multiple regression analysis. The response variables are:

- the number of iterations
- the CPU time

while the predictor variables consist of a large number of variables that are combinations of n and m (e.g., $logn$, $logm$, n^2, n^3, m^2, nm, nm^2, etc.). A similar approach for the evaluation of the experimental performance of the classical Simplex algorithm for the linear programming problem was presented by R. Vanderbei in [92], as also in [17] and [22].

R-squared (R^2) is usually called the coefficient of determination and equals to the ratio of the sum of squares explained by our regression model and the total sum of squares around the mean. Furthermore, adjusted R-squared ($\bar{R}^2$) is a modification of R-squared that adjusts for the number of explanatory variables in a model. Unlike R-squared, the adjusted R-squared increases only if the new variable clearly improves the regression model (and not by chance).

The regression analysis, regarding the number of iterations, with an adjusted R-Squared value equal to $\bar{R}^2 = 97.5\%$ (very nearly unity) is indicated in the following regression equation:

$$niter = a_1 + b_1 nlogm + c_1\sqrt{n} \tag{4.2}$$

where $a_1 = 775.991$, $b_1 = 1.45$, and $c_1 = -105.336$. Hence, about 97.5% of variation in the number of iterations can be explained by the variables $nlogm$ and $\sqrt{n}$. However, function $g_1(n, m) = nlogm$ is the one having the largest order of growth.

In a similar way, the regression analysis regarding the CPU time needed to solve each problem instance based on the problem dimension, with an adjusted R-Squared value $\bar{R}^2 = 97.4\%$ (that is almost unity), is indicated in the following regression equation:

$$cpu = a_2 + b_2 mn^2 + c_2 nm + d_2\sqrt{nm} \tag{4.3}$$

where $a_2 = -16.871$, $b_2 = 1.959 \times 10^{-8}$, $c_2 = -1.208 \times 10^{-5}$, and $d_2 = 0.29$. Hence, about 97.4% of variation in the CPU time can be explained by the variables mn^2, nm, and $\sqrt{nm}$. In this case, function $g_2(n, m) = mn^2$ has the largest order of growth.

Moreover, the Analysis of Variance (ANOVA) resulted $P < 0.001$, showing an absolute linear correlation between the variables of each regression equation. The fit of both polynomials in Relations (4.2) and (4.3) were quite good at 5% level of significance.

Figure 4.18 depicts the normal probability plot of the standardized residuals regarding the number of iterations, while Figure 4.19 refers to the CPU time needed. The standardized residuals of both regressions are normally distributed (Kolmogorov-Smirnov normality tests $P > 0.05$ in both cases).

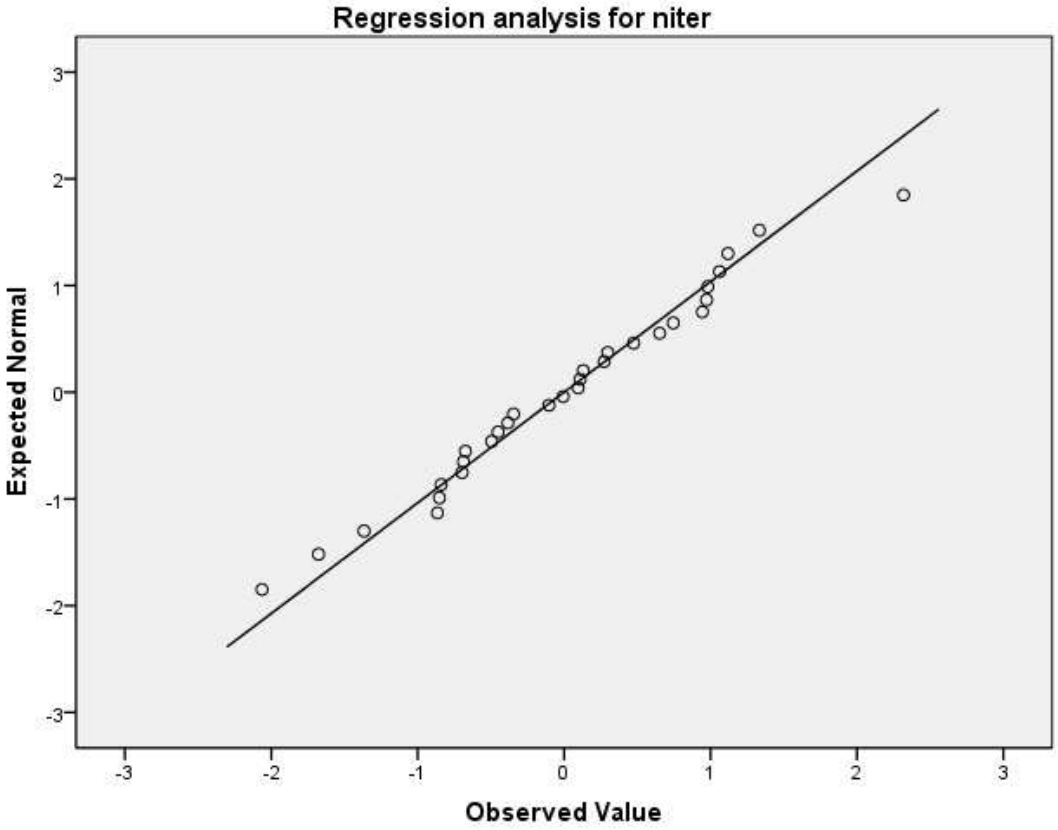

Figure 4.18: Normal Q-Q Plot of standardized residuals (number of iterations).

Therefore, the regression analysis indicates that the DNEPSA algorithm requires $O(nlogm)$ number of iterations and $O(mn^2)$ cpu time. Thus, DNEPSA has a polynomial empirical computational behavior regarding the required number of iterations and the cpu time, observed statistically.

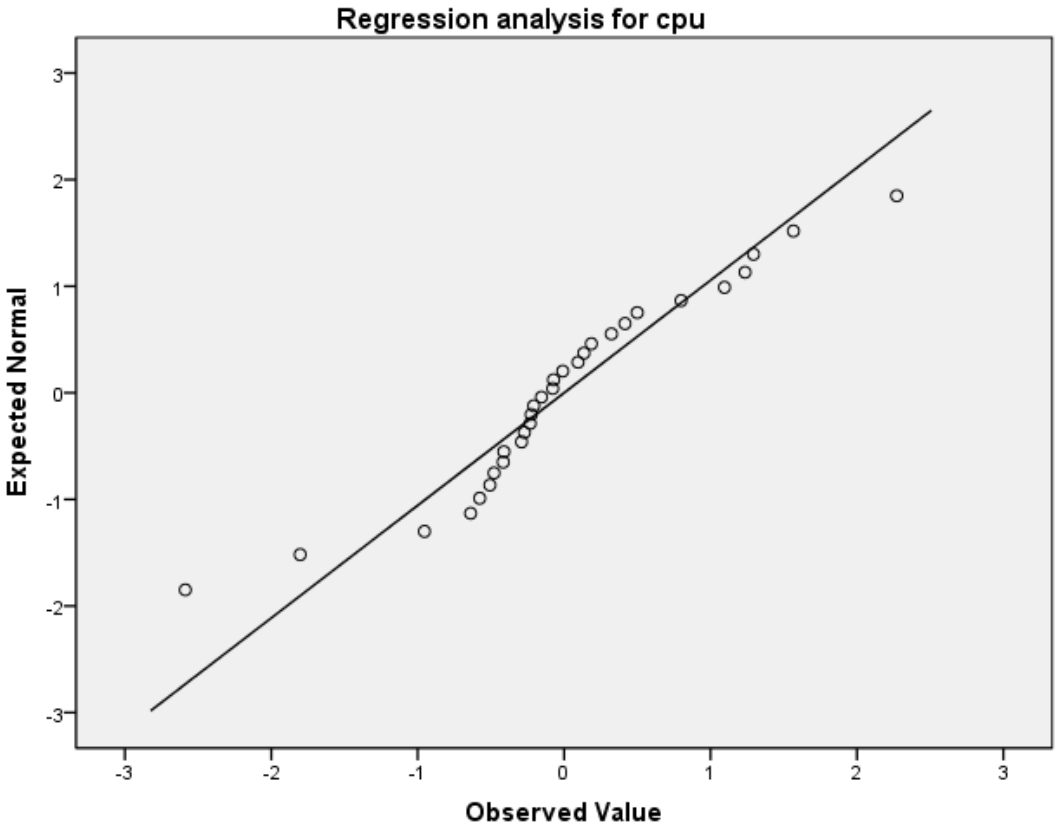

Figure 4.19: Normal Q-Q Plot of standardized residuals (CPU time).

Chapter 5

Implementation of DNEPSA by using Dynamic Trees

5.1 Introduction to Dynamic Trees

A *dynamic tree* is an abstract data type that allows the maintenance of a collection of vertex-disjoint rooted trees. These trees can be linked together by adding a new edge and they can be split into subtrees by removing any edge. These operations are usually referred as *linking* and *cutting* and dynamic trees are often referred as *link/cut trees*.

Dynamic trees were first introduced by Sleator and Tarjan in [85] and [86]. Sleator and Tarjan proposed two versions of the dynamic trees data structure. The first version has a time bound of O(1ogn) per operation when the time is amortized over a worst-case sequence of operations. The second version is slightly more complicated and it has a worst-case time bound of O(logn) per tree operation. One (or more) values may be associated with

every edge or vertex of a dynamic tree. The data structure may also be modified in order to contain arcs instead of edges.

Dynamic trees may be used for the design of fast algorithms for various kinds of problems like, finding the nearest common ancestor of two nodes in a tree, solving network flow problems (minimum or maximum flow), solving transportation problems etc.

Dynamic trees allow the easy modification of their structure by using mainly three kinds of operations:

- *link(u,v,c)*: links together two dynamic trees, where u is the root of the first tree and v a vertex of the second tree, by adding a new edge (u,v) of value (cost) c. The root of the second tree is the root of the extended tree produced.

- *cut(v)*: divides a dynamic tree into two subtrees by deleting the edge that connects v to its parent. It cannot be applied if v is the root of the tree.

- *evert(v)*: makes node v the root of the tree by turning the dynamic tree "inside out".

Beyond the three operations shown above, there can be other useful operations that may be implemented for a dynamic tree data structure, like the five operations described below:

- *parent(v)*: returns the parent of a vertex.

- *root(v)*: returns the root of the dynamic tree containing v.

- *cost(v)*: returns the cost of the edge (v, parent(v)).
- *mincost(v)*: finds the edge of minimum cost on the path joining vertex v to root(v).
- *update(v,x)*: adds a certain value x to all edges on the path from vertex v to root(v).

The above operations may be modified or new operations can be added in order to fit any special needs of an algorithm that uses dynamic trees. In Figure 5.1 two dynamic trees are shown. Nodes a and q are the roots of the two trees so, operations *root(j)* and *root(t)* return nodes a and q respectively. Operation *parent(e)* gives node b because node b is the parent of node e. Operation *cost(e)* returns the value 3 since this is the value (cost) associated with the edge *(e, parent(e))* and operation *mincost(n)* returns edge *(a,b)*, because this is the edge of minimum cost on the path from node n of the tree to the root node *root(n)=a*.

After applying operation *update(p,-1)* on the tree of Figure 5.1(a), all the costs from node p to node *root(p)* are reduced by 1, and we take the tree shown in Figure 5.2. The linking operation *link(q,i,4)* joins the trees in Figures 5.1(a) and 5.1(b) by adding a new edge (q,i) of cost 4. The dynamic tree produced is depicted in Figure 5.3. Operation *cut(d)* on the tree of Figure 5.1(a), deletes edge (a,d) and produces the two new trees depicted in Figure 5.4. Finally, by applying the operation evert(f) on the tree in Figure 5.1(a), node f becomes the root of the new dynamic tree, as it can be seen in Figure 5.5. In section 5.2 a method is described for the representation of

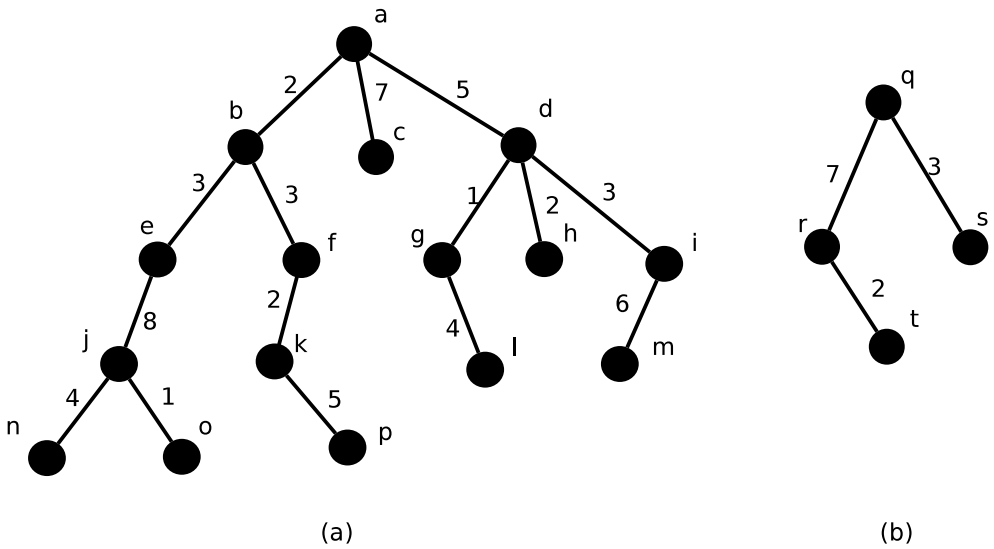

Figure 5.1: Dynamic trees

dynamic trees as a set of vertex disjoint paths. Section 5.3 describes how can dynamic trees be used for the implementation of DNEPSA.

5.2 Representation of Dynamic Trees as a set of paths

The techniques described in section 4.2 can be used for the representation of dynamic trees; for every vertex v of the tree, its parent p(v) is stored together with the cost of the edge (v,p(v)). Using this representation, *parent*, *cost*, *link* and *cut* operations require O(1) time. The problem is that the time requirements for each of the other operations is proportional to the length of the tree path from v to root(v). This cost is O(n) in the worst case, where n is the number of vertices in the graph. It is possible to reduce the cost of time-consuming operations to O(logn), at the cost of increasing the time

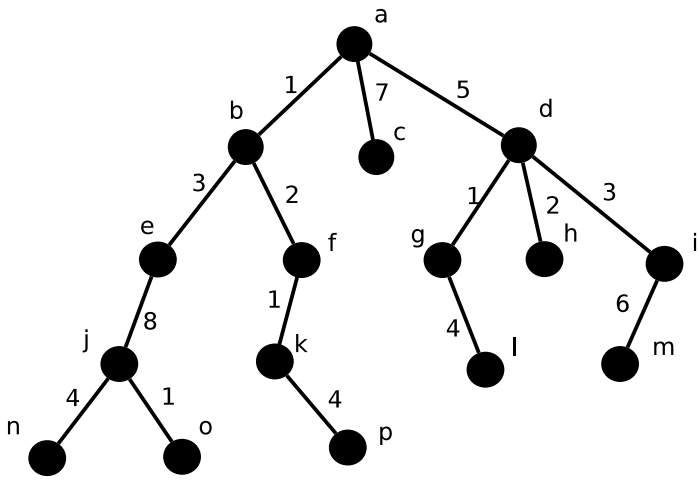

Figure 5.2: Dynamic tree after applying operation update

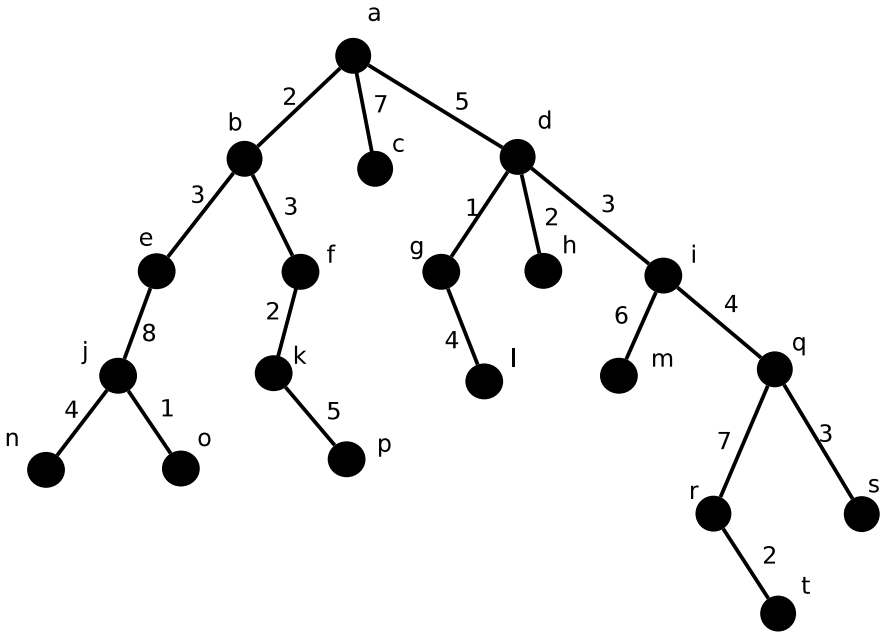

Figure 5.3: Linking of dynamic trees

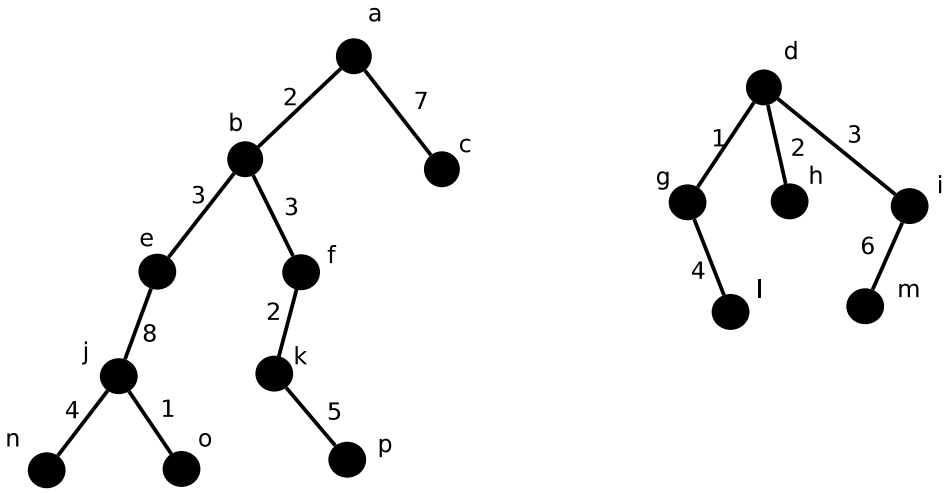

Figure 5.4: Cutting operation on dynamic trees

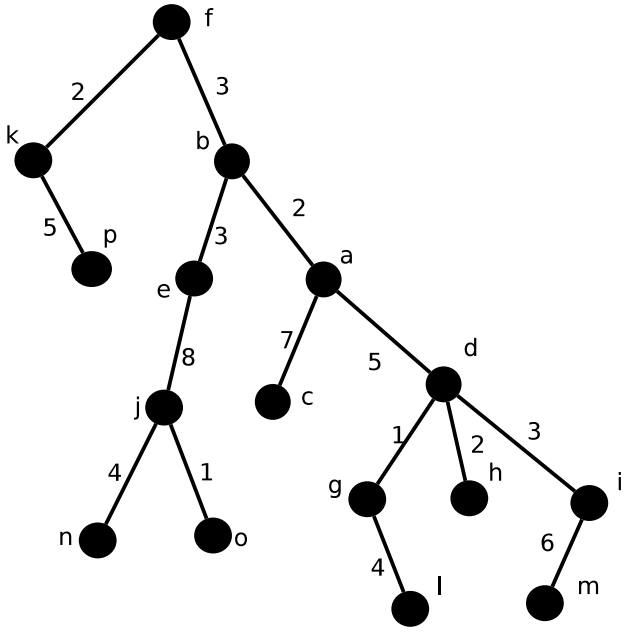

Figure 5.5: Evert operation on dynamic trees

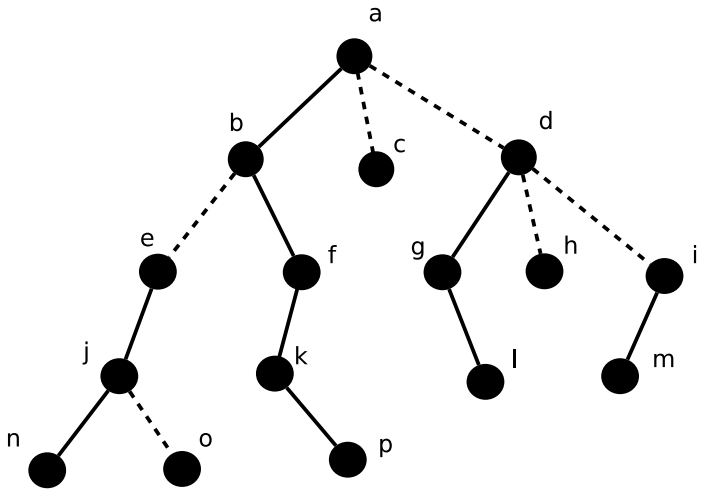

Figure 5.6: Representation of a dynamic tree as a set of vertex disjoint paths

for the the less time-consuming operations to O(1ogn). This can be done by representing a dynamic tree as a set of vertex-disjoint paths, as it is described in [85]. In order to do that, the edges of a dynamic tree will be partitioned into two types, the solid and the dashed edges.

Figure 5.6 gives a representation of the dynamic tree of Figure 5.1(a), as a collection of vertex disjoint paths by drawing some of its edges as dashed lines. Of course this representation is not unique. Every vertex belongs to exactly one path. A vertex may belong to at most one solid edge but it can belong to more than one dashed edges. The node of a path that is closer to the root of the dynamic tree is the *tail* of the path and the node that is farther is the *head* of the path. We have the following seven paths: [n,j,e], [o], [p,k,f,b,a], [c], [l,g,d], [h] and [m,i]. In path [p,k,f,b,a] for example, node p is the head of the path, while node a is the tail of the path.
A set of primitive path operations have to be implemented:

- *path(v)*: returns the path containing vertex v.

- *head(p)*: returns the first (bottommost) node of path p.
- *tail(p)*: returns the the last(topmost) node of path p.
- *before(v)*: returns the previous node of node v on path *path(v)*.
- *after(v)*: returns the next node of node v on *path(v)*.
- *pcost(v)*: returns the cost of edge (v, after(v)).
- *pmincost(p)*: finds a node v on path p so that the edge *(v, after(v))* has the minimum cost on p.
- *pupdate(p,x)*: adds a value x to the cost of every edge of path p.
- *reverse(p)*: reverses the direction of path p by making its head to be the tail and vice versa.
- *concatenate(p,q,x)*: combines two paths p and q by adding a new edge *(tail(p), head(q))* of cost x and returns the combined path.
- *split(v)*: deletes edges *(before(v),v)* and *(v, after(v))* producing two subpaths.

In addition, two additional composite path operations, described below, are also needed. These operations can be implemented by using the primitive path operations described above.

- *splice(p)*: transforms the dashed edge leaving *tail(p)* into a solid edge and converts to dashed the solid edge entering *parent(tail(p))*(if any). This way, path p is extended to include more vertices.

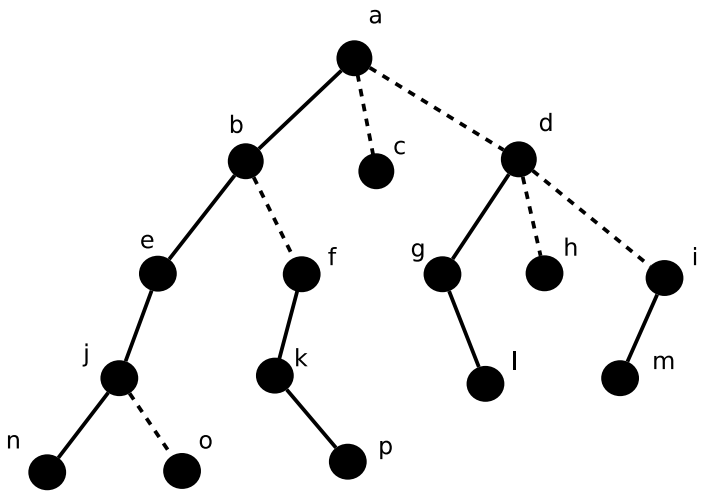

Figure 5.7: Splice operation on a path

- *expose(v)*: transforms every dashed edge from v to *root(v)* into a solid edge.

In Figure 5.7, the result of a *splice(path(e))* operation on the dynamic tree of Figure 5.6 is depicted. Edge (b,e) was converted to solid, while edge (b,f) was converted to dashed. After that, if a *expose(m)* operation is performed, the dynamic tree of Figure 5.8 is produced. All the dashed edges from node m to *root(m)* have been transformed to dashed and edge (d,g) is transformed to dashed so that, every vertex of the dynamic tree belongs to at most one solid edge.

All the dynamic tree operations, described in section 5.1, can be implemented by using the primitive and composite path operations presented here. Sleator and Tarjan proved in [85] that, in a sequence of m dynamic tree operations, there are O(mlogn) path operations, with splice and expose broken down into their component operations (n is the total number of vertices).

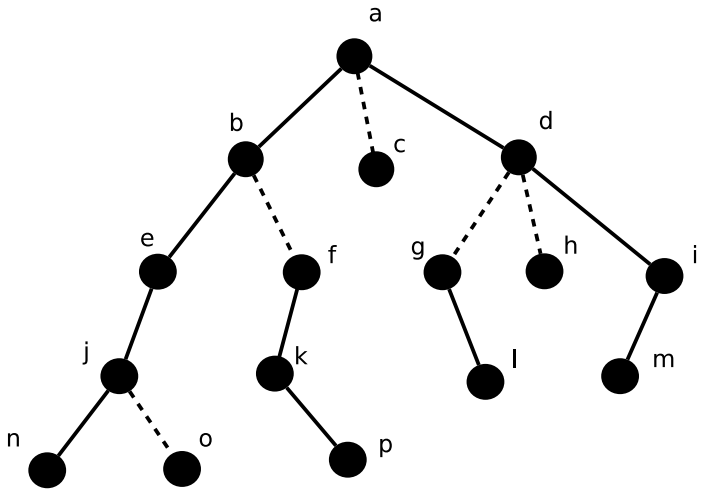

Figure 5.8: Expose operation on a path

5.3 Use of Dynamic Trees in the implementation of DNEPSA

DNEPSA, as it was described in section 2.1, maintains a tree-solution that is updated iteration by iteration. Dynamic tree data structures can be used in order to store the tree-solutions the algorithm produces. First of all, DNEPSA starts from an initial dual feasible tree-solution that can be represented as a dynamic tree. Figure 5.9 shows a possible initial dual feasible tree-solution used by DNEPSA in order to solve a MCNFP problem. Such a dynamic tree can be built by the method described in section 4.3, by slightly modifying it to start from a collection of $|V|$ single-vertex dynamic trees and use a sequence of link operations to construct the tree. The tree consists of three dynamic paths: [8, 2, 4, 6], [7, 3] and [5,1]. The supply for each vertex and the flow on each basic arc are also shown in the figure. We have arcs instead of edges in the dynamic tree but, this is not a problem because the

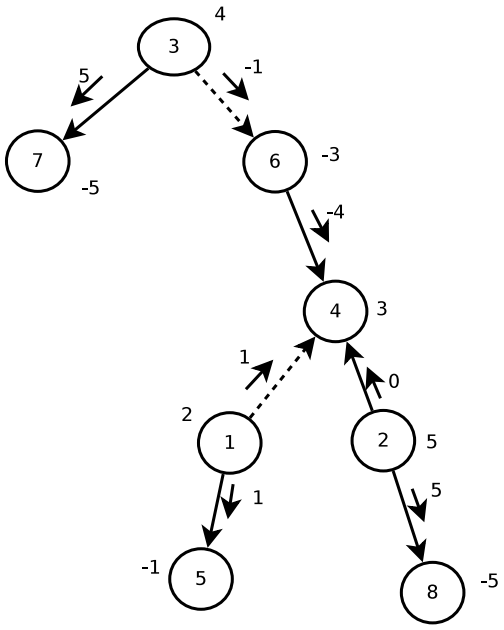

Figure 5.9: Initial tree-solution for DNEPSA as a dynamic tree

direction can be represented by adding a simple field in the data structure, as it is described later in this section.

The paths of a dynamic tree can be implemented as *binary trees*. Figure 5.10 shows the representation of the three paths of the dynamic tree of Figure 5.9 as three different binary trees (a), (b) and (c), one for each path. As it is easy to see in Figure 5.10, the external nodes of a binary tree, in left-to-right order, correspond to the vertices of the path from head to tail. An internal node w in a binary tree B that represents a path p, corresponds to an edge (u,v) of p, where u and v are the leaf nodes of B that are just before and after w when the nodes of B are traversed in symmetric order (inorder traversal). Next to each internal node in the figure, a label indicates the edge they correspond to.

For every node of a binary tree used to represent a dynamic path, a num-

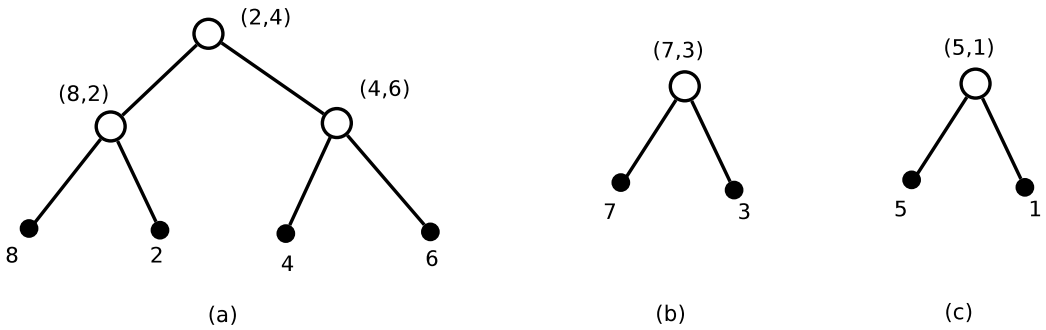

Figure 5.10: Representation of paths as binary trees

ber of fields have to be used in the data structure to store all the information needed. Such fields include:

- *external*: it can only take value 0 or 1, indicating whether the node is internal or external (leaf node).
- *direction*: for an internal node it stores the direction of the corresponding arc.
- *reversed*: shows if an arc was reversed. This way it is easy to reverse an arc making it easy to implement operation *evert*.
- *bparent*: a pointer to the parent of a node. It is null if the node is the root of the tree.
- *bhead*: a pointer to the head of the path.
- *btail* : a pointer to the tail of the path.
- *bleft* : a pointer to the left child of an internal node.
- *bright*: a pointer to the right child of an internal node.

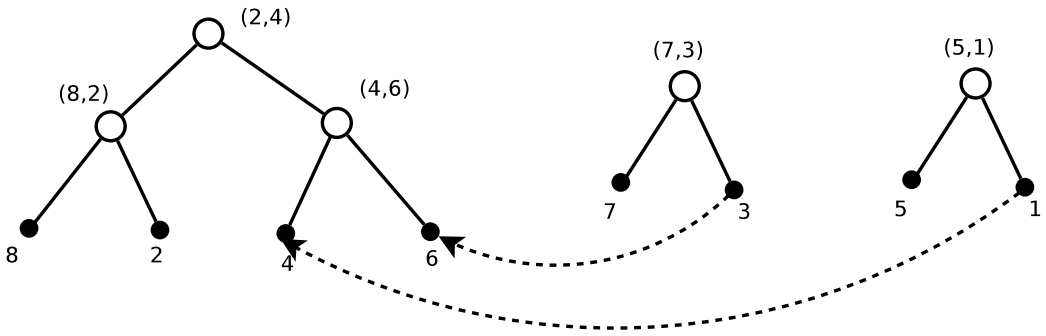

Figure 5.11: Representation of a dynamic tree as a collection of linked as binary trees

In order to store the whole dynamic tree, the binary trees of Figure 5.10 have to be linked together. For all the vertices of the dynamic tree with an outgoing dashed line (Figure 5.9), we have to connect the corresponding leaf node to the leaf node that correspond to its parent in the original dynamic tree. In practice, this can be implemented by maintaining in every node's data structure two additional fields, named *dparent* and *dcost*, that store information for the arc linking two paths and the corresponding cost for that arc. Figure 5.11 shows how the binary trees depicted in Figure 5.10 have to be connected together in order to represent the dynamic tree shown in Figure 5.9.

More fields are needed in order to store information useful for the DNEPSA algorithm. We need additional fields that store information about the flow of the arc that corresponds to an internal node and a flag indicating whether it belongs to set I_- or I_+. For every external node (leaf) of a binary tree we can also have fields to store the supplies b_i and the values of dual variable values w_i.

5.4 Theoretical time complexity per pivot for DNEPSA algorithm

The use of dynamic trees can help improve the performance of the most time consuming steps of DNEPSA. In step 0 of the algorithm, vector d is computed by using Relation (2.7). A value d_{ij} has to be computed for each arc (i,j). It is very easy to compute d_{ij}, if (i,j) belongs to the basic tree T. But, in case it is $(i,j) \notin T$, then the algorithm has to traverse the cycle C_{ij} existing in $T \cup (i,j)$ and check the orientation of every arc $(u,v) \in C_{ij}$ comparing it with the orientation of (i,j). Then d_{ij} becomes equal to the sum of $h_{ij}(u,v)$ for all $(u,v) \in I\_$, where $h_{ij}(u,v)$ is given by Relation (2.6). This is a very time consuming process. By using dynamic trees, in order to compute d_{ij}, $\forall (i,j) \in T$, it is enough to do the following steps: First, we make node j to be the root of the dynamic tree by using operation *evert(j)*. After that, we just traverse the path from node j to node i. For every arc (u,v) on that path, it is enough to check the orientation field of the node in order to find the value of $h_{ij}(u,v)$ and sum up these values for the arcs that belong to $I\_$ (which is indicated by a field of the node). Therefore, the amortized time complexity needed for the calculation of d_{ij} is the same as the time complexity of the *evert* operation used, i.e. it is O(logn). After finding the d_{ij} values for every arc (i,j), the algorithm can create the set $J\_$ by using Relation (2.8) (in step 1 of the algorithm). Then, in step 2, the algorithm can select the entering arc (g,h) by finding the minimum ratio α from Relation (2.9).

In step 3, DNEPSA has to select the leaving arc. The flows in the cycle

C, created when adding the entering arc (g,h) into the basic tree T, have to be checked so that, the values of θ_1 and θ_2 are computed by Relation (2.11). By using dynamic trees, in order to compute the values of θ_1 and θ_2, we can just make node h to be the root of the dynamic tree (by using operation *evert(h)*) and then we just traverse the path from node g up to node *root(g)* (to the root of the tree). For every arc (i,j) on that path, it is enough to check if it belongs to set I_- or to set I_+ and then compare its orientation to the orientation of the entering arc (g,h). Again, the time complexity for this process is the same as the time complexity of the *evert* operation, i.e. O(logn).

After selecting the entering and the leaving arc, the new tree-solution $T \setminus (k,l) \cup (g,h)$ has to be built. When using dynamic trees, this can be very easily done by using the *cut* and *link* dynamic tree operations. The entering arc (g,h) is added into set I_+ by setting the proper bit in the nodes structure and its flow x_{gh} becomes equal to θ_1 or to θ_2, depending on the type of iteration the algorithm performs (type A or type B iteration). The flows x_{ij} for the basic arcs $(i,j) \notin C$ do not change. On the other hand, for the basic arcs (i,j) that belong to the cycle C, their flow x_{ij} changes and its new value $x_{ij}^{(t+1)}$ has to be computed, as Table 2.2 shows. This value is either equal to $x_{ij}^{(t)} - x_{kl}^{(t)}$ or equal to $x_{ij}^{(t)} + x_{kl}^{(t)}$ depending on the orientation of arc (i,j) compared to the orientation of the entering arc (g,h). By using dynamic tree operations, in the same way as described above, it is easy to determine the orientation of an arc $(i,j) \in C$ in O(logn) time.

Vectors s and d has also to be updated for all non basic arcs $(i,j) \notin T$. If the leaving arc (k,l) is removed from the basic tree T, then two subtrees

T^+ and T^- are created, as it is shown in Figures 2.1 and 2.2. The values s_{ij} and d_{ij} do not change when both i and j belong to subtree T^+ or they both belong to subtree T^-. But, in the cases where it is $i \in T^+$ and $j \in T^-$ or $i \in T^-$ and $j \in T^+$, then it is either $s_{ij}^{(t+1)} = s_{ij}^{(t)} - s_{gh}^{(t)}$ or $s_{ij}^{(t+1)} = s_{ij}^{(t)} + s_{gh}^{(t)}$, depending on the type of iteration (type A or type B). The same happens for the new value of d_{ij} for every non basic arc $(i, j) \notin T$. By using dynamic trees, it is easy to determine for nodes i and j whether they belong to subtree T^+ or to subtree T^-. This can be done by using a cut operation for node l, i.e. perform *cut(l)*, in order to produce the two subtrees T^+ and T^-. Then by checking if *root(i)* is the same as *root(j)* it is easy to check if nodes k and l belong to same subtree. The time complexity for these tree operations is again O(logn).

Therefore, dynamic trees can be used by DNEPSA, during the whole process of the algorithm, in order to limit the time needed for the most time-consuming operations. So, the amortized time complexity per pivot is O(logn). Sleator and Tarjan in [85] also propose a more sophisticated representation of dynamic trees, that offers an O(logn) worst-case time bound. This representation though, uses very complicated data structures and in practice may be proved to be slower.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

An Exterior-point Network Simplex-type algorithm (DNEPSA) for the MCNFP problem that starts from a dual feasible solution was presented in detail here. A sequence of theorems were also proved to certify the correctness of the algorithm. The algorithm was also compared against other classic algorithms for the MCNFP problem, like DNSA, and the experimental results that show its superiority were also presented. A statistical analysis of the experimental results was also given together with some results on the empirical complexity of DNEPSA. The algorithm's implementation was also described and furthermore, a description on how the algorithm can use dynamic tree to improve its time complexity was presented. Finally, the algorithm's amortized time complexity per pivot, when using dynamic trees data structures, was analysed.

The conclusion that Exterior-point algorithms can offer a promising alter-

native approach in Network Optimization problems, can be safely extracted. It is also clear that the use of sophisticated data structures, like dynamic trees, is a key factor for the improvement on the performance of the algorithm.

6.2 Future Work

A subject for future work is the examination of the algorithm's performance when using other sophisticated data structures, other than dynamic trees, for storing and updating the necessary variables. Such data structures include Fibonacci heaps as they are described in [31]. It would be very interesting to use such data structures to implement DNEPSA and then compare its performance against some state-of-the-art implementations, like RELAX IV ([13]), combinatorial code CS2 ([48]), interior-point code DLNET ([82]), RNET ([50]) and NETFLO ([58]). The algorithm's behavior has also to be examined in some well-known pathological problem instances, as they are described in [93] and [94].

Furthermore, it would be very interesting to develop a capacitated version of DNEPSA. Generally, it is possible for any capacitated network to be transformed into an uncapacitated equivalent one by removing its arc capacities. This technique is analytically described in [2]. The only drawback of this transformation is that, it increases the number of nodes in the network. However, in most cases, the original and transformed networks can be solved by algorithms of the same time complexity. This is due to the fact that the transformed network possesses a special structure that permits us to design

more efficient algorithms.

Moreover, statistical techniques were used in order to present some experimental results on the empirical complexity of DNEPSA. The fit of both polynomials (Relations (4.2) and (4.3)) were quite good at 5% level of significance. Furthermore, high adjusted R-Squared values equal to 97.5% and 97.4% for the estimation of the number of iterations and the total cpu time respectively, provide the required validity of our experimental results. However, it is well known that the statistical measures of an algorithm's complexity do not always tally with the mathematical counterpart. Thus, it is very interesting to also derive the computational complexity of DNEPSA with rigorous theoretical proofs.

A parallelized version of the DNEPSA algorithm would be of much interest, especially after the great developments in GRID technology and parallel computing during the last years. Some efforts on the parallelization of Exterior-point Simpex-type algorithms for the Linear Problem have already been made, as it is described in [7]. Some time-consuming processes of Network Exterior-point Simplex-type algorithms (dual or primal) could probably be parallelized and, in that case, the algorithm's performance would be improved drastically.

Finally, it would be very interesting to develop a visualization software tool for DNEPSA for educational or other reasons. Similar educational tools have been already developed for other network optimization algorithms, as in [5] and [8] and it would be worthy to develop visual presentation tools for Exterior-point algorithms, like DNEPSA. Also, it would be useful to develop an on-line tool that uses DNEPSA in order to solve MCNFP net-

work instances submitted by users in some standard representation form and presents the results to the user. A software optimization suite, named *Web-NetPro*, has already been developed (presented in Karag2006) and it would be good to add DNEPSA algorithm into it.

Bibliography

[1] R. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan, Finding minimum-cost flows by double scaling, *Mathematical Programming* **53** (3) (1992) 243-266.

[2] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] R. Ahuja, T. Magnanti, J. Orlin, and M. Reddy, Applications of Network Optimization, *Handbooks of Operations Research and Management Science* **7** (1995) 1-83.

[4] R. Ahuja, J. Orlin, P. Sharma, P. Sokkalingam, A Network Simplex Algorithm with O(n) Consecutive Degenerate Pivots, *OR Letters* **30** (2002) 141-148

[5] D. Andreou, K. Paparrizos, N. Samaras, and A. Sifaleras, Visualization of the network exterior primal simplex algorithm for the minimum cost network flow problem, *Operational Research* **7**(3) (2007) 449-464.

[6] R. Armstrong and Z. Jin, A new strongly polynomial dual network simplex algorithm, *Mathematical Programming* **78**(2) (1997) 131-148.

[7] E. Badr, M. Moussa, K. Paparrizos, N. Samaras, and A. Sifaleras, Some computational results on MPI Parallel Implementation of Dense Simplex Method, *in Proc. of World Academy of Science, Engineering and Technology* **23** pp. 39-42 (presented in the Seventeenth International Conference on Computer and Information Science and Engineering (CISE 2006), 8-10 December, Cairo, Egypt, 2006)

[8] Th. Baloukas, K. Paparrizos, and A. Sifaleras, An Animated Demonstration of the Uncapacitated Network Simplex Algorithm, *INFORMS Transactions on Education* **10**(1) (2009) 34-40.

[9] R. Barr, F. Glover, and D. Klingman, Enhancements of spanning tree labelling procedures for network optimization, *INFOR* **17** (1) (1979) 16-34.

[10] E. Beale, Cycling in the dual Simplex algorithm, *Naval Res. Logist. Quart.* **2** (1955) 269-275

[11] S. Bent, D. Sleator, and R.Tarjan, Biased 2-3 trees, *Proc. Twenty-First Annual IEEE Symp. on Foundations of Computer Science* (1980) 248-254.

[12] D.P. Bertsekas, and P. Tseng, Relaxation methods for minimum cost ordinary and generalized network flow problems, *Operations Research* **36** (1) (1988) 93-114.

[13] D.P. Bertsekas, and P. Tseng, RELAX-IV: A Faster Version of the RELAX Code for Solving Minimum Cost Flow Problems, *Technical Report*,

Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, 1994.

[14] D. P. Bertsekas, Linear Network Optimization: Algorithms and Codes, *Cambridge,MA: MIT Press*, 1991.

[15] R. G. Bland, New finite pivoting rules for the simplex method, *Mathematics of Operations Research* **2** (2) (1977) 103-107.

[16] R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladnyi, An empirical study of min cost flow algorithms, *DIMACS, Series in Discrete Mathematics and Theoretical Computer Science* (1993)

[17] S. Chakraborty, and P.P. Choudhury, A statistical analysis of an algorithm's complexity, *Applied Mathematics Letters* **13**(5) (2000) 121-126.

[18] S. Chakraborty, P. P. Choudhury, Can statistics provide a realistic measure for an algorithm's complexity?, *Applied Mathematics Letters 12* (7) (1999) 113-118

[19] S. Chakraborty, and S.K. Sourabh, A Computer Experiment Oriented Approach to Algorithmic Complexity, *Lambert Academic Publishing* (2010)

[20] M. Coffin, and M.J. Saltzman, Statistical Analysis of Computational Tests of Algorithms and Heuristics, *INFORMS Journal on Computing* **12**(1) (2000) 24-44.

[21] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 3rd edition, 2009.

[22] C. Cotta, and P. Moscato, A mixed evolutionary-statistical analysis of an algorithm's complexity, *Applied Mathematics Letters* **16**(1) (2003) 41-47.

[23] W. Cunningham, A network simplex method, *Mathematical Programming* **11** (1976) 105-116

[24] W. Cunningham, Theoretical properties of the network simplex method, *Mathematics of Operations Research* **4** (1979) 196-208

[25] N. Curet, Applying steepest-edge techniques to a network primal-dual algorithm, *Computers and Operations Research* **24** (7) (1997) 601-609.

[26] N. Curet, Implementation of a steepest-edge primal-dual simplex method for network linear programs, *Annals of Operations Research* **81** (0) (1998) 251-270.

[27] G. Dantzig, Applications of the simplex method to a transportation problem, Activity Analysis of Production and Allocation, New York, Wiley, 1951.

[28] J. Edmonds, Paths, trees, and flowers, *Canadian Journal of Mathematics* **17** (1965) 449-467.

[29] J. Edmonds and, R. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the ACM* **19** (2) (1972) 248264.

[30] T.R. Ervolina, and S.T. McCormick, Two strongly polynomial cut canceling algorithms for minimum cost network flow, *Discrete Applied Mathematics* **46**(2) (1993) 133-165.

[31] M. Fredman, and R. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM* **34**(3) (1987) 596-615.

[32] K. Fukuda, and T. Terlaky, Criss-cross methods: a fresh view on pivot algorithms. *Mathematical Programming* **79**(1) (1997), 369-395.

[33] D. Fulkerson, An out-of-kilter method for minimal cost flow problems, *J. SIAM* **9** (1) (1961) 18-27.

[34] H. Gabow, and R. Tarjan, Faster scaling algorithms for network problems, *SIAM Journal on Computing* **18** (5) (1989) 1013-1036.

[35] Z. Galil, and E. Tardos, A $o(n^2(m+nlogn)logn)$ min-cost flow algorithm, *Journal of the ACM* **35** (2) (1988) 374-386.

[36] S. Gassa, and S. Vinjamurib Cycling in linear programming problems, textitComputers and Operations Research **31** (2) (2004) 303311.

[37] G. Geranis, K. Paparrizos, and A. Sifaleras, A dual exterior point simplex type algorithm for the minimum cost network flow problem, *Yugoslav Journal of Operations Research* **19**(1) (2009) 157-170.

[38] G. Geranis, K. Paparrizos, and A. Sifaleras, On the Computational Behavior of a Dual Network Exterior Point Simplex Algorithm for the Minimum Cost Network Flow Problem, *Proceedings of the International*

Network Optimization Conference (INOC 2009), 26-29 April, Pisa, Italy, (2009).

[39] F. Glover, D. Klingman, and A. Napier, Basic Dual Feasible solutions for a Class of Generalized Networks, *Operations Research* **20**(1) (1972) 126-136.

[40] F. Glover, D. Klingman, and A. Napier, An efficient dual approach to network problems, *Opsearch* **9** 1 (1972) 1-18.

[41] F. Glover, D. Karney, and D. Klingman, The augmented predecessor index method for locating stepping stone paths and assigning dual prices in distribution problems, *Transportation Science* **6**(2) (1972) 171-180.

[42] F. Glover, D. Klingman, and J. Stutz, Extensions of the augmented predecessor index method to generalized transportation problems, *Transportation Science* **7** (4) (1973) 377-384.

[43] F. Glover, D. Klingman, and J. Stutz, Augmented threaded index method for network optimization, *INFOR* **12** (3) (1974) 293-298.

[44] F. Glover, D. Karney, D. Klingman, and A. Napier, A computation study on start procedures, basis change criteria, and solution algorithms for transportation problems, *Management Science* **20** (5) (1974) 793-813.

[45] F. Glover, D. Klingman, and N. Phillips, *Network Models in Optimization and Their Applications in Practice*, Wiley Publications, 1992.

[46] A. Goldberg, and R. Tarjan, Finding minimum-cost circulations by canceling negative cycles, *Journal of the ACM* **36** (4) (1989) 873-886.

[47] A. Goldberg, E. Tardos, and R. Tarjan, Network flow algorithms, *Technical report*, Department of Computer Science, Stanford University (1989).

[48] A.V. Goldberg, An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm, *Journal of Algorithms* **22**(1) (1997) 1-29.

[49] A. Goldberg, M. Grigoriadis, and R.E. Tarjan, Use of dynamic trees in a network simplex algorithm for the maximum flow problem, *Mathematical Programming* **50**(3) (1991) 277-290.

[50] M. Grigoriadis, An efficient implementation of the network simplex method, *Mathematical Programming Studies* **26** (1984) 83-111.

[51] J. Hall, and K. McKinnon, The simplest examples where the simplex method cycles and conditions where expand fails to prevent cycling, *Mathematical Programming* **100** (1) (2004) 133-150.

[52] A. Hoffman, Cycling in the Simplex method, *Techn. Report Nat. Bureau Standards* **2974** (1953)

[53] J. Hultz and D. Klingman, An Advanced Dual Basic Feasible Solution for a Class of Capacitated Generalized Networks, *Operations Research* **24**(2) (1976) 301-313.

[54] P. Karagiannis, K. Paparrizos, N. Samaras, and A. Sifaleras, A short bibliography on the assingment problem and its applications, *Technical*

report, Department of Applied Informatics, University of Macedonia, Deliverable in Work Package 1, from the Research Project (EPEAEK II) PYTHAGORAS I (2004).

[55] P. Karagiannis, K. Paparrizos, N. Samaras, and A. Sifaleras, A new simplex type algorithm for the minimum cost network flow problem, *in Proc. of the 7th Balkan Conference on Operational Research (BACOR 05)*, 25-28 May, Constanta, Romania, pp. 133-139, 2005.

[56] P. Karagiannis, I. Markelis, K. Paparrizos, N. Samaras, and A. Sifaleras, E-learning technologies: employing matlab web server to facilitate the education of mathematical programming, *International Journal of Mathematical Education in Science and Technology* **37** (7) (2006) 765-782.

[57] N. Karmarkar, A new polynomial time algorithm for linear programming, *Combinatorica* **4** (4) (1984) 373-395.

[58] J. Kennington and R. Helgason, *Algorithms for Network Programming*, Wiley, New York, 1980.

[59] D. Klingman, A. Napier, and J. Stutz, NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow networks, *Management Science* **20**(5) (1974) 814-821.

[60] Y. Lee, and J. B. Orlin, Computational testing of a network simplex algorithm, *Proceedings of the First DIMACS International Algorithm Implementation Challenge* (1991).

[61] Y. Lee, Computational analysis of network optimization algorithms, *Ph.D. thesis* M.I.T. (1993).

[62] I. Maros, A practical anti-degeneracy row selection technique in network linear programming, *Annals of Operations Research* **47**(2) (1993) 431-442.

[63] C.C. McGeoch, A Guide to Experimental Algorithmics, *Cambridge University Press* (2012)

[64] C.C. McGeoch, Toward an experimental method for algorithm simulation, *INFORMS Journal on Computing* **8**(1) (1996) 1-15.

[65] C. McGeoch, P. Sanders, R. Fleischer, P.R. Cohen, and D. Precup, Using finite experiments to study asymptotic performance, *Lecture Notes In Computer Science*, Vol. 2547, Springer-Verlag New York, Inc., (2002) 93-126

[66] R. Nance, R. Moose, and R. Foutz, A statistical technique for comparing heuristics: an example from capacity assignment strategies in computer network design, *Communications of the ACM*, **30**(5) (1987) 430-442.

[67] J.B. Orlin, Genuinely Polynomial Simplex and Non-Simplex Algorithms for the Minimum Cost Flow Problem, *Technical Report No. 1615-84* Sloan School of Management, M.I.T., Cambridge, MA, (1984).

[68] J.B. Orlin, A faster strongly polynomial minimum cost flow algorithm, *Operations Research* **41** (2) (1993) 338-350.

[69] J.B. Orlin, S. Plotkin, and E. Tardos, Polynomial dual network simplex algorithms, *Mathematical Programming* **60** (3) (1993) 255-276.

[70] J.B. Orlin, A polynomial time primal network simplex algorithm for minimum cost flows, *Mathematical Programming* **78** (2) (1997) 109-129.

[71] C. Papamanthou, K. Paparrizos, and N. Samaras, On the initialization methods of an exterior point algorithm for the assignment problem, *International Journal of Computer Mathematics*, Taylor and Francis Publications, **87** (8) (2010) 1831-1846.

[72] K. Paparrizos, An infeasible (exterior point) simplex algorithm for assignment problems, *Mathematical Programming* **51** (1) (1991) 45-54.

[73] K. Paparrizos, A non improving simplex algorithm for transportation problems, *RAIRO Operations Research* **30** (1) (1996) 1-15.

[74] K. Paparrizos, Exterior point simplex algorithms: simple and short proofs of correctness, *Proceedings of the XXIII SYMOPIS*, Zlatibor (1996) 13-18.

[75] K. Paparrizos, N. Samaras, and G. Stephanides, An efficient simplex type algorithm for sparse and dense linear programs, *European Journal of Operational Research* **148** (2) (2003) 323-334.

[76] K. Paparrizos, N. Samaras, and A. Sifaleras, A learning tool for the visualization of general directed or undirected rooted trees, *K. Morgan and J. M. Spector (Eds.)*, WIT Transactions on Information and

Communication Technologies, Volume 30, Chapter The Internet Society: Advances in Learning, Commerce and Security, Skiathos, Greece: WIT Press (2004), pp. 205-213,.

[77] Paparrizos K., Samaras N., and Sifaleras A., Network Optimization (in Greek), *ZYGOS Publications*, ISBN: 978-960-8065-68-0, Thessaloniki, 2009.

[78] K. Paparrizos, N. Samaras, and A. Sifaleras, An exterior Simplex type algorithm for the minimum cost network flow problem, *Computers & Operations Research* **36**(4) (2009) 1176-1190.

[79] K. Paparrizos, N. Samaras, and A. Sifaleras, On the empirical behaviour of a new network exterior point simplex algorithm for the minimum cost network flow problem, *Proceedings of the Veszprem Optimization Conference: Advanced Algorithms (VOCAL 2006)*, Veszperm Hungary, (2006).

[80] S. Plotkin, and E. Tardos, Improved dual network simplex, *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, (1990) 367-376.

[81] T. Radzik, Implementation of Dynamic Trees with In-Subtree Operations, *Department of Computer Science, Kings College, London*, Technical Report (1998).

[82] M. Resende, and G. Veiga, An efficient implementation of a network interior point method, D.S. Johnson and C.C. McGeoch, (Eds.), *Network flows and matching: First DIMACS implementation challenge*, **12**,

DIMACS series in discrete mathematics and theoretical computer science, American Mathematical Society, Providence, Rhode Island, (1993) 299-348.

[83] H. Rock, Scaling techniques for minimal cost network flows, *Munich: C. Hansen* (1980) 181-191.

[84] A. Sifaleras, Development and Implementation of Exterior Point Simplex Type Algorithms for Network Optimization Problems, *Ph.D. Thesis*, University of Macedonia, Department of Applied Informatics, Thessaloniki, Greece, (2007).

[85] D. Sleator, and R. Tarjan, A data structure for dynamic trees, *Journal of Computer and System Sciences* **26** (1985) 362-391.

[86] D. Sleator, and R. Tarjan , Self-adjusting binary search trees, *Journal of the Association of Computer Machinery* **32** (1985) 652-686.

[87] P. Sokkalingam, R. Ahuja, and J. Orlin, New polynomial-time cycle-canceling algorithms for minimum cost flows, *Networks* **36** (1) (2000) 53-63.

[88] E. Tardos, A strongly polynomial minimum cost circulation algorithm. *Combinatorica* **5** (3) (1985) 247-255.

[89] R. Tarjan, Data Structures and Network Algorithms, *Society for Industrial and Applied Mathematics*, Philadelphia, PA (1983).

[90] R. Tarjan, Amortized computational complexity, *SIAM Journal on Algebraic and Discrete Methods* **6** (1985) 306-318.

[91] R. Tarjan, Dynamic Trees as Search Trees via Euler Tours, Applied to the Network Simplex Algorithm, *Mathematical Programming* **78**(2) (1997) 169-177.

[92] R. Vanderbei, *Linear Programming: Foundations and Extensions*, Springer, 3rd Ed. New York, NY, 2007.

[93] N. Zadeh, More Pathological Examples for Network Flow Problems, *Mathematical Programming* **5**(1) (1973) 217-224.

[94] N. Zadeh, A bad network problem for the simplex method and other minimum cost flow algorithms, *Mathematical Programming* **5**(1) (1973) 255-266.

[95] P. Zörnig, Systematic construction of examples for cycling in the simplex method, *Computers and Operations Research* **33** (8) (2006) 2247-2262.