CHALMERS | GÖTEBORG UNIVERSITY

MASTER'S THESIS

Lagrangian Heuristics for Strictly Convex Quadratic Minimum Cost Network Flow Problems

Caroline Olsson

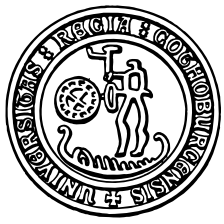
Department of Mathematics
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG UNIVERSITY
Göteborg Sweden 2005

Thesis for the Degree of Master of Science

Lagrangian Heuristics for Strictly Convex Quadratic Minimum Cost Network Flow Problems

Caroline Olsson

CHALMERS | GÖTEBORG UNIVERSITY



Department of Mathematics
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Göteborg, October 2005

Matematiskt centrum
Göteborg 2005

Abstract-

This thesis presents a study of five different Lagrangian heuristics applied to the strictly convex quadratic minimum cost network flow problem. Tests are conducted on randomly generated transportation networks with different degrees of sparsity and nonlinearity according to a system devised by Ohuchi and Kaji [18]. The different heuristics performance in time and quality are compared. The unconstrained dual version of the problem is first solved to near-optimality using the conjugate gradient method with an exact line search. Then a Lagrangian heuristic is applied to obtain (near-optimal) primal solutions to the original problem. In the computational study, we show results for two modifications of the Lagrangian heuristic Flowroute, FlowrouteBS and FlowrouteD, and one modification of the Lagrangian heuristic Shortest Path, Shortest PathL. FlowrouteBS, FlowrouteD and Shortest PathL are novel Lagrangian heuristics, but Flowroute and Shortest Path are constructed according to Marklund [15]. The results demonstrate that although FlowrouteBS has the drawback of being significantly slower than Flowroute and FlowrouteD, it produces results of almost as good quality as Shortest Path and Shortest PathL, and is therefore the most promising Lagrangian heuristic.

KEYWORDS: duality, Lagrangian heuristics, strictly convex quadratic minimum cost network flow problem

Sammanfattning

Detta examensarbete presenterar en studie av fem olika Lagrangeheuristiker applicerade på minkostnadsproblemet för nätverk med strikt konvex, kvadratisk kostnad. Tester har genomförts på slumpmässigt genererade nätverk med olika grader av gleshet och ickelinjäritet enligt ett system givet av Ohuchi and Kaji [18]. De olika heruristikernas prestanda jämförs med avseende på tid och kvalitet. Den obegränsade duala versionen av problemet löses först till näroptimalitet med en konjugatgradient metod i kombination med en exakt linjesökning. Sedan appliceras en Lagrageheuristik för att uppnå (näroptimla) primala lösningar till det ursprungliga problemet. Vi presenterar resultat för två modifieringar av Lagrangeheuristiken Flowroute, FlowrouteBS och FlowrouteD, samt en modifiering av Lagrangeheuristiken Shortest Path, Shortest PathL. FlowrouteBS, FlowrouteD samt Shortest PathL är tidigare okända Lagrangeheuristiker, men Flowroute och Shortest Path är presenterade av Marklund [15]. Resultaten visar att FlowrouteBS producerar resultat av nästan lika god kvalitet som Shortest Path och Shortest PathL och är därmed den mest lovande Lagrangeheuristiken, trots att den har nackdelen att vara signifikant långsammare än Flowroute och FlowrouteD.

NYCKELORD: dualitet, Lagrangeheuristiker, minkostnadsproblemet för nätverk med strikt konvex, kvadratisk kostnad

Acknowledgments

This thesis completes my studies in Mathematics and Computing Science at the University of Gothenburg. I wish to express my sincere gratitude to my supervisor Professor Michael Patriksson at the Department of Mathematics, and to Ph. D. student Tapani Utrainen at the Department of Computing Science. The former for taking me on board, giving me guidance when I got stuck, and patience with the endless delays; the latter for helpful hints when I struggled with C.

Behind every woman who achieves success stands a man, a daughter and the IRS (or whatever the saying is), and even if there at times can be doubts about the first statement in my case, there is no doubt about the second and third. I consider myself very lucky to have had support from my family throughout the years in school. I could not have made it without you, and I am happy to say that payback time is finally coming in every way. Love always!

Göteborg, 28th September 2005
CAROLINE OLSSON

Contents

1 Introduction

The problem under consideration in this thesis is the minimum cost network flow problem with a strictly convex quadratic cost function. In the minimum cost network flow problem we aim to find the least cost of transporting a commodity through a network in order to satisfy demands at certain vertices from available supplies at other vertices. The arcs of the network have a cost associated to them, as well as limits of how much flow that can be transported through them. There are several applications within engineering, economics, and statistics when the cost of the arcs varies quadratically with the amount of flow. Such situations occur for instance in resistive electrical networks, equilibrium import-export trade problems, quadratic data-fitting problems, and in urban traffic flows (Ventura [22]) . These problems have in common that they are large-scale in structure, with many thousands of variables and constraints. Thus, they are very complex to solve. Moreover, they tend to grow larger and more complex with time, and the algorithms commonly used to solve them are approaching the limit of what is feasible in terms of CPU time (for instance Ventura [22] gives a summary of such algorithms).

In this thesis, an approach involving five Lagrangian heuristics is used to solve the minimum cost network flow problem with a strictly convex quadratic cost function. The unconstrained dual version of the problem is first solved to near-optimality using the conjugate gradient method with an exact line search, followed by the application of a Lagrangian heuristic to obtain (near-optimal) primal feasible solutions to the original problem. When the conjugate gradient method is stopped at a high level of accuracy, we expect that the quality in solution for the Lagrangian heuristics will improve compared to when it is stopped at a low level of accuracy. We base this assumption on Marklund [15], Theorem 9, and on the experimental convergence studies also presented by Marklund [15]. Randomly generated transportation networks of different degrees of sparsity and nonlinearity will be generated according to a system by Ohuchi and Kaji [18]. We expect that the running times for the Lagrangian heuristics will be shorter when applied to smaller networks, and likewise to more dense networks. The first assumption is based on the fact that solving problems for larger networks involve more computations, and this inevitable leads to longer solution times. The second assumption is based on the fact that solving problems for more dense networks is easier, i.e. the running time becomes shorter, since there are many options to reroute flow in the network. These assumptions have also been verified by the experimental studies of the Lagrangian heuristics Flowroute and Shortest Path by Marklund [15]. In this thesis, we suggest

two new modifications of Flowroute, FlowrouteD and FlowrouteBS, and one new modification of Shortest Path, Shortest PathL. Due to the modifications made to Flowroute and Shortest Path, we expect that FlowrouteD and FlowrouteBS will produce results of similar quality, but faster than Flowroute, and that Shortest PathL will produce results in similar running time as, but with better quality than Shortest Path.

The contents and outline of this thesis is the following: first several terms that are commonly used when dealing with network problems are introduced. This is followed by a discussion on Lagrangian duality, and methods of how to solve the Lagrangian dual problem. Hereafter, the different Lagrangian heuristics are presented, followed by a description of the data structure used to model the problem, and the network generator used to generate large-scale transportation networks. Eventually the computational results of using the different Lagrangian heuristics on large-scale networks are presented. The thesis ends with a discussion of the findings, and the conclusions drawn when comparing the different Lagrangian heuristic performance in time and quality.

2 Terminology and notation

This section presents many of the terms that will be encountered throughout the thesis. The contents rests on the facts presented in the text books by Ahuja [1], Biggs [5], Cormen [6], Evans [9], and Migdalas&Göthe–Lundgren [16].

2.1 Graphs, networks, and flows

A **graph**, G, consists of two finite sets; **vertices**, V, and **edges**, E. The vertices are the connection points of the graph. Two vertices are joined by an edge. A **bipartite graph** has the set of vertices partitioned in two subsets V_1 and V_2 so that for each edge $\{i, j\} \in E$ either $i \in V_1$ and $j \in V_2$ or $j \in V_1$ and $i \in V_2$. A vertex is usually identified by a label or coordinate and an edge by a weight or cost. The **degree** of a vertex is the number of edges connected to it. An edge can be directed or undirected depending on whether it is traversed in a given direction only or in both. A directed edge is called an **arc** and is identified by its origin (tail), and its destination (head). The set of arcs is usually denoted A. An arc between a specific vertex $i \in V$ and an adjacent vertex $j \in V$, is either forward (outgoing) if it connects (i, j) or backward (incoming) if it connects (j, i). A graph that does not contain any undirected edges is called a **digraph**.

A **walk** in a graph is a sequence of vertices $1, 2, \ldots, k$ such that $i, i+1$ are adjacent for i, $1 \leq i \leq k-1$. Vertex 1 is called the origin and vertex k is called the destination of the walk. If all the vertices are distinct, the walk is called a **path**. A path that starts and ends at the same vertex is called a **cycle**. The **length** of a walk, path or cycle is the number of edges on the walk, path or cycle respectively. A graph that contains no directed cycles is called **acyclic**.

An **acyclic layered digraph**, ALD, has the set V divided into $l \geq 2$ different subsets, so called levels, $L(k)$, for some $k \in \boldsymbol{Z}^+$, $0 \leq k \leq l-1$. Each vertex is encountered exactly once at a certain level. For an arc (i, j) in an ALD, if the vertex i is found at $L(k)$, then vertex j is found at $L(k+1)$.

A **network**, N, is a digraph where all arcs have a weight associated to them. A **flow network** is a network where all the arcs have, in addition to a weight, a nonnegative capacity, $c(i, j)$ associated with them. The flow networks we will study in this thesis, transport one commodity only. There are two specific types of vertices in a flow network called a **source**, s, and a **sink**, t. The source is the origin of the flow which provides a **supply**, and the sink is the destination of the flow that possesses a **demand**. A pure source (sink) has outgoing (incoming) arcs only. A vertex that has neither supply

nor demand is called a pure **transshipment**. A minimum cost flow network can consist of several sources and sinks, but it can be reduced to a regular flow network by the introduction of a super-source and a super-sink.

A **flow** in a network is a real-valued function $f : A \to \Re$ that assigns a nonnegative value to each arc of the network and satisfies the following two properties:

$$\textit{(i) Capacity constraint: } f(i,j) \leq c(i,j), \qquad \forall (i,j) \in A,$$

$$\textit{(ii)Flow conservation: } \sum_{k \in V} f(k,i) = \sum_{j \in V} f(i,j), \qquad \forall i \in V \backslash \{s,t\}.$$

The **value** of a flow is defined as the total flow out of the source or, equivalently, the total flow into the sink. An **augmenting path** with respect to a given flow f is a directed path from the source to the sink along which more flow can be transported.

3 Lagrangian duality

The problem under consideration is the separable minimum cost network flow problem with a strictly convex quadratic cost function for the network $N = (V, A)$. It is formulated as follows:

$$\underset{\boldsymbol{v}}{\text{minimize}} \sum_{(i,j)\in A} \Phi_{ij}(v_{ij}) = \sum_{(i,j)\in A} b_{ij}v_{ij}^2 + a_{ij}v_{ij}, \qquad \forall (i,j) \in A, \quad (1)$$

$$\text{subject to} \sum_{j:(i,j)\in A} v_{ij} - \sum_{j:(j,i)\in A} v_{ji} = d_i, \qquad \forall i \in V, \quad (2)$$

$$l_{ij} \leq v_{ij} \leq u_{ij}, \qquad \forall (i,j) \in A. \quad (3)$$

We want to solve the problem indirectly by using its dual formulation:

$$\underset{\boldsymbol{\pi}}{\text{maximize}} \left\{ -\sum_{i\in V} \pi_i d_i + \underset{\boldsymbol{l}\leq \boldsymbol{v}\leq \boldsymbol{u}}{\text{minimum}} \left\{ \sum_{(i,j)\in A} b_{ij}v_{ij}^2 + (a_{ij} + \pi_i - \pi_j)v_{ij} \right\} \right\} \quad (4)$$

$$\text{subject to } \boldsymbol{\pi} \in \Re^{|V|}. \quad (5)$$

In the following sections, several theorems will be presented that explain why solving the dual version of the problem is easier than solving the original problem. The theory for this section is taken from the text books by Andréasson et al. [2], Bazaraa et al. [3], and Bertsekas [4].

3.1 Duality theorems and properties of the dual function

Consider the general nonlinear network flow problem for the network $N = (V, A)$:

$$\underset{\boldsymbol{v}}{\text{minimize}}\ \Phi(\boldsymbol{v}), \quad (6)$$

$$\text{subject to } \boldsymbol{E}\boldsymbol{v} = \boldsymbol{d}, \quad (7)$$

$$\boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}, \quad (8)$$

where $\Phi : \Re^{|A|} \to \Re$ is a nonlinear, continuous function; $\boldsymbol{l} \in \Re^{|A|}$, $\boldsymbol{u} \in \Re^{|A|}$, and $\boldsymbol{d} \in \Re^{|V|}$ are constant vectors; $\boldsymbol{E} \in \Re^{|V|\times|A|}$ is the node-arc incidence matrix of N; $\boldsymbol{v} \in \Re^{|A|}$ is the vector of decision variables that represent the flows on the arcs. It is assumed that $-\infty < \Phi^* < \infty$, $l_{ij} < u_{ij}, \forall (i,j) \in A$, and that $\sum_{i\in V} d_i = 0$. The set $\left\{\boldsymbol{v} \in \Re^{|A|} : \boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}\right\}$ will be denoted X; the set $\left\{\boldsymbol{v} \in \Re^{|A|} : \boldsymbol{E}\boldsymbol{v} = \boldsymbol{d}; \boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}\right\}$ will be denoted F. The statements (6)–(8) will together be denoted the primal problem. For the vector of Lagrange

multipliers, $\boldsymbol{\pi} \in \Re^{|V|}$ of the relaxed constraint (7), we define the Lagrange function, $L(\boldsymbol{v}, \boldsymbol{\pi})$, as:

$$L(\boldsymbol{v}, \boldsymbol{\pi}) = \Phi(\boldsymbol{v}) + \boldsymbol{\pi}^T(\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d}).$$

The Lagrangian dual function, whose evaluation also is known as the Lagrangian dual subproblem, is defined as:

$$\eta(\boldsymbol{\pi}) = \underset{\boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}}{\text{minimum}} \;\; L(\boldsymbol{v}, \boldsymbol{\pi}), \qquad \boldsymbol{\pi} \in \Re^{|V|}. \tag{9}$$

Consider then the Lagrangian dual formulation of the general network flow problem which will be denoted the dual problem:

$$\underset{\boldsymbol{\pi}}{\text{maximize}} \;\; \eta(\boldsymbol{\pi}) \tag{10}$$

$$\text{subject to } \boldsymbol{\pi} \in \Re^{|V|}. \tag{11}$$

Any feasible solution to the dual problem will always constitute a lower bound to any feasible solution of the primal problem. This relation is commonly known as weak duality, and is stated in Theorem 1.

Theorem 1 *Let $\boldsymbol{v}^*$ be an optimal solution to the primal problem, and let $\boldsymbol{\pi}^*$ be an optimal solution to the dual problem. Then,*

$$\eta(\boldsymbol{\pi}^*) \leq \Phi(\boldsymbol{v}^*).$$

Proof Take any dual feasible $\boldsymbol{\pi} \in \Re^{|A|}$. By definition,

$$\eta(\boldsymbol{\pi}) = \underset{\boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}}{\text{minimum}} \;\; \left\{\Phi(\boldsymbol{v}) + \boldsymbol{\pi}^T(\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d})\right\}.$$

It follows that for any primal feasible $\boldsymbol{v} \in \Re^{|V|}$

$$\eta(\boldsymbol{\pi}) \leq \Phi(\boldsymbol{v}) + \boldsymbol{\pi}^T(\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d}).$$

Moreover, for such a primal feasible $\boldsymbol{v}$, $\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d} = \boldsymbol{0}$. Hence,

$$\eta(\boldsymbol{\pi}) \leq \Phi(\boldsymbol{v}) + \boldsymbol{\pi}^T(\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d}) \leq \Phi(\boldsymbol{v}).$$

In particular,

$$\eta(\boldsymbol{\pi}) \leq \eta(\boldsymbol{\pi}^*) \leq \Phi(\boldsymbol{v}^*) \leq \Phi(\boldsymbol{v}). \qquad \blacksquare$$

Now we know that there is an association between the primal problem and the dual problem, and more importantly, that it seems as that they can have the same solution. Could it be that the dual problem has properties that makes it easier to solve than the primal problem, and that the primal solution somehow can be obtained from the dual solution? To be able to answer those questions, we begin with the matter of what the Lagrangian dual function looks like. Theorem 2 states that the dual function is concave.

Theorem 2 *The Lagrangian dual function, η, is a concave function of $\boldsymbol{\pi}$.*

Proof Take any $\boldsymbol{v}$, $\boldsymbol{\pi}_1$ and $\boldsymbol{\pi}_2$, and $\alpha \in [0,1]$. We have that

$$L(\boldsymbol{v}, (\alpha\boldsymbol{\pi}_1 + (1-\alpha)\boldsymbol{\pi}_2)) = \alpha L(\boldsymbol{v}, \boldsymbol{\pi}_1) + (1-\alpha)L(\boldsymbol{v}, \boldsymbol{\pi}_2).$$

If we take the minimum over $\boldsymbol{v}$ on both sides for $\boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}$, the relation becomes:

$$\begin{aligned}\underset{v}{\text{minimum}}\ \ L(\boldsymbol{v}, (\alpha\boldsymbol{\pi}_1+(1-\alpha)\boldsymbol{\pi}_2)) &= \underset{v}{\text{minimum}}\ \ \{\alpha L(\boldsymbol{v}, \boldsymbol{\pi}_1)+(1-\alpha)L(\boldsymbol{v}, \boldsymbol{\pi}_2)\} \\ &\geq \underset{v}{\text{minimum}}\ \ \alpha L(\boldsymbol{v}, \boldsymbol{\pi}_1) \\ &\quad + \underset{v}{\text{minimum}}\ \ L(\boldsymbol{v}, (1-\alpha)\boldsymbol{\pi}_2),\end{aligned}$$

since taking the minimum over the two functions of the RHS gives us a freedom of choice for $\boldsymbol{v}$. It is therefore possible to obtain lower values in the RHS than in the single function of the LHS. ∎

The concavity of the Lagrangian dual function means that a local maximum of η also is a global maximum of η. Hence, solving the dual problem is equivalent to maximizing a concave function over $\Re^{|A|}$. Provided that the dual problem is differentiable, solving the dual problem is an easier task than solving the primal problem.

Next, we are especially interested in situations when existence of optimal solutions in both the dual and the primal problem are guaranteed, and when the optimal values of the two problems are equal, i.e., when a duality gap is absent. These situations occur if we impose stronger conditions on the objective function in the primal problem. Firstly, we address the relation $\eta(\boldsymbol{\pi}^*) = \Phi(\boldsymbol{v}^*)$, commonly known as strong duality, which is stated in Theorem 3.

Theorem 3 *Let $\Phi : \Re^{|V|} \to \Re$ be a convex function of $\boldsymbol{v}$ such that $-\infty < \Phi^* < \infty$. Then there exists a feasible $\boldsymbol{v} \in X$ such that $\boldsymbol{E}\boldsymbol{v} = \boldsymbol{d}$, and*
** there exists at least one optimal solution, $\boldsymbol{\pi}^*$, to the dual problem;*
** if there exists an optimal solution, $\boldsymbol{v}^*$, to the primal problem, then:*
(1) $\boldsymbol{v}^* \in$ arg minimum $_v$ $L(\boldsymbol{v}, \boldsymbol{\pi}^*)$
(2) $\boldsymbol{v}^* \in X$
(3) $\boldsymbol{E}\boldsymbol{v}^* = \boldsymbol{d}$;
** there is no duality gap.*

Proof Andréasson et al. [2], Theorem 6.10. ∎

The objective function of our problem is convex, and satisfies $-\infty < \Phi^* < \infty$, so we are guaranteed that there exist at least one optimal solution to the dual problem. However, strong duality holds for our problem only if there exist an optimal solution to the primal problem. Our primal objective function is not only convex, it is strictly convex and continuous, and minimized over the set X which is non-empty and closed. By Weierstrass' Theorem (Andréasson et al. [2], Theorem 4.7), we are therefore assured that there exists a primal optimal solution. Knowing that strong duality holds for our problem, the optimal primal vector $\boldsymbol{v}^*$ is obtained by solving the Lagrangian dual subproblem for the vector $\boldsymbol{\pi}^*$. Moreover, our primal objective function is quadratic, and the expression for $\boldsymbol{v}^*$ is thus available in closed form:

$$\boldsymbol{v}^* = -2\boldsymbol{B}^{-1}(\boldsymbol{a} + \boldsymbol{E}^T\boldsymbol{\pi}^*), \tag{12}$$

where $\boldsymbol{B} \in \Re^{|A|\times|A|}$ is the diagonal matrix of strictly positive coefficients, and $\boldsymbol{a} \in \Re^{|A|}$ is the constant vector of linear coefficients.

What remains to determine is if our dual problem really is differentiable. Theorem 4 presents the conditions that must be satisfied for this to be true:

Theorem 4 *Let $\Phi(\boldsymbol{v}) : \Re^{|V|} \to \Re$ be a strictly convex function on a convex set X. Then the Lagrangian subproblem has a unique solution, $\boldsymbol{v}(\boldsymbol{\pi})$ for all $\boldsymbol{\pi} \in \Re^{|V|}$, and*

$$\bigtriangledown\eta(\boldsymbol{\pi}) = \boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}) - \boldsymbol{d}.$$

Proof Andréasson et al. [2], Proposition 6.20. ∎

Theorem 4 is applicable to our problem, since our primal objective function is strictly convex, and the set X is convex. It follows that our Lagrangian dual subproblem has a unique solution, and we can conclude that our dual problem is differentiable.

4 Gradient methods for solving the dual problem

Our Lagrangian dual problem is concave, and differentiable. Therefore, it can be solved by any gradient method that provides dual ascent. The essence of such an algorithm is to find a direction $\boldsymbol{p}^\tau$, and a step size $\alpha^\tau > 0$ in every iteration τ such that the objective function value improves. The direction $\boldsymbol{p}^\tau$ is chosen such that the dual directional derivative in the direction $\boldsymbol{p}^\tau$, $\eta'(\boldsymbol{\pi}^\tau; \boldsymbol{p}^\tau) = \bigtriangledown\eta(\boldsymbol{\pi}^\tau)^T\boldsymbol{p}^\tau > 0$, and the step size α^τ such that $\eta(\boldsymbol{\pi}^\tau + \alpha^\tau\boldsymbol{p}^\tau) > \eta(\boldsymbol{\pi}^\tau)$. However, to obtain a solution for $\boldsymbol{\pi}^\tau$, we also need a solution $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ to the Lagrangian dual subproblem. The generic algorithm for a gradient method that provides dual ascent for our problem is shown below:

Gradient method$(\eta, \boldsymbol{\pi}^0)$
1. determine $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ by solving the Lagrangian dual subproblem
2. **if** $\frac{||\bigtriangledown\eta(\boldsymbol{\pi}^\tau)||}{||\boldsymbol{d}||} = \frac{||\boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau)-\boldsymbol{d}||}{||\boldsymbol{d}||} < \epsilon$ ($\epsilon > 0$ is a small tolerance)
3. **then goto** 9.
4. **else** determine $\boldsymbol{p}^\tau$ such that $\bigtriangledown\eta(\boldsymbol{\pi}^\tau)^T\boldsymbol{p}^\tau > 0$
5. determine $\alpha^\tau > 0$ such that $\eta(\boldsymbol{\pi}^\tau + \alpha^\tau\boldsymbol{p}^\tau) > \eta(\boldsymbol{\pi}^\tau)$
6. $\boldsymbol{\pi}^{\tau+1} = \boldsymbol{\pi}^\tau + \alpha^\tau\boldsymbol{p}^\tau$
7. $\tau = \tau + 1$
8. **goto** 1.
9. **return** $\boldsymbol{\pi}^\tau$

We will present two different methods for selecting the direction of ascent, and two methods for selecting the step size. The steepest ascent method with the Armijo step size rule is a dual scheme that is guaranteed to generate a sequence $\{\boldsymbol{\pi}^\tau\}$ that converge to the optimal solution $\{\boldsymbol{\pi}^*\}$ (Theorem 11.4, Andréasson et al. [2]). The conjugate gradient method with the Helgason step size rule also gives a convergence guarantee (Proposition 1:2:1, Bertsekas [4]). We will also present one periodic basis ascent method, the coordinate ascent method. The theory for this section is presented according to the findings of Ventura [22] and Helgason [11], and according to the facts in the textbooks by Andréasson et al. [2], Bertsekas [4], Heath [10], and Nash&Shofer [17].

4.1 Solving the Lagrangian dual subproblem

To make the Lagrangian dual subproblem easy to solve, we impose another condition on the objective function of the primal problem. We let the primal objective function of our problem be separable. Thus, the Lagrange function for our problem is formulated:

$$L(\boldsymbol{v}, \boldsymbol{\pi}) = \sum_{(i,j)\in A} b_{ij}v_{ij}^2 + a_{ij}v_{ij} + \sum_{i\in V} \pi_i \left(\sum_{j:(i,j)\in A} v_{ij} - \sum_{j:(j,i)\in A} v_{ji} - d_i \right) \quad (13)$$

$$= \sum_{(i,j)\in A} b_{ij}v_{ij}^2 + (a_{ij} + \pi_i - \pi_j)v_{ij} - \sum_{i\in V} \pi_i d_i, \quad (14)$$

and the Lagrangian dual subproblem becomes:

$$\eta(\boldsymbol{\pi}) = \underset{\boldsymbol{l}\le \boldsymbol{v}\le \boldsymbol{u}}{\text{minimum}} \; L(\boldsymbol{v}, \boldsymbol{\pi}) \quad (15)$$

$$= -\sum_{i\in V} \pi_i d_i + \underset{\boldsymbol{l}\le \boldsymbol{v}\le \boldsymbol{u}}{\text{minimum}} \left\{ \sum_{(i,j)\in A} t_{ij}(v_{ij}, \pi_i, \pi_j) \right\}, \quad (16)$$

where $t_{ij}(v_{ij}, \pi_i, \pi_j) = b_{ij}v_{ij}^2 + (a_{ij} + \pi_i - \pi_j)v_{ij}$. We refer to t_{ij} as the dual cost function over arc (i, j).

Solving our Lagrangian dual subproblem for an iteration τ, means performing $|A|$ minimizations over the dual cost function of each arc (i, j), adding these results together, and then subtracting the term $(\boldsymbol{\pi}^\tau)^T \boldsymbol{d}$. The minimum of the dual cost function t_{ij} over $[l_{ij}, u_{ij}]$ when $\boldsymbol{\pi}$ is fixed, is found at one of three possible locations: either at l_{ij} when $\frac{dt_{ij}(l_{ij},\pi_i,\pi_j)}{dv_{ij}} > 0$; at u_{ij} when $\frac{dt_{ij}(u_{ij},\pi_i,\pi_j)}{dv_{ij}} < 0$, or at $\overline{v}_{ij} = \frac{\pi_j - \pi_i - a_{ij}}{2b_{ij}}$ when $\frac{dt_{ij}(\overline{v}_{ij},\pi_i,\pi_j)}{dv_{ij}} = 0$. Note that our primal objective function is strictly convex, i.e. each $b_{ij} > 0$, so the last expression for $\overline{v}_{ij}$ is well defined.

4.2 Selecting the ascent direction

Newton's method is a commonly used gradient method for finding a stationary point of a function. Newton's method assumes that $\eta \in C^2$, and uses the second order Taylor expansion of $\eta(\boldsymbol{\pi}^\tau)$ in the direction $\boldsymbol{p}^\tau$ to determine an ascent direction:

$$\eta(\boldsymbol{\pi}^\tau + \boldsymbol{p}^\tau) - \eta(\boldsymbol{\pi}^\tau) = \bigtriangledown \eta(\boldsymbol{\pi}^\tau)^T \boldsymbol{p}^\tau + (\boldsymbol{p}^\tau)^T \bigtriangledown^2 \eta(\boldsymbol{\pi}^\tau)\boldsymbol{p}^\tau + o(||\boldsymbol{p}^\tau||). \quad (17)$$

The closer the RHS approximates the LHS in 17, assuming $o(||\boldsymbol{p}^\tau||) \approx 0$, the better ascent directions can be found. The ascent direction in Newton's' method, the Newton direction, is selected as the best such direction that can be obtained from the second order information, and constitutes the solution to the following system of linear equations:

$$\bigtriangledown^2 \eta(\boldsymbol{\pi}^\tau)\boldsymbol{p}^\tau = \bigtriangledown \eta(\boldsymbol{\pi}^\tau).$$

Although Newton's method usually has the fastest convergence rate of the gradient methods, with the Newton direction as the best ascent direction, it cannot be applied to our problem. For our dual problem $\eta \in C^1$, i.e. η is not twice differentiable, and thus the matrix $\bigtriangledown^2\eta(\boldsymbol{\pi}^\tau)$ is not available. To solve our dual problem, we must therefore use other methods that does not require second derivatives. Two such methods are the steepest ascent method and the conjugate gradient method.

4.2.1 Steepest ascent method

The steepest ascent method is an alternative to Newton's method that does not require the evaluation of second derivatives. Steepest ascent chooses the ascent direction as the gradient of the function to be maximized. The gradient points in the direction in which the function increases the fastest. In our problem $\boldsymbol{p}^\tau = \bigtriangledown\eta(\boldsymbol{\pi}^\tau) = \boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau) - \boldsymbol{d}$. The steepest ascent method has the drawback of having a slow convergence rate, but it has the advantage of always progressing as long as the gradient is nonzero.

4.2.2 Conjugate gradient method

The conjugate gradient method is another alternative to Newton's method that also uses the gradient of the function to be maximized, but neither requires the evaluation of second derivatives. The conjugate gradient method uses an updating formula to compute the ascent direction. The purpose of the updating formula is to modify the gradient so that the current ascent direction differs from the ascent directions in previous iterations. The updating formula includes the gradient of the function to be maximized, a scalar β^τ, and a previously computed ascent direction. Both the gradient and the ascent direction is taken from the iteration prior to the current one.

$$\boldsymbol{p}^{\tau+1} = \bigtriangledown\eta(\boldsymbol{v}^{\tau+1}) + \beta^\tau \boldsymbol{p}^\tau.$$

β^τ is set to zero when τ is zero or some multiple of a chosen number $m > 0$. Otherwise, there are several suggestions of how to compute β^τ. We use the Polak-Ribiere formula (shown below) in our problem, since the findings of Ventura [22] suggest that this is the superior method:

$$\beta^\tau = \bigtriangledown\eta(\boldsymbol{v}^{\tau+1})^T(\bigtriangledown\eta(\boldsymbol{v}^{\tau+1}) - \bigtriangledown\eta(\boldsymbol{v}^\tau))/||\bigtriangledown\eta(\boldsymbol{v}^\tau)||^2. \quad (18)$$

If the ascent direction in the conjugate gradient method is not set to $\bigtriangledown\eta(\boldsymbol{v}^\tau)$ every m:th iteration, but exclusively given by formula 18, the resulting search directions actually approaches $\bigtriangledown^2\eta(\boldsymbol{v}^\tau)$ as the iterations proceed. The conjugate gradient method usually have a faster convergence rate than the

steepest ascent method, since the former avoids to search in an already visited direction, whilst the latter tends to perform searches in the same direction several times.

4.3 Selecting the step size

4.3.1 Inexact line search

The Armijo step size rule performs an approximate line search. For our problem, this means to find an $\alpha^\tau = \beta^m s$ for scalars $\sigma \in (0, 1)$, $\beta \in (0, 1)$, and $s > 0$ that satisfies the inequality below for the first nonnegative integer m:

$$\eta(\boldsymbol{v}^\tau + \beta^m s \boldsymbol{p}^\tau) - \eta(\boldsymbol{v}^\tau) \geq \sigma \beta^m s \bigtriangledown \eta(\boldsymbol{v}^\tau)^T \boldsymbol{p}^\tau.$$

According to Bertsekas [4], σ is recommended to be within the interval [0.00001, 0.1], β within the interval [0.1, 0.5], and s is suggested to be 1.

4.3.2 Exact line search

The Helgason step size rule suggested by Helgason et al. [11], gives a method to perform an exact line search. By solving the line search subproblem below, an optimal α^τ is obtained.

$$\underset{\alpha}{\text{maximize}} \;\; \eta(\boldsymbol{\pi}^\tau + \alpha \boldsymbol{p}^\tau) \tag{19}$$

$$\text{subject to } \alpha \geq 0 \tag{20}$$

We first apply the Helgason rule to our problem in matrix form. The expression of η then becomes:

$$\eta(\boldsymbol{\pi}^\tau + \alpha \boldsymbol{p}^\tau) = \underset{\boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}}{\text{minimum}} \;\; \boldsymbol{v}^T \boldsymbol{B} \boldsymbol{v} + \boldsymbol{a}^T \boldsymbol{v} + (\boldsymbol{\pi}^\tau + \alpha \boldsymbol{p}^\tau)^T (\boldsymbol{E} \boldsymbol{v} - \boldsymbol{d}).$$

Since η is concave and differentiable, its optimal solution $\alpha = \alpha^\tau$ for a solution $\boldsymbol{v}^\tau$ to the minimization problem above, is found when:

$$\bigtriangledown \eta(\boldsymbol{\pi}^\tau + \alpha^\tau \boldsymbol{p}^\tau)^T \boldsymbol{p}^\tau = (\boldsymbol{E} \boldsymbol{v}^\tau - \boldsymbol{d})^T \boldsymbol{p}^\tau = 0. \tag{21}$$

Using (21), the line search subproblem can be reformulated as follows:

$$\underset{\boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}}{\text{minimize}} \;\; \boldsymbol{v}^T \boldsymbol{B} \boldsymbol{v} + \boldsymbol{a}^T \boldsymbol{v} + \boldsymbol{\pi}^{\tau T} (\boldsymbol{E} \boldsymbol{v} - \boldsymbol{d}) \tag{22}$$

$$\text{subject to } \boldsymbol{p}^{\tau T} (\boldsymbol{E} \boldsymbol{v} - \boldsymbol{d}) = 0, \tag{23}$$

with α^τ being the optimal Lagrange multiplier of constraint (23). Since our problem is separable, we can also apply the Helgason rule component-wise. The line search subproblem is then formulated:

$$\underset{\boldsymbol{l}\leq \boldsymbol{v}\leq \boldsymbol{u}}{\text{minimize}} \quad \sum_{(i,j)\in A} b_{ij}{v_{ij}}^2 + \sum_{(i,j)\in A} (a_{ij} + {\pi_i}^\tau - {\pi_j}^\tau)v_{ij} - \sum_{i\in V} {\pi_i}^\tau d_i \tag{24}$$

$$\text{subject to} \quad \sum_{(i,j)\in A} ({p_i}^\tau - {p_j}^\tau)v_{ij} = \sum_{i\in V} {p_i}^\tau d_i \tag{25}$$

The solution to the problem (24)-(25) is found by considering its optimality conditions, and finding the value of the Lagrange multiplier to the equality constraint. From the KKT conditions, the solution as a function of α, $v_{ij}(\alpha)$, is, for each arc, given by the following expression (Ventura [22], Theorem 4):

$$v_{ij}(\alpha) = \text{ MID}\left\{l_{ij}, -\frac{(a_{ij} + {\pi_i}^\tau - {\pi_j}^\tau) + ({p_i}^\tau - {p_j}^\tau)\alpha}{2b_{ij}}, u_{ij}\right\}, \forall (i,j) \in A,$$

where MID is a function that selects the middle value of its three arguments. Each $v_{ij}(\alpha)$ is a piece-wise linear and continuous nonincreasing function with two breakpoints at $\alpha = \frac{-(2b_{ij}u_{ij}+a_{ij}+\pi_i-\pi_j)}{p_i-p_j}$ and $\alpha = \frac{-(2b_{ij}l_{ij}+a_{ij}+\pi_i-\pi_j)}{p_i-p_j}$. Let $h(\alpha)$ denote the function in the LHS of constraint (25):

$$\begin{aligned} h(\alpha) &= \sum_{(i,j)\in A} ({p_i}^\tau - {p_j}^\tau)v_{ij}(\alpha) \\ &= \sum_{(i,j)\in A} ({p_i}^\tau - {p_j}^\tau)\ \text{MID}\left\{l_{ij}, -\frac{(a_{ij} + {\pi_i}^\tau - {\pi_j}^\tau) + ({p_i}^\tau - {p_j}^\tau)\alpha}{2b_{ij}}, u_{ij}\right\} \end{aligned}$$

$h(\alpha)$ is also a piece-wise linear and continuous nonincreasing function (Ventura [22], Theorem 5). By considering the at most $2|A|$ breakpoints of the function $h(\alpha)$, the optimal α^τ such that $h(\alpha^\tau) = \sum_{i\in V} {p_i}^\tau d_i$, is found when applying the bisection method to the sorted breakpoints in increasing order. Either α^τ is found directly among the sorted breakpoints or defined by a linear interpolation between two consecutive breakpoints. We have chosen to sort the breakpoints using Shellsort. The worst case running time of Shellsort is quadratic in the size of the input, but it has the advantage of having a close to linear complexity if applied to a sequence of nearly sorted elements (Sedgewick [20]). As the dual solution approaches optimality, there is a good chance that the ordered sequence of breakpoints of two consecutive iterations are identical. We have therefore chosen to start Shellsort with the unsorted breakpoints from a current iteration in the same order as the

sorted breakpoints from the previous iteration. Thus, we can conclude that the running time of Shellsort for our problem will become closer to linear than quadratic as the dual solution approaches its optimal value.

4.4 Coordinate ascent method

The coordinate ascent method maximizes the objective function along one coordinate direction at each iteration. The coordinate ascent method consists of selecting a coordinate direction i, and a step size α in that direction such that the objective function eventually is maximized. The way the coordinate directions are chosen varies. We will describe the approach that is used in the Curet heuristic [7]. The essence of the coordinate ascent algorithm, as described by Curet, is to select a vertex i with a violated flow conservation constraint to determine the coordinate direction $\boldsymbol{e}_i$, where $\boldsymbol{e}_i$ denotes the i:th coordinate vector in $\Re^{|A|}$. The direction $\boldsymbol{p}$ is selected as $\pm\boldsymbol{e}_i$ such that $\eta'(\boldsymbol{\pi};\boldsymbol{p}) > 0$, and the step size α^τ is chosen such that $\eta(\boldsymbol{\pi}^\tau + \alpha^\tau \boldsymbol{p}^\tau) > \eta(\boldsymbol{\pi}^\tau)$. The generic algorithm for the coordinate ascent method is shown below:

Coordinate ascent method$(\eta, \boldsymbol{\pi}^0)$
1. determine $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ by solving the Lagrangian dual subproblem
2. **if** $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ is feasible in the primal problem
3. **goto** 11
4. **else**
5. select a vertex i with a violated flow conservation constraint
6. $\boldsymbol{p}^\tau = \pm\boldsymbol{e}_i$ such that $\eta'(\boldsymbol{\pi}^\tau;\boldsymbol{p}^\tau) > 0$
7. $\alpha^\tau > 0$ such that $\eta(\boldsymbol{\pi}^\tau + \alpha^\tau \boldsymbol{p}^\tau) > \eta(\boldsymbol{\pi}^\tau)$
8. $\boldsymbol{\pi}^{\tau+1} = \boldsymbol{\pi}^\tau + \alpha^\tau \boldsymbol{p}^\tau$
9. $\tau = \tau + 1$
10. **goto** 1
11. **return** $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$

In general the coordinate ascent method has a convergence rate similar to the steepest ascent method. A convergence guarantee for the coordinate ascent method is present if the objective function to be maximized is strictly convex, and differentiable. The convexity assumption is needed as the method performs a search for a unique maximum along each coordinate; the differentiability assumption is needed as the method is known to get stuck at nondifferentiable points. If not the prerequisites for the objective function are met, the coordinate ascent method is known to cycle without approaching any stationary point. The coordinate ascent method have the advantage of being suitable for parallel computing if the objective function is separable. The method can then be applied independently to each

coordinate subset, where a coordinate subset consists of those coordinates that are not coupled through the objective function.

5 Lagrangian heuristics

So, why not be satisfied with the solution to the Lagrangian dual problem as produced by some dual gradient method? Our problem is strictly convex and quadratic; the primal and dual problem are feasible. Thus, we are guaranteed by the theorems presented i Section 3.1 that the dual optimal solution and the primal optimal solution exists and have the same objective value. Moreover, we have an explicit method to obtain the primal solution from the dual solution. The dilemma is that the dual solution is not primal feasible until we reach optimality, and finding the optimal dual solution can take forever if we are unlucky. Fortunately, we can still make use of the non-optimal dual solutions.

The purpose of a Lagrangian heuristic is to find near-optimal primal solutions by solving the Lagrangian dual problem and manipulate the solutions obtained to make them primal feasible. Even if the solutions produced by a Lagrangian heuristic are non-optimal in the primal problem, they provide a (primal) upper bound, and as we have the lower bound from the dual solution, we can estimate how far from optimum we actually are.

In the following sections, seven Lagrangian heuristics that are used in combination with a gradient method that is guaranteed to converge to the optimal solution will be presented. Three of them have been presented in the M.Sc. thesis by Marklund [15], and in the textbook by Patriksson [19]. One is presented according to the findings of Curet [7]. FlowrouteD, FlowrouteBS and Shortest PathL are novel approaches that have not been presented in the literature before. Note that all function names used in the pseudocode for this section are either commonly used names for functions, operations on ordinary data structures, or names used in the computer code presented in Appendix A. The graph search techniques in Sections 5.3–5.6 are presented according to the descriptions in the textbooks by Ahuja [1], Biggs [5], Cormen [6], and Migdalas&Göthe–Lundgren [16].

5.1 A projection-like property

That the dual solution will not be primal feasible until the optimal solution to the dual problem is found has already been stated. Let $\boldsymbol{\pi}^\tau \in \Re^{|V|}$ denote the dual solution at iteration τ, and let $\boldsymbol{v}(\boldsymbol{\pi}^\tau) \in \Re^{|A|}$ denote the corresponding solution to the Lagrangian dual subproblem (9) for $\boldsymbol{\pi}^\tau$. We want to find a primal feasible point by projecting $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ onto the primal feasible set F. A good candidate to perform such an operation onto F is the Euclidian projection, $Proj$; the vector-valued mapping that produces a primal feasible point of minimum distance from $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. Unfortunately, finding the

Euclidian projected point is nearly as hard as solving our original problem. Therefore, a "fake projection", P, is required. P should be computationally cheaper than the Euclidian projection, but with similar characteristics. A point projected by P must not be worse than a point projected by $Proj$ in the sense that if the distance between $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ and the projected point given by $Proj$ becomes very small, so must the distance between $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ and the projected point given by P. Thus, to make sure that P converges towards the optimal solution, the following properties must be established. Firstly, P must have a projection-like property. Secondly, P must produce a primal feasible point that approaches $\boldsymbol{v}^*$ as $\boldsymbol{\pi}^\tau$ approaches $\boldsymbol{\pi}^*$.

Definition 5 (Patriksson [19], Definition 10.2.1.)
We say that for a sequence $\{\boldsymbol{v}(\boldsymbol{\pi}^\tau)\} \subset \Re^{|A|}$, *the vector-valued mapping* $P : \Re^{|A|} \to F$ *is projection-like onto* F, *if* P *has the property that*

$$Proj(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|} \Longrightarrow P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}. \tag{26}$$

If a Lagrangian heuristic with the projection-like property is used, Theorem 6 shows that the primal feasible points, $P(\boldsymbol{v}(\boldsymbol{\pi}^\tau))$, produced by the heuristic will approach $\boldsymbol{v}^*$ in the limit.

Theorem 6 (Liu [14], Theorem 5.) *Consider the dual sequence* $\{\boldsymbol{\pi}^\tau\}$ *with the property that* $\boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{v}^*$ *as* $\tau \to \infty$, *and the primal sequence* $\{\boldsymbol{v}^\tau\} = \{P(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ *generated by a Lagrangian heuristic with a projection-like property. Then* $\boldsymbol{v}^\tau \to \boldsymbol{v}^*$.

Proof We have that

$$\begin{aligned} ||\boldsymbol{v}^\tau - \boldsymbol{v}^*|| &= ||P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}^*|| \\ &= ||P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) + \boldsymbol{v}(\boldsymbol{\pi}^\tau) - \boldsymbol{v}^*|| \\ &\leq ||P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)|| + ||\boldsymbol{v}(\boldsymbol{\pi}^\tau) - \boldsymbol{v}^*||. \end{aligned}$$

Since $\boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{v}^*$ by assumption, we must have that $Proj(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$ as $\tau \to \infty$. By the projection-like property of P, also $P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, and hence, $\boldsymbol{v}^\tau \to \boldsymbol{v}^*$. ∎

A dual gradient method produces a sequence,$\{\boldsymbol{v}(\boldsymbol{\pi}^\tau)\}$, that converges to $\boldsymbol{v}^*$ in the limit. Since the Lagrangian heuristics we present in the following sections are used in combination with a dual gradient method, the condition $\boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{v}^*$ is satisfied automatically. Hence, Theorem 6 holds for each Lagrangian heuristic if the relation $P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) \to \boldsymbol{v}(\boldsymbol{\pi}^\tau)$ is proven. Therefore, that $P(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) \to \boldsymbol{v}(\boldsymbol{\pi}^\tau)$ holds for each Lagrangian heuristic will be established along with its presentation.

5.2 Minimum Deviation

The Minimum Deviation heuristic was first presented by Marklund [15]. Minimum Deviation aims to minimize the total amount of flow that needs to be rerouted in the network from the flow produced by the dual solution in order to accomplish primal feasibility. By associating a deviation variable, $\boldsymbol{\delta}_{ij}$, with each arc (i, j) of the network, the original problem is converted to a linear programming problem that can be solved by a regular linear minimum cost network flow solver. The purpose of the deviation variable is to find the minimum amount of flow that needs to be adjusted from the initial infeasible flow given by the dual solution in order to satisfy the constraints of the primal problem. Hence, for $\boldsymbol{v}(\boldsymbol{\pi}^{\tau})$, the problem is formulated as the following linear program:

$$\underset{(\boldsymbol{\delta}^{+},\boldsymbol{\delta}^{-})}{\text{minimize}} \ (\boldsymbol{\delta}^{+} + \boldsymbol{\delta}^{-})^{T}(\mathbf{1}^{|A|}), \tag{27}$$

$$\text{subject to } \boldsymbol{E}(\boldsymbol{\delta}^{+} - \boldsymbol{\delta}^{-}) = \boldsymbol{d} - \boldsymbol{E}(\boldsymbol{v}(\boldsymbol{\pi}^{\tau})), \tag{28}$$

$$\boldsymbol{\delta}^{+} \leq \boldsymbol{u} - \boldsymbol{v}(\boldsymbol{\pi}^{\boldsymbol{\tau}}), \tag{29}$$

$$\boldsymbol{\delta}^{-} \leq -\boldsymbol{l} + \boldsymbol{v}(\boldsymbol{\pi}^{\boldsymbol{\tau}}), \tag{30}$$

$$\boldsymbol{\delta}^{+}, \boldsymbol{\delta}^{-} \geq \boldsymbol{0}^{|A|}. \tag{31}$$

Minimum Deviation introduces a deviation variable, $\boldsymbol{\delta}_{ij} = \boldsymbol{\delta}_{ij}{}^{+} - \boldsymbol{\delta}_{ij}{}^{-}$, that denotes the deviation for each arc of the network. $\boldsymbol{\delta}_{ij}{}^{+}$ denotes the addition of flow needed for adjustment on the current arc, and $\boldsymbol{\delta}_{ij}{}^{-}$ denotes the subtraction of flow needed for adjustment on the current arc. Using the original structure of the network, an arc in the same direction as the original arc models the variable $\boldsymbol{\delta}_{ij}{}^{+}$, and a new arc in the opposite direction as the original arc models the variable $\boldsymbol{\delta}_{ij}{}^{-}$. The pseudocode for Minimum Deviation is presented below, and as suggested on line 9, the problem for the altered network is solved by a linear minimum cost network flow solver.

Minimum Deviation heuristic(N)
1. **for** all arcs $a \in A$
2. associate the decision variable $\boldsymbol{\delta}^{+}$ with a
3. set the upper limit of a to (Upper(a) - Flow(a))
4. make a copy a_{rev} of a in the opposite direction
5. associate the decision variable $\boldsymbol{\delta}^{-}$ with a_{rev}
6. set the upper limit of a_{rev} to (-Lower(a) + Flow(a))
7. **for** all vertices $v \in V$
8. set the demand of v to (Demand(v) - Balance(v))
9. $\boldsymbol{\delta}^{*}$ = MCFSolver(N, $\boldsymbol{\delta}^{+}$, $\boldsymbol{\delta}^{-}$)
10. **return**($\boldsymbol{v}(\boldsymbol{\pi}^{\tau}) + \boldsymbol{\delta}^{*}$)

Minimum Deviation satisfies property (26), and therefore the sufficient conditions in Theorem 6.

Theorem 7 *Consider the Lagrangian heuristic Minimum Deviation, P^{MD}, and the by P^{MD} generated primal sequence $\{\boldsymbol{v}^\tau\} = \{P^{MD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ starting at $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. If P^{MD} has the property that $P^{MD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, then $\{\boldsymbol{v}^\tau\} \to \boldsymbol{v}^*$.*

Proof The expression $||P^{MD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)||$ is closely related to the expression $\sum_{(i,j)\in A} |P^{MD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))_{i,j} - \boldsymbol{v}(\boldsymbol{\pi}^\tau)_{i,j}|$. If one of the expressions is equal to zero, the other expression must also be equal to zero. Moreover, the vector $\boldsymbol{\delta}^* = (\boldsymbol{\delta}^+)^* - (\boldsymbol{\delta}^-)^*$ is the solution to the problem:

$$\underset{(\boldsymbol{\delta}^+,\boldsymbol{\delta}^-)}{\text{minimize}} \left\{ \sum_{(i,j)\in A} |\boldsymbol{\delta}_{ij}| \;\middle|\; \begin{array}{c} \boldsymbol{E}(\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \boldsymbol{\delta}) = \boldsymbol{d}, \\ (\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \boldsymbol{\delta}) \in X \end{array} \right\}.$$

The maximum amount of flow that is rerouted by Minimum Deviation can be no more than $\sum_{(i,j)\in A} |\boldsymbol{\delta}_{ij}|$. Therefore, this value constitutes an upper bound for the maximum amount of flow that can be adjusted on any arc. Since $\boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{v}^*$ as $\tau \to \infty$, it follows that $\sum_{(i,j)\in A} |\boldsymbol{\delta}_{ij}| \to \boldsymbol{0}^{|A|}$. Then $\sum_{(i,j)\in A} |P^{MD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))_{i,j} - \boldsymbol{v}(\boldsymbol{\pi}^\tau)_{i,j}| = \sum_{(i,j)\in A} |\boldsymbol{\delta}_{ij}{}^*| \leq \sum_{(i,j)\in A} |\boldsymbol{\delta}_{ij}| \to \boldsymbol{0}^{|A|}$. Hence, we can conclude that $||P^{MD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)|| \to \boldsymbol{0}^{|A|}$. ∎

5.3 Flowroute

The Flowroute heuristic was first presented by Marklund [15]. Flowroute aims to accomplish primal feasibility by rerouting flow in a residual graph constructed from the dual solution. As the dual solution does not respect the flow conservation constraints, situations where these constraints are violated will occur. When more flow is entering a vertex than leaving it, the vertex will become a source in the residual graph. When more flow is leaving a vertex than entering it, the vertex will become a sink in the residual graph. Using the breadth first search technique (BFS) augmenting paths with the minimum number of arcs from the residual sources to the residual sinks are found. While respecting the limits of the involved arcs along the paths, the flow is transported from the residual sources to the residual sinks. When the supply at the residual sources satisfies the demand at the residual sinks we have obtained a primal feasible solution, since rerouting flow in the residual graph also means rerouting flow in the original network. The pseudocode for Flowroute is presented below. The algorithm for BFS is found in the textbook by Cormen [6], and has a running time bounded by $O(|V| + |A|)$ when applied from a single vertex.

```
Flowroute heuristic(N)
1. for all sources s ∈ V
2.     BFS(N, s)
3.         for all sinks t ∈ V connected to s
4.             while Supply(s) > 0
5.                 find an augmenting path from s to t
6.                 augment as much flow as possible along the path
7. return
```

Flowroute satisfies property (26), and therefore the sufficient conditions in Theorem 6.

Theorem 8 *Consider the Lagrangian heuristic Flowroute, P^F, and the by P^F generated primal sequence $\{\boldsymbol{v}^\tau\} = \{P^F(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ starting at $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. If P^F has the property that $P^F(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, then $\{\boldsymbol{v}^\tau\} \to \boldsymbol{v}^*$.*

Proof The expression $||P^F(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)||$ is closely related to the expression $\sum_{(i,j)\in A} |P^F(\boldsymbol{v}(\boldsymbol{\pi}^\tau))_{i,j} - \boldsymbol{v}(\boldsymbol{\pi}^\tau)_{i,j}|$. If one of the expressions is equal to zero, the other expression must also be equal to zero. Let $g_i(\boldsymbol{\pi}^\tau) = -\frac{\partial \eta(\boldsymbol{\pi}^\tau)}{\partial \boldsymbol{\pi}^\tau{}_i}$ denote the flow imbalance at vertex i. The maximum amount of flow that is rerouted by Flowroute can be no more than $\sum_{i\in V} |g_i(\boldsymbol{\pi})|$. Hence, this value constitutes an upper bound for the maximum amount of flow that can be adjusted on any arc. Since $\boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{v}^*$ as $\tau \to \infty$, it follows that $g_i(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$. Then $||P^F(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)|| \leq |A| \sum_{i\in V} |g_i(\boldsymbol{\pi})| \to \boldsymbol{0}^{|A|}$. Hence, we can conclude that $||P^F(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)|| \to \boldsymbol{0}^{|A|}$. ∎

5.4 FlowrouteD

The FlowrouteD heuristic is a modification of the Flowroute heuristic. As in Flowroute, primal feasibility is accomplished by rerouting flow in a residual graph constructed from the dual solution. However, in FlowrouteD the depth first search technique (DFS) is used to find augmenting paths from the residual sources to the residual sinks. The running time for DFS is $\Theta(|V| + |A|)$ (Cormen [6]), but by using the information provided by DFS instead of BFS it is possible to find residual sinks faster. The DFS algorithm favors deep progress in the network, and can be truncated as soon as any sink has been found, usually before all of the vertices in the network have been visited. The BFS, on the other hand, is normally truncated at a later stage, since the algorithm favors broad progress, and usually visits more vertices in the network before any sink can be found. So, with DFS replacing BFS in step 2 of the pseudocode for Flowroute presented above, the pseudocode for FlowrouteD is identical to the pseudocode for Flowroute.

FlowrouteD satisfies property (26), and therefore the sufficient conditions in Theorem 6.

Theorem 9 *Consider the Lagrangian heuristic FlowrouteD, P^{FD}, and the by P^{FD} generated primal sequence $\{\boldsymbol{v}^\tau\} = \{P^{FD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ starting at $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. If P^{FD} has the property that $P^{FD}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, then $\{\boldsymbol{v}^\tau\} \to \boldsymbol{v}^*$.*

Proof Apply the proof of Theorem 8 with P^{FD} replacing P^F. ∎

5.5 FlowrouteBS

The FlowrouteBS heuristic is another modification of the Flowroute heuristic. The graph search technique used in FlowrouteBS is the multiple breadth search technique (BS). At termination, the BS algorithm produces an acyclic layered digraph (ALD) which is used in the polynomially bounded max flow algorithm (MFALD) presented by Migdalas&Göthe–Lundgren [16] to reroute flow along many paths simultaneously in the residual graph constructed from the dual solution.

The BS algorithm is a modification of the BFS algorithm. The modification lies in how the information about reaching a vertex is stored. In the BFS algorithm a scalar with the index of the vertex that proceeds the current vertex in the graph search is kept. In the BS algorithm this is changed to a set of vertices (predecessors) that give the same path length to the current vertex. The running time of BS is $O(|V| + |A|)$ (Migdalas&Göthe–Lundgren [16]).

The information provided by keeping the set of predecessors at each vertex makes it possible to transport the flow along several paths simultaneously. The MFALD is a modification of the maximum flow algorithm by Ford–Fulkerson (Cormen [6]) that uses this possibility. The MFALD is based on the BS search technique, and uses a push and pull principle to update the flows of the ALD as long as there exist paths from the source to the sink, or until the source has a supply equal to zero. First, each vertex $v \in V$ is assigned a capacity:

$$\text{VertexCap}(v) = \text{MIN}\left\{\sum \Delta u_{ij}, \sum \Delta u_{ji}\right\}, \qquad \forall i \in V \backslash \{s,t\},$$

where Δu_{ij} denotes the residual capacity of an arc, and MIN is a function that selects the minimum of its two arguments. The value of Δu_{ij} is defined as the upper capacity of the arc minus the current flow of the arc. The capacity of the source is calculated as the sum of the residual capacities of the outgoing arcs from the source, and the capacity of the sink is

calculated as the sum of the residual capacities of the incoming arcs to the sink. The vertex with the minimum capacity is then chosen as the start vertex. In the push procedure, a flow equal to the minimum capacity value is transported from the start vertex along increasing layers towards the sink. In the pull procedure, the flow is transported from the start vertex along decreasing layers towards the source. If there are more paths from the source to the sink, the next push and pull procedure is performed in the residual graph of the current digraph. The residual graph has the vertices with a capacity equal to zero removed along with the incoming and outgoing arcs connected to them. The arcs with a flow equal to the arcs' upper capacity are also removed. Each vertex is assigned a new vertex capacity based on the appearance of the residual graph, and the procedure repeats until the sink is unreachable from the source, or the source has a supply equal to zero. The running time for one iteration of the MFALD algorithm is $O(|V|^2 + |A|)$ (Migdalas&Göthe–Lundgren [16]). The pseudocode for FlowrouteBS is presented below. The algorithms for BS and MFALD are found in the textbook by Migdalas&Göthe–Lundgren [16]; together they have a running time bounded by $O(|V|^3)$.

FlowrouteBS heuristic(N)
1. **for** all sources $s \in V$
2. link s to a supersource s'
3. **for** all sinks $t \in V$
4. link t to a supersink t'
5. **while** Supply(s') > 0
6. ald = BS(N, s')
7. MFALD(ald, s', t')
8. **return**

FlowrouteBS satisfies property (26), and therefore the sufficient conditions in Theorem 6.

Theorem 10 *Consider the Lagrangian heuristic FlowrouteBS, P^{FBS}, and the by P^{FBS} generated primal sequence $\{\boldsymbol{v}^\tau\} = \{P^{FBS}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ starting at $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. If P^{FBS} has the property that $P^{FBS}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, then $\{\boldsymbol{v}^\tau\} \to \boldsymbol{v}^*$.*

Proof The expression $||P^{FBS}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)||$ is closely related to the expression $\sum_{(i,j)\in A} |P^{FBS}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))_{i,j} - \boldsymbol{v}(\boldsymbol{\pi}^\tau)_{i,j}|$. If one of the expressions is equal to zero, the other expression must also be equal to zero. Let $\Delta u_{ij} = u_{ij} - \boldsymbol{v}(\boldsymbol{\pi}^\tau)_{ij}$ denote the remaining capacity of arc (ij), and let $c_i(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) = min\{\sum \Delta u_{ij}, \sum \Delta u_{ji}\}$ denote the capacity at vertex i. For the super-source

s', $u_{s's} = \sum_{s \in V} \text{Supply}(s)$, and $c_{s'}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) = \min\{\sum \Delta u_{s's}, 0\}$. For the super-sink t', $u_{tt'} = \sum_{t \in V} \text{Demand}(t)$, and $c_{t'}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) = \min\{0, \sum \Delta u_{tt'}\}$. The maximum amount of flow that is rerouted by FlowrouteBS can be no more than $c(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) = \sum_{n=0}^{k} \min_{i \in V}\{c_i(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$, where k denotes the maximum number of flow augmenting sweeps in the MFALD algorithm. Hence, this value constitutes an upper bound for the maximum amount of flow that can be adjusted on any arc. Since $\{\boldsymbol{v}(\pi^\tau)\} \rightarrow \boldsymbol{v}^*$ as $\tau \rightarrow \infty$, it follows that $c(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) \rightarrow \boldsymbol{0}^{|A|}$. Then $||P^{FBS}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)|| \leq |A| \sum_{i \in V} c(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) \rightarrow \boldsymbol{0}^{|A|}$. Hence, we can conclude that $||P^{FBS}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau)|| \rightarrow \boldsymbol{0}^{|A|}$. ∎

5.6 Shortest Path

The Shortest Path heuristic was first presented by Marklund [15]. As in the Flowroute heuristic, Shortest Path aims to accomplish primal feasibility by rerouting flow in a residual graph constructed from the dual solution. However, in Shortest Path the rerouting of flow is done along the cheapest paths in the residual graph. The cost to traverse an arc is defined as the derivative of the arcs' quadratic cost function. The graph search algorithm used in Shortest Path is the Bellman–Ford algorithm. The Bellman–Ford algorithm finds the shortest (cheapest) paths from a start-vertex to all other vertices in a graph with possibly negative arc costs, provided that the graph does not contain any negative cycles. A negative cycle means that the cost of a path can be reduced infinitely. The residual graph, as constructed in Section 5.3, will include arcs with negative costs. Therefore, to find feasible solutions by using Bellman–Ford, we must show that the residual network does not contain any negative cycles, and this is done by contradiction in the theorem below.

Theorem 11 *The residual graph constructed from the dual solution, $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$, does not contain any negative cycles, C.*

Proof By Everett's theorem (Andréasson et al. [2], Theorem 6.31), $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ is also an optimal solution to the following problem:

$$\begin{aligned} &\underset{\boldsymbol{v}}{\text{minimize}}\ \Phi(\boldsymbol{v}), \\ &\text{subject to } \boldsymbol{E}\boldsymbol{v} = \boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau), \\ &\qquad\qquad \boldsymbol{l} \leq \boldsymbol{v} \leq \boldsymbol{u}. \end{aligned} \tag{32}$$

If the residual graph contains a negative cycle for the fixed flow $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$, $\sum_{(ij) \in C} a_{ij} < 0$, where a_{ij} denotes the linearized cost of arc (i, j). Let p_{ij}

define an addition of flow on arc (i,j):

$$p_{ij} = \begin{cases} 1, \text{ if } (i,j) \in C, \text{ and } (i,j) \text{ is used in forward direction} \\ -1, \text{ if } (i,j) \in C, \text{ and } (i,j) \text{ is used in the backward direction}, \\ 0 \text{ otherwise.} \end{cases}$$

By the definition of $\boldsymbol{p}$, any cycle satisfies $\boldsymbol{E}\boldsymbol{p} = \boldsymbol{0}^{|A|}$, but a negative cycle also fulfils: $\sum_{(ij)\in C} a_{ij} < 0 \Leftrightarrow \bigtriangledown\Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau))^T\boldsymbol{p} < \boldsymbol{0}^{|A|}$.

If a negative cycle exists, the circulating flow must satisfy $\boldsymbol{l} \leq \boldsymbol{v} + \alpha\boldsymbol{p} \leq \boldsymbol{u}$ for some small scalar $\alpha > 0$. Moreover, the point $(\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \alpha\boldsymbol{p})$ is feasible in the problem (32), since $\boldsymbol{E}(\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \alpha\boldsymbol{p}) = \boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \alpha\boldsymbol{E}\boldsymbol{p} = \boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \alpha\boldsymbol{0}^{|A|} = \boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. Let us therefore study the Taylor expansion of Φ around $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ in the direction $\boldsymbol{p}$:

$$\Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \alpha\boldsymbol{p}) = \Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) + \alpha \bigtriangledown \Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau))^T\boldsymbol{p} + O(\alpha^2)$$

By using $\bigtriangledown\Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau))^T\boldsymbol{p} < \boldsymbol{0}^{|A|}$, and assuming that $O(\alpha^2) \approx 0$, we see that $\Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \alpha\boldsymbol{p}) < \Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. This contradicts the optimality of $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$ by Everett's theorem. Hence, we can conclude that the residual graph cannot contain any negative cycles. ∎

With Bellman–Ford replacing BFS in step 2 of the pseudocode for Flowroute presented in section 5.3, the pseudocode for Shortest Path is identical to the pseudocode for Flowroute. The algorithm for Bellman–Ford is found in Cormen [6], and has a running time bounded by $O(|V||A|)$.

Shortest Path satisfies property (26), and therefore the sufficient conditions in Theorem 6.

Theorem 12 *Consider the Lagrangian heuristic Shortest Path, P^{SP}, and the by P^{SP} generated primal sequence $\{\boldsymbol{v}^\tau\} = \{P^{SP}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ starting at $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. If P^{SP} has the property that $P^{SP}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, then $\{\boldsymbol{v}^\tau\} \to \boldsymbol{v}^*$.*

Proof Apply the proof of Theorem 8 with P^{SP} replacing P^F. ∎

5.7 Shortest PathL

The Shortest PathL is a modification of the Shortest Path heuristic. As in Shortest Path, primal feasibility is accomplished by rerouting flow in a residual graph constructed from the dual solution. However, in Shortest PathL the cost to traverse an arc is defined as the derivative of the arcs' Lagrangian function. This is motivated by considering the results in Larsson&Patriksson [13]. The relaxed primal–dual optimality conditions are for

our problem formulated:

$$\Phi(\boldsymbol{v}) + \boldsymbol{\pi}^T(\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d}) \leq \eta(\boldsymbol{\pi}) + \epsilon, \tag{33}$$

$$\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d} = \boldsymbol{0}^{|A|}, \tag{34}$$

$$\epsilon \leq \kappa, \tag{35}$$

$$\epsilon, \kappa \geq 0. \tag{36}$$

From (33) we see that $\boldsymbol{v}$ is ϵ-optimal in the Lagrangian dual subproblem for a given $\boldsymbol{\pi}$, and from (34) that $\boldsymbol{v}$ satisfies $\Phi(\boldsymbol{v}) \leq \eta(\boldsymbol{\pi}) + \epsilon$. Therefore, adjusting a primal infeasible $\boldsymbol{v}(\boldsymbol{\pi})$ towards feasibility, means adjusting the value of the Lagrangian function also. Assuming that $\boldsymbol{\pi}$ is non-optimal in the dual problem, but that $\eta(\boldsymbol{\pi})$ is at most $\beta \geq 0$ from the optimal value Φ^*, it follows that $\boldsymbol{v} \in X^{\kappa-\beta}$ (Larsson&Patriksson [13], Corollary 12). Here, $X^{\kappa-\beta}$ denotes the set of vectors that is feasible in the primal problem, and deviates in objective value at most $\kappa - \beta$ from the optimal one. Note, if $\beta = \kappa$, then $\boldsymbol{v}$ is optimal in the primal problem. Note also, if $\beta = \kappa = 0$, then $\boldsymbol{\pi}$ is optimal in the dual problem since by (35) it follows that $\epsilon = 0$, and $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{\pi})$ is optimal in the primal problem. Hence, if β is very small and the system (33)–(36) is consistent, $\boldsymbol{v}$ is a very good approximation of the primal optimal solution, and it follows that κ also is very small. A Lagrangian heuristic that wants to obtain primal feasibility should therefore aim to adjust the Lagrangian function as little as possible. Therefore, minimizing the Lagrangian function, as done in Shortest PathL, is motivated. Note, however, that when $\boldsymbol{\pi} \to \boldsymbol{\pi}^*$, the term $\boldsymbol{\pi}^T(\boldsymbol{E}\boldsymbol{v} - \boldsymbol{d}) \to \boldsymbol{0}^{|A|}$, and the value of the Lagrangian function is almost the same as the value of the primal cost function.

As in the Shortest Path heuristic, the graph search algorithm used in Shortest PathL is the Bellman–Ford algorithm. Hence, with the derivative of the arcs' quadratic cost function replaced with the derivative of the arcs' Lagrangian function, the pseudocode for Shortest PathL is identical to the pseudocode for Shortest Path.

Shortest PathL satisfies property (26), and therefore the sufficient conditions in Theorem 6.

Theorem 13 *Consider the Lagrangian heuristic Shortest PathL, P^{SPL}, and the by P^{SPL} generated primal sequence $\{\boldsymbol{v}^\tau\} = \{P^{SPL}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))\}$ starting at $\boldsymbol{v}(\boldsymbol{\pi}^\tau)$. If P^{SPL} has the property that $P^{SPL}(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{v}(\boldsymbol{\pi}^\tau) \to \boldsymbol{0}^{|A|}$, then $\{\boldsymbol{v}^\tau\} \to \boldsymbol{v}^*$.*

Proof Apply the proof of Theorem 8 with P^{SPL} replacing P^F. ∎

5.8 Curet

The Curet heuristic was first presented by Curet [7], but a modified version that is used in combination with the steepest ascent method appears in the thesis by Marklund [15] and in the textbook by Patriksson [19]. The essence of the Curet heuristic is to truncate the coordinate ascent method prior to the optimal dual solution, and yet produce a near-optimal primal solution. The Curet heuristic follows the coordinate ascent method as described in section 4.4, but uses a perturbed directional derivative, $\boldsymbol{D}_\epsilon(\boldsymbol{\pi}^\tau, \boldsymbol{p}^\tau)$, by introducing a small number $\epsilon > 0$ in the real directional derivative when approaching the solution to the Lagrangian dual problem. The purpose of the ϵ factor is twofold. Firstly, it determines bounds within which the current primal infeasible flow can be adjusted to become primal feasible. Secondly, it determines the perturbation of the gradient in the Lagrangian dual problem. If a primal feasible flow is found within the given bounds, the heuristic terminates. Otherwise, the coordinate ascent method continues, and a new dual point is determined by taking a step α^τ in a direction $\boldsymbol{p}^\tau$ such that $\boldsymbol{D}_\epsilon(\boldsymbol{\pi}^\tau, \boldsymbol{p}^\tau) > 0$. α^τ is in the Curet heuristic determined by an exact line search.

The perturbed directional derivative acts as a lower bound for the true directional derivative, meaning that a step in the direction determined by the perturbed gradient improves the dual function sufficiently, but not as much as a step in the direction determined by the gradient of the Lagrangian dual function. The perturbed directional derivative used in the Curet heuristic is defined as follows:

$$\boldsymbol{D}_\epsilon(\boldsymbol{\pi}^\tau, \boldsymbol{p}^\tau) = (\boldsymbol{p}^\tau)^T \boldsymbol{d} - \sum_{w_{ij}>0} w_{ij} u_{ij}(\boldsymbol{\pi}^\tau, \epsilon) + \sum_{w_{ij}<0} w_{ij} l_{ij}(\boldsymbol{\pi}^\tau, \epsilon)$$

where $\boldsymbol{w} = \boldsymbol{E}^T \boldsymbol{p}^\tau$, and

$$u_{ij}(\boldsymbol{\pi}^\tau, \epsilon) = \begin{cases} u_{ij}, & \text{if } \frac{d\eta_{ij}(u_{ij})}{dv_{ij}} < 0 \text{ or } v_{ij}(\boldsymbol{\pi}^\tau) + \epsilon_{ij} > u_{ij}, \\ v_{ij}(\boldsymbol{\pi}^\tau) + \epsilon_{ij}, & \text{otherwise;} \end{cases} \quad \forall (i,j) \in A$$

$$l_{ij}(\boldsymbol{\pi}^\tau, \epsilon) = \begin{cases} l_{ij}, & \text{if } \frac{d\eta_{ij}(l_{ij})}{dv_{ij}} > 0 \text{ or } v_{ij}(\boldsymbol{\pi}^\tau) - \epsilon_{ij} < l_{ij}, \\ v_{ij}(\boldsymbol{\pi}^\tau) - \epsilon_{ij}, & \text{otherwise,} \end{cases} \quad \forall (i,j) \in A.$$

The terms $u_{ij}(\boldsymbol{\pi}^\tau, \epsilon)$ and $l_{ij}(\boldsymbol{\pi}^\tau, \epsilon)$ replace the current flow $\boldsymbol{v}_{ij}(\boldsymbol{\pi}^\tau)$ in the true directional derivative, and ϵ_{ij} denotes the ϵ perturbation for each $v_{ij}(\boldsymbol{\pi}^\tau)$. By definition, $u_{ij}(\boldsymbol{\pi}^\tau, \epsilon) \leq u_{ij}$ and $l_{ij}(\boldsymbol{\pi}^\tau, \epsilon) \geq l_{ij}$, which means that the capacity constraint of the primal problem will never be violated. The search for a primal feasible solution in the Curet heuristic is thus performed on the fol-

lowing problem:

find a $\boldsymbol{v}$ such that:

$$\sum_{j:(i,j)\in A} v_{ij} - \sum_{j:(j,i)\in A} v_{ji} = d_i, \qquad \forall i \in V, \tag{37}$$

$$l_{ij}(\boldsymbol{\pi}^\tau, \epsilon) \leq v_{ij} \leq u_{ij}(\boldsymbol{\pi}^\tau, \epsilon), \qquad \forall (i,j) \in A. \tag{38}$$

That the Curet heuristic produces a primal feasible solution from the solution by the truncated coordinate ascent method in a finite number of iterations is shown in Curet [7], Proposition 3.1 and Proposition 3.2. Theorem 14 states that the primal feasible solution produced by the Curet heuristic in fact is arbitrarily close to the primal optimal solution.

Theorem 14 (Curet, Proposition 3.3.) *Let $\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)$ be the primal feasible solution produced by the Curet heuristic, P^C, at termination. Then,*

$$\Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \eta(\boldsymbol{\pi}^\tau) \leq \sum_{(i,j)\in A} \epsilon_{ij}\sigma_{ij}$$

where $\sigma_{ij} = \max\{(\Phi'_{ij}(v_{ij}) - \Phi'_{ij}(w_{ij})) : l_{ij} \leq w_{ij}, v_{ij} \leq u_{ij}; |w_{ij} - v_{ij}| \leq \epsilon_{ij}\}$.

Proof By the definition of η, we have that

$$\Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \eta(\boldsymbol{\pi}^\tau) = \Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \Phi(\boldsymbol{v}(\boldsymbol{\pi}^\tau)) - \boldsymbol{\pi}^{\tau T}\boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \boldsymbol{\pi}^T\boldsymbol{d}. \tag{39}$$

Φ is a convex and differentiable function so for any $\boldsymbol{v}, \boldsymbol{\xi} \in \Re^{|V|}$ the following relation holds:

$$\Phi(\boldsymbol{\xi}) - \Phi(\boldsymbol{v}) \geq (\boldsymbol{\xi} - \boldsymbol{v})^T \bigtriangledown \Phi(\boldsymbol{v}).$$

The expression (39) can therefore be rewritten as follows:

$$\Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \eta(\boldsymbol{\pi}^\tau) \leq (\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau) - \boldsymbol{v}(\boldsymbol{\pi}^\tau))^T \bigtriangledown \Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \boldsymbol{\pi}^{\tau T}\boldsymbol{E}\boldsymbol{v}(\boldsymbol{\pi}^\tau) + \boldsymbol{\pi}^{\tau T}\boldsymbol{d}.$$

By adding and subtracting the term $\boldsymbol{E}\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)$ (which is feasible in problem (37)-(38)), we obtain the following relation:

$$\begin{aligned}\Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \eta(\boldsymbol{\pi}^\tau) &\leq (\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau) - \boldsymbol{v}(\boldsymbol{\pi}^\tau))^T(\bigtriangledown\Phi(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \boldsymbol{E}^T\boldsymbol{\pi}^\tau) \\ &\leq \sum_{(i,j)\in A} \epsilon_{ij}(\Phi'_{ij}(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \boldsymbol{\pi}^{\tau T}\boldsymbol{E}^{ij}),\end{aligned}$$

where $\boldsymbol{E}^{ij}$ denotes the corresponding column for arc (i,j) in matrix $\boldsymbol{E}$. By the definition of $\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)$, if $\frac{d\eta_{ij}(u_{ij})}{dv_{ij}} = \Phi'_{ij}(u_{ij}) - \boldsymbol{\pi}^{\tau T}\boldsymbol{E}^{ij} < 0$ then $\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau) = u_{ij}$, and if $\frac{d\eta_{ij}(l_{ij})}{dv_{ij}} = \Phi'_{ij}(l_{ij}) - \boldsymbol{\pi}^{\tau T}\boldsymbol{E}^{ij} > 0$ then $\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau) = l_{ij}$. So each term in the sum is bounded by $\epsilon_{ij}(|\Phi'_{ij}(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \Phi'_{ij}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))|)$. Hence, we can conclude that given bound $\sum_{(i,j)\in A}\epsilon_{ij}\sigma_{ij}$ in the theorem holds, since σ_{ij} is an upper estimation of $\Phi'_{ij}(\boldsymbol{v}^\epsilon(\boldsymbol{\pi}^\tau)) - \Phi'_{ij}(\boldsymbol{v}(\boldsymbol{\pi}^\tau))$. ∎

6 Network modeling and generation

This section describes our choices of data structure and network generator needed for the computational studies in Section 7. The object oriented (OO) principle was used to create the minimum cost flow network data structure described in Section 6.1. Section 6.2 describes a system according to Ohuchi and Kaji [18] that generates transportation networks.

6.1 Network modeling

The straightforward way to represent a minimum cost flow network, without considering any specific method of modeling, is to use vectors (arrays) and matrices (Evans [9]). Characteristics connected to the arcs can for instance be stored in matrices of size $|V| \times |A|$, so called node-arc incidence matrices, and characteristics connected to the vertices can be stored in arrays of size $|V|$. This is an easy way to model a network, but not the most descriptive one. The OO principle, on the other hand, takes into account the inherent relations of the ingoing objects that the network is made of. Hence, a data structure that provides easy access to specific elements of the network, as well as a good environment to speed up critical computational steps was developed using OO design. The data structure supports both bipartite and non-bipartite networks.

The network data structure have three main objects; the Vertex, the Arc, and the Network. The Vertex is modeled as an object that is recognized by its $index$, $demand$, and $price$ (dual variable). A source has a positive demand, a sink a negative demand, and a transshipment a demand equal to zero. Moreover, each Vertex has two arrays of indices, in and out, of those vertices connected to the current Vertex by incoming and outgoing arcs respectively. Because of graph searching reasons, the variables $from$, $marked$, $cost$, $level$, $predecessors$, and $capacity$ are also stored in the Vertex object. $from$ denotes the index of the vertex preceding the current Vertex, and is used in the BFS and DFS algorithms. $marked$ indicates, in any graph search, whether the current Vertex can be reached. $cost$ denotes the total cost (weight) to reach the current Vertex, and is used in the shortest path algorithm. $level$ denotes the level at which the current Vertex is found, and is used in the BS and DS algorithms. $predecessors$ denotes the set of indices of vertices preceding the current Vertex, and is used in the BS and DS algorithms. $capacity$ denotes the capacity of the current Vertex, and is used in the MFALD algorithm.

The Arc object stores the variables $from$ and to to denote the indices of the vertices at the current Arcs tail and head respectively. Further, infor-

mation about the lower and upper capacities of the Arc are stored in $lower$ and $upper$. The decision variable for the primal problem is stored in $flow$. The cost function for the current Arc is modeled as a freestanding object, a QuadFun, in which information about the linear a, quadratic, b, and constant coefficient, c, of the function is stored. We also store the variables $residual$, $marked$, $alpha_l$, and $alpha_u$ in the Arc object. $residual$ indicates whether the current Arc is contained in an original network or in a residual copy of a network. $marked$, $alpha_l$, and $alpha_u$ are used in the Helgason step length rule algorithm. $marked$ indicates if the current Arc should be considered when calculating $h(\alpha)$. $alpha_l$ and $alpha_u$ stores the values for the breakpoints $v_{ij}(\alpha)$ evaluated at the lower and upper limits of the current Arc.

The Network object itself keeps an $|V|$-length array of vertices, $vertices$, and a $|V| \times |V|$ matrix of arcs, $arcs$. Further, the Network contains two arrays, $sources$ and $sinks$, with indices of those vertices that have their $demand > 0$ and $demand < 0$ respectively. Several, mainly dual elements are stored in the Network object; $dual\_grad$ ($\boldsymbol{Ev - d}$), an array of length $|V|$; $dual\_grad\_norm\_square$ ($||\boldsymbol{Ev - d}||^2$), a scalar; $balance$ ($\boldsymbol{Ev}$), an array of length $|V|$; $price$ ($\boldsymbol{\pi}$), an array of length $|V|$; $demand\_norm\_square$ ($||\boldsymbol{d}||^2$), a scalar. The variables $path\_length$ and max_push are stored for graph searching reasons. $path\_length$ denotes the length of a path along which a specific amount of flow can be pushed, and max_push the same specific amount of flow. The Network object also stores the variables min_index, no_marked and no_alpha. min_index is used in the MFALD algorithm to store the index of the vertex with the minimum capacity. no_marked and no_alpha are used in the Helgason step length rule algorithm. no_marked indicates how many arcs that should be used when calculating $h(\alpha)$, and no_alpha stores the number of breakpoints that should be sorted by the Shellsort algorithm.

Most numerical variables have the long double precision type provided by the C programming language. Accessor and mutator functions were constructed as needed for all the above mentioned objects. The computer code for each object, along with the exact types of the variables that make up the objects, can be found under Section "Data structure related code" in Appendix A.

6.2 Network generation by Ohuchi and Kaji

The system by Ohuchi and Kaji generates complete transportation networks. In this system, the generated costs of the arcs become strictly convex, and

has for each arc the following form:

$$\Phi_{i,j} = b_{i,j} {v_{i,j}}^2 + a_{i,j} v_{i,j}, \qquad \forall (i,j) \in A$$

Values for the following parameters must be provided by the user:

* s: number of sources
* t: number of sinks
* $seed$: seed for the random number generator
* h: average amount of flow of an arc
* a maximum value of linear coefficient
* b maximum value of quadratic coefficient

The lower, l_{ij}, and upper, u_{ij}, limits of each arc are generated as uniform random numbers over the interval [0,100], with $l_{ij} < u_{ij}$. The coefficients $b_{i,j}$ and $a_{i,j}$ are generated from four uniform random distributions:

$$\begin{aligned}
\text{Set 1} &= \{0 \leq a_{i,j} \leq 10, 0 < b_{i,j} \leq 1\} \\
\text{Set 2} &= \{0 \leq a_{i,j} \leq 5, 0 < b_{i,j} \leq 2\} \\
\text{Set 3} &= \{0 \leq a_{i,j} \leq 2, 0 < b_{i,j} \leq 5\} \\
\text{Set 4} &= \{0 \leq a_{i,j} \leq 1, 0 < b_{i,j} \leq 10\}
\end{aligned}$$

For the values of each vertex demand and supply, the following formulas are used:

$$\begin{aligned}
\text{Supply at vertex i} &= \sum_k l_{i,k} + \sum_k (u_{i,k} - l_{i,k}) h, \qquad i \in S \\
\text{Demand at vertex j} &= \sum_p l_{p,j} + \sum_p (u_{p,j} - l_{p,j}) h, \qquad j \in T,
\end{aligned}$$

where S and T denotes the sets of sources and sinks respectively, and $h \in [0.1, 0.2, \ldots, 0.9]$ indicates if the average flow on the arcs should be closer to the arcs' lower or upper limits. Below the pseudocode for the generation of transportation networks according to Ohuchi and Kaji is presented.

OHUCHI&KAJI($sources$, $sinks$, $seed$, h, b, a)
1. **for** each source s
2. create arcs (s,t) to each sink t
3. set the lower limit of (s,t) to RAND(0,50)
4. set the upper limit of (s,t) to RAND(0,100), $l_{st} < u_{st}$
5. set the linear cost of (s,t) to RAND(0,a)
6. set the quadratic cost of (s,t) to RAND(0,b), $b_{st} > 0$
7. **for** each source s
8. set the supply of s to $\sum_k l_{s,k} + \sum_k (u_{s,k} - l_{s,k}) * h$
9. **for** each sink t
10. set demand of t to $\sum_p l_{p,t} + \sum_p (u_{p,t} - l_{p,t}) * h$
11. **return** the network

RAND denotes a function that when given two arguments, with the first argument strictly less the second argument, returns a value randomly selected between the first and second argument.

7 Results

This section sums up our choices of test problems, and the computational results performed. We show results for the Lagrangian heuristics Flowroute (Section5.3), FlowrouteD (Section5.4), FlowrouteBS (Section5.5), Shortest Path (Section5.6), and Shortest PathL (Section5.7). The Lagrangian heuristics are used in combination with the conjugate gradient method (Section 4.2.2) and an exact line search (Section 4.3.2). All computer programs have been written in the C programming language (Bilting&Skansholm [21]), compiled using the GNU C compiler with the gcc -O command, and run on a Sun Fire 480R under the SunOS 5.9 operating system. The computer code is found under Section "Algorithmic related code" in Appendix A. Note that in our implementation, the Lagrangian heuristic FlowrouteBS is a combination of FlowrouteBS and Flowroute. If the original FlowrouteBS reroutes all the flow in the residual network, our version of FlowrouteBS does the same. However, if the original FlowrouteBS fails to reroute all the flow in the residual network, our version of FlowrouteBS uses Flowroute to reroute the remaining flow. Note also that Shortest Path and Shortest PathL have an extra constant, $tolerance$, added to the cost function of each arc to avoid computationally generated negative cycles. The generated networks are described in Section 7.1; the computational results for dense transportation networks (DTNs) are presented in section 7.2.1; the computational results for sparse transportation networks (STNs) are presented in section 7.2.2. The tables in this section have the result of the Lagrangian heuristic that performed best for a certain network marked bold, and the Lagrangian heuristic that performed worst for a certian network marked slanted. The figures presented in this section have been plotted in MATLAB (Pärt–Enander&Sjöberg [8]).

7.1 Transportation problems

The tests for transportation networks were conducted on twelve different networks generated by the system described in Section 6.2. The selection of parameters shown in Table 1 creates large scale networks with different degrees of nonlinearity, and also permits the relationship between capacity and demand to be tested. Note that the odd numbered networks are have a more linear cost, and that the even numbered networks have a more nonlinear cost. Note also that $Ex.$ 1–6 have a total capacity far from the total demand, and that $Ex.$ 7–12 have a total capacity close to the total demand. All networks were generated with the seed for the random number generator equal to 13502460 as recommended by Klingman et al. [12].

Ex	s	t	h	a	b
1	50	50	0.3	10	1
2	50	50	0.3	1	10
3	100	100	0.3	10	1
4	100	100	0.3	1	10
5	100	200	0.3	10	1
6	100	200	0.3	1	10
7	50	50	0.7	10	1
8	50	50	0.7	1	10
9	100	100	0.7	10	1
10	100	100	0.7	1	10
11	100	200	0.7	10	1
12	100	200	0.7	1	10

Table 1: Selection of parameters for network generation according to Ohuchi and Kaji.

7.2 Computational results

In the presentation of the computational results, the following abbrevations for the different Lagrangian heuristics are used:

* F: Flowroute
* FBS: FlowrouteBS
* FD: FlowrouteD
* SP: Shortest Path
* SPL: Shortest PathL

In several of the quantities measured, the optimal value of the primal problem, $\Phi(\boldsymbol{v}^*)$, is used. To obtain a good estimation of $\Phi(\boldsymbol{v}^*)$, the conjugate gradient method has been stopped when the scaled gradient $\frac{||\bigtriangledown\eta(\boldsymbol{\pi}^\tau)||}{||\boldsymbol{d}||}$ is less than or equal to $1e^{-12}$.

In all the results presented, the Lagrangian heuristics have been run after stopping the conjugate gradient method at a predetermined percentage tolerance δ, which is defined as:

$$\delta = 100 * ((\Phi(\boldsymbol{v}^*) - \eta(\boldsymbol{\pi}^\tau))/\Phi(\boldsymbol{v}^*). \tag{40}$$

The quality Q, of a certain Lagrangian heuristic, at a final iteration τ for a given δ, is defined as the relative error in percent between the projected primal value given by the current heuristic, and the primal optimal value :

$$Q = 100 * ((\Phi(P(\boldsymbol{v}(\boldsymbol{\pi}^\tau))) - \Phi(\boldsymbol{v}^*)/\Phi(\boldsymbol{v}^*). \tag{41}$$

Further, the difference in quality of the solution returned by the dual scheme QD, and the solution given by a certain Lagrangian heuristic Q are presented. QD, at a final iteration τ for a given δ, is defined as the relative error between the dual value, and the primal optimal value:

$$QD = (\Phi(\boldsymbol{v}^*) - \eta(\boldsymbol{\pi}^\tau))/\Phi(\boldsymbol{v}^*) \tag{42}$$

The running time T, of a certain Lagrangian heuristic, at a final iteration τ for a given δ, is defined as the quotient in percent of the CPU time for the Lagrangian heuristic, CPU_{LH} and the average CPU time for one dual iteration, CPU_D. The time for one dual iteration have exclusively been measured as the time spent solving the Lagrangian dual subproblem.

$$T = 100 * (CPU_{LH}/CPU_D) \tag{43}$$

7.2.1 Dense transportation networks

The DTNs we have tested were complete bipartite networks that had their parameters set as in the twelve examples in Section 7.1.

Quality When the stopping criterion for the conjugate gradient method was $\delta \leq 0.1\%$ ($\tau = 5$), the performance in quality, Q, for the different heuristics is shown in Table 2.

Ex	Q_F	Q_{FBS}	Q_{FD}	Q_{SP}	Q_{SPL}
1	*1.171*	0.716*	1.080	0.381	**0.380**
2	0.869	0.786*	*0.896*	**0.593**	0.665
3	0.986	0.523*	*1.795*	0.374	**0.357**
4	1.375	0.740*	*2.393*	0.612	**0.601**
5	2.050	0.677*	*2.370*	0.632	**0.627**
6	*0.562*	0.394*	0.524	0.327	**0.313**
7	*1.535*	**0.325***	0.549	0.499	0.497
8	*1.776*	**0.141***	0.641	0.772	0.554
9	*0.830*	**0.106***	0.498	0.338	0.341
10	*1.000*	**0.190***	0.758	0.630	0.403
11	*1.465*	**0.290***	1.291	0.788	0.837
12	*1.777*	**0.426***	1.510	1.049	1.066

Table 2: Quality in solution for DTNs when $\delta \leq 0.1\%$. In FBS, a * denotes that Flowroute is used to reroute on average 10% of the initial flow imbalance.

When the stopping criterion for the conjugate gradient method was set to $\delta \leq 0.001\%$ ($\tau = 9$), the performance in quality, Q, for the different heuristics is shown in Table 3.

Ex	Q_F	Q_{FBS}	Q_{FD}	Q_{SP}	Q_{SPL}
1	*0.0877*	0.0650	0.0584	**0.0356**	0.0393
2	*0.1686*	0.0824	0.1388	**0.0571**	0.0573
3	*0.0943*	0.0742*	0.0763	0.0170	**0.0158**
4	*0.1332*	0.0951*	0.1311	0.0294	**0.0240**
5	*0.0906*	0.132*	0.0825	**0.0425**	0.0439
6	*0.1262*	0.0964	0.1149	**0.0748**	0.0750
7	*0.1597*	0.0603*	0.1070	**0.0558**	0.0598
8	*0.1709*	0.0780	0.0977	0.0839	**0.0632**
9	*0.0641*	0.0271*	0.0345	**0.0219**	0.0219
10	*0.0730*	0.0322*	0.0369	0.0389	**0.0256**
11	*0.1085*	0.0394*	0.0574	**0.0362**	0.0372
12	*0.1358*	0.0526*	0.0671	**0.0467**	0.0474

Table 3: Quality in solution for DTNs when $\delta \leq 0.001\%$. In FBS, a * denotes that Flowroute is used to reroute on average 3% of the initial flow imbalance. In problem 5, $\delta \leq 0.0014\%$.

Running time When the stopping criterion for the conjugate gradient method was $\delta \leq 0.1\%$ ($\tau = 5$), the performance in running time, T, for the different heuristics is shown in Table 4.

Ex	T_F	T_{FBS}	T_{FD}	T_{SP}	T_{SPL}
1	94	3259*	**62**	220,013	*290,394*
2	129	3681*	**65**	258,716	*345,309*
3	168	4897*	**104**	1,016,327	*1,370,664*
4	167	4439*	**103**	1,022,258	*1,327,400*
5	219	7288*	**132**	>2,000,000	*>2,000,000*
6	242	8314*	**87**	>2,000,000	*>2,000,000*
7	122	1988*	**61**	239,751	*311,725*
8	125	2067*	**63**	211,790	*280,173*
9	181	3468*	**67**	832,260	*991,966*
10	188	3324*	**83**	844,525	*1,055,102*
11	214	5525*	**105**	>2,000,000	*>2,000,000*
12	222	5611*	**111**	>2,000,000	*>2,000,000*

Table 4: Running time for DTNs when $\delta \leq 0.1\%$. In FBS, a * denotes that Flowroute is used to reroute on average 10% of the initial flow imbalance.

When the stopping criterion for the conjugate gradient method was $\delta \leq 0.001\%$ ($\tau = 9$), the performance in running time, T, for the different heuristics is shown in Table 5.

Ex	T_F	T_{FBS}	T_{FD}	T_{SP}	T_{SPL}
1	125	2444	**31**	213,025	*277,923*
2	129	2163	**65**	263,494	*345,309*
3	169	6888*	**65**	1,118,209	*1,355,418*
4	183	6425*	**79**	1,111,696	*1,399,605*
5	248	8657*	**80**	>2,000,000	*>2,000,000*
6	246	8752	**79**	>2,000,000	*>2,000,000*
7	122	2141*	**61**	211,762	*288,355*
8	125	2349	**94**	206,904	*253,359*
9	181	3422*	**68**	906,742	*1,242,600*
10	173	3354*	**60**	863,514	*1,096,975*
11	222	5988*	**87**	>2,000,000	*>2,000,000*
12	211	6073*	**89**	>2,000,000	*>2,000,000*

Table 5: Running time for DTNs when $\delta \leq 0.001\%$. In FBS, a * denotes that Flowroute is used to reroute on average 3% of the initial flow imbalance. In problem 5, $\delta \leq 0.0014\%$.

Convergence of Lagrangian heuristics vs. dual scheme When the stopping criterion for the conjugate gradient method was $\delta \leq 10^{-7}\%$ ($\tau = 20$) for the network in $Ex.1$, Figure 1 shows the convergence of the primal objective value given by the different heuristics vs. the dual objective value given by the dual scheme.

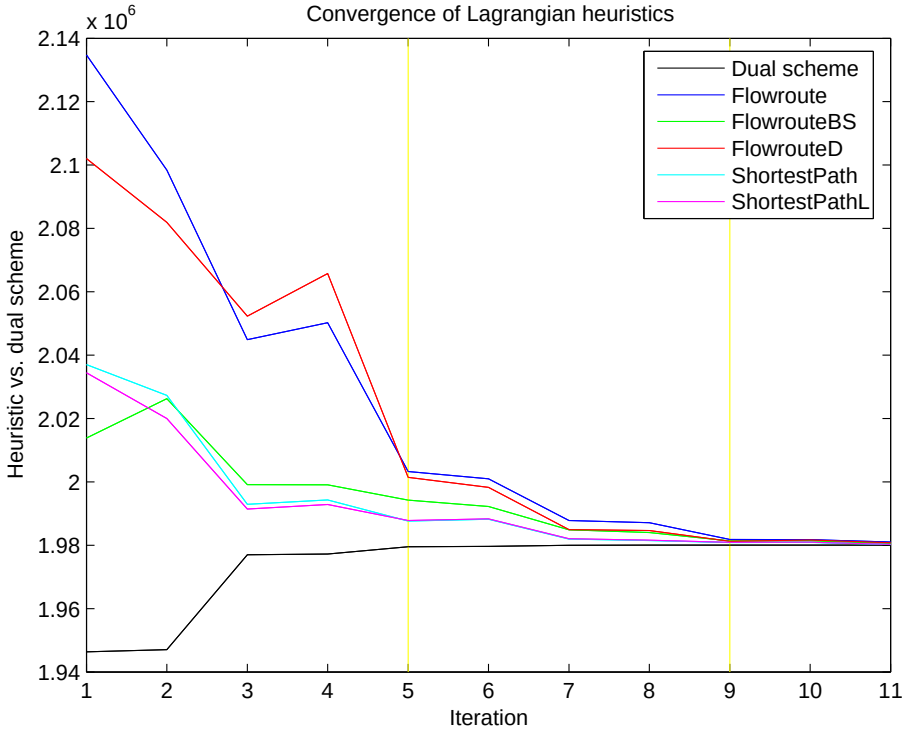


Figure 1: Convergence towards $\Phi^* = 1980065$ of the primal objective value given by the different heuristics and the dual objective value given by the dual scheme (DTNs, $Ex.1$). $\delta \leq 0.1\%$ shown at $\tau = 5$, and $\delta \leq 0.001\%$ shown at $\tau = 9$. FlowrouteBS produced a value below the optimal value at iteration $\tau = 0$.

Quality in Lagrangian heuristics vs. dual scheme When the stopping criterion for the conjugate gradient method was $\delta \leq 10^{-7}\%$ ($\tau = 20$) for the network in $Ex.1$, the difference between the solutions given by the different heuristics, $Q/100$, vs. the solutions given by the dual sheme, QD, is shown in Figure 2.

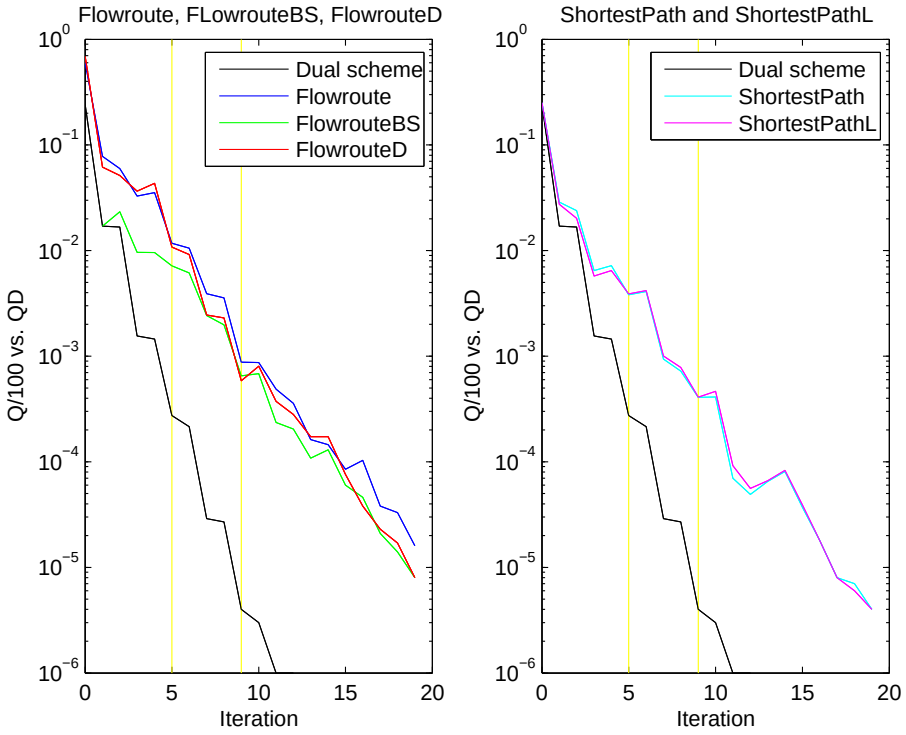


Figure 2: Each figure shows $Q/100$ (upper curves) for the different heuristics vs. QD (lower curve) plotted in logarithmic scale for each iteration τ (DTNs, $Ex.1$). $\delta \leq 0.1\%$ shown at $\tau = 5$, and $\delta \leq 0.1\%$ shown at $\tau = 9$. FlowrouteBS (left figure) produced a value below the optimal value at iteration $\tau = 0$. This value was excluded by MATLAB.

7.2.2 Sparse transportation networks

The STNs we have tested were bipartite networks where 10% of arcs existed (compared to the number of arcs in the same complete bipartite network). The parameters for the STNs were set as in the twelve examples in Section 7.1.

Quality When the stopping criterion for the conjugate gradient method was $\delta \leq 0.1\%$ ($\tau = 19$), the performance in quality, Q, for the different heuristics is shown in Table 6.

Ex	Q_F	Q_{FS}	Q_{FD}	Q_{SP}	Q_{SPL}
1	4.448	0.951	*4.836*	0.977	**0.914**
2	3.455	1.040	*6.820*	0.412	**0.391**
3	1.559	0.578	*3.800*	**0.431**	0.435
4	1.903	0.720	*4.287*	0.301	**0.292**
5	1.865	0.698	*5.105*	**0.345**	0.381
6	1.867	0.607	*8.279*	0.356	**0.305**
7	2.600	0.774	*7.549*	0.746	**0.675**
8	3.151	**0.578**	*6.332*	0.771	0.690
9	2.149	0.869	*6.123*	0.816	**0.783**
10	2.523	1.126	*6.511*	1.196	**1.122**
11	1.756	0.672*	*5.035*	**0.613**	0.621
12	1.920	0.767*	*5.440*	0.776	**0.662**

Table 6: Quality in solution for STNs when $\delta \leq 0.1\%$. In FBS, a * denotes that Flowroute is used to reroute on average <1% of the initial flow imbalance.

When the stopping criterion for the conjugate gradient method was set to $\delta \leq 0.001\%$ ($\tau = 3384$), the performance in quality, Q, for the different heuristics is shown in Table 7.

Ex	Q_F	Q_{BS}	Q_{FD}	Q_{SP}	Q_{SPL}
1	0.1397	0.0822	*0.2582*	0.0656	**0.0629**
2	0.1639	0.0945	*0.3403*	**0.0585**	0.0586
3	0.0810	0.0287	*0.3406*	0.0208	**0.0203**
4	0.2081	0.0732	*0.5462*	0.0548	**0.0540**
5	0.1284	0.0614*	*0.6083*	**0.0332**	0.0355
6	0.1591	0.0508	*0.5690*	0.0256	**0.0241**
7	0.0707	**0.0232**	*0.2049*	0.0358	0.0346
8	0.1178	**0.0370**	*0.3744*	0.0655	0.0619
9	0.1533	0.0614	*0.2821*	**0.0574**	0.0637
10	0.1479	0.0529	*0.3293*	0.0574	**0.0490**
11	0.1259	0.0561	*0.5311*	**0.0502**	0.0535
12	0.1286	0.0611	*0.6255*	0.0649	**0.0594**

Table 7: Quality in solution for STNs when $\delta \leq 0.001\%$. In FBS, a * denotes that Flowroute is used to reroute <1% of the initial flow imbalance.

Running time When the stopping criterion for the conjugate gradient method was $\delta \leq 0.1\%$ ($\tau = 19$), the performance in running time, T, for the different heuristics is shown in Table 8.

Ex	T_F	T_{FBS}	T_{FD}	T_{SP}	T_{SPL}
1	**212**	6369	**212**	185,751	*238,610*
2	**432**	5189	**432**	203,651	*265,914*
3	355	10,458	**295**	816,762	*1,119,406*
4	**470**	9357	530	815,564	*1,031,491*
5	645	14,977	**430**	2,081,528	*2,781,786*
6	**544**	14,479	635	2,005,486	*2,663,019*
7	636	4661	**424**	159,959	*188,985*
8	407	4478	**204**	136,384	*187,274*
9	**493**	8105	548	575,164	*766,174*
10	**592**	8445	**592**	623,079	*782,560*
11	631	13,753*	**500**	1,518,445	*1,965,905*
12	928	15,888*	**551**	1,666,675	*1,997,284*

Table 8: Running time for STNs when $\delta \leq 0.1\%$. In FBS, a * denotes that Flowroute is used to reroute on average <1% of the initial flow imbalance.

When the stopping criterion for the conjugate gradient method was $\delta \leq 0.001\%$ ($\tau = 3384$), the performance in running time, T, for the different heuristics were the following:

Ex	T_F	T_{FBS}	T_{FD}	T_{SP}	T_{SPL}
1	424	5307	**212**	155,182	*202,522*
2	**216**	4972	**216**	124,525	*167,331*
3	532	9454	**414**	1,196,515	*1,510,150*
4	470	10,005	**294**	895,456	*1,139,424*
5	645	16,696*	**368**	2,416,986	*2,857,747*
6	695	15,023	**423**	2025,285	*2,645,034*
7	**636**	5085	**636**	146,823	*182,417*
8	**204**	4071	407	174,857	*215,976*
9	493	8488	**274**	600,848	*786,819*
10	**430**	8552	**430**	568,538	*723,555*
11	710	12,991	**421**	1,285,524	*1,719,282*
12	899	13,917	**493**	1,620,778	*2,070579*

Table 9: Running time for STNs when $\delta \leq 0.001\%$. In FBS, a * denotes that Flowroute is used to reroute <1% of the initial flow imbalance.

Convergence of Lagrangian heuristics vs. dual scheme When the stopping criterion for the conjugate gradient method was $\delta \leq 0.005\%$ ($\tau = 228$) for the network in $Ex.1$, Figure 7.2.2 shows the convergence of the primal objective value given by the different heuristics vs. the dual objective value given by the dual scheme for $Ex.1$.

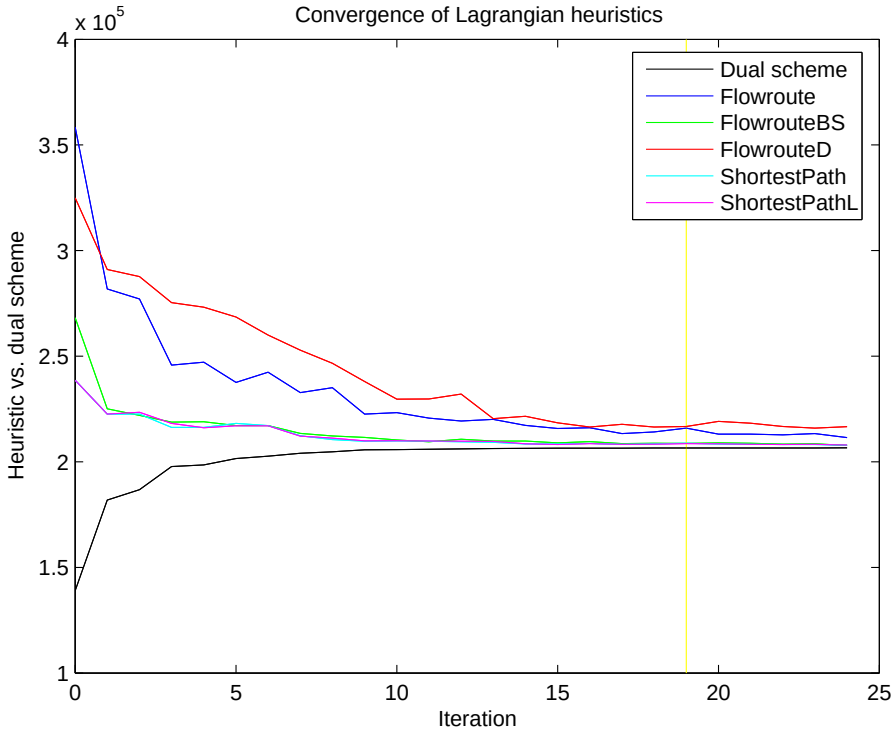


Figure 3: Convergence towards $\Phi^* = 206737$ of the objective primal value given by the different heuristics and the objective dual value given by the dual scheme (STNs, $Ex.1$). $\delta \leq 0.1\%$ shown at $\tau = 19$.

Quality in Lagrangian heuristic vs. dual scheme When the stopping criterion for the conjugate gradient method was $\delta \leq 0.005\%$ ($\tau = 228$) for the network in $Ex.1$, the difference between the solutions given by the different heuristics, $Q/100$, vs. the solutions given by the dual sheme, QD, is shown in Figure 4.

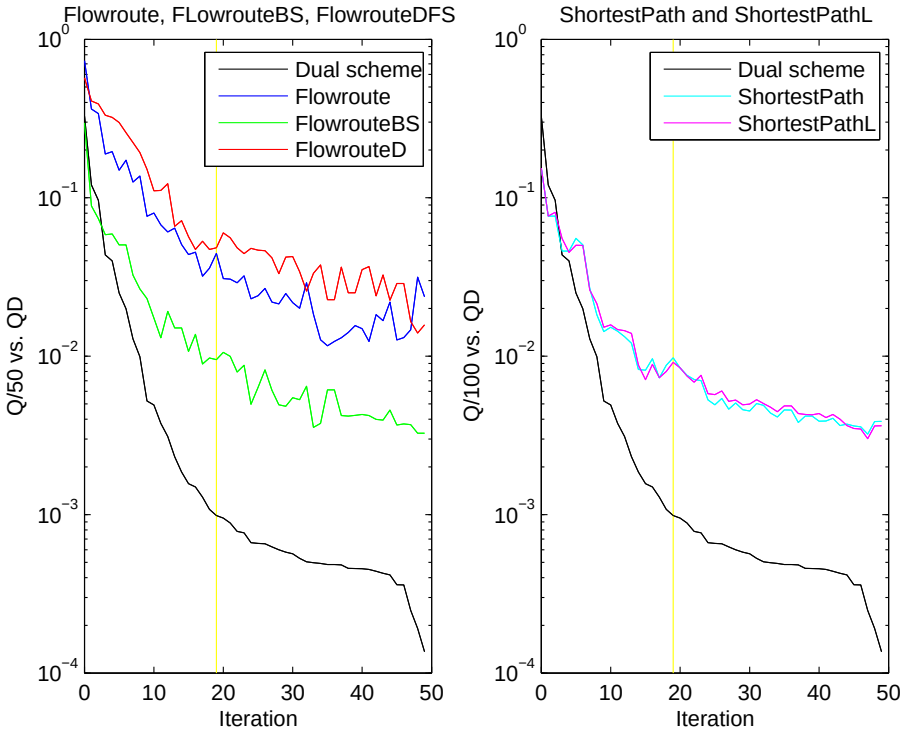


Figure 4: Each figure shows $Q/100$ (upper curves) for the different heuristics vs. QD (lower curve) plotted in logatritmic scale for each iteration τ (STNs, $Ex.1$). $\delta \leq 0.1\%$ shown at $\tau = 19$.

8 Discussion

The first aim was to verify the results given in the thesis by Marklund [15] for the original Flowroute and Shortest Path to obtain a reference for futher algorithmic development. Therefore, we had to conduct tests on similar networks. The second aim was to compare the performance in quality and time of the Lagrangian heuristics FlowrouteBS, FlowrouteD and Shortest PathL with the performance of the original Flowroute and Shortest Path.

We expected that FlowrouteD would produce results faster than, but of similar quality as Flowroute. We based these assumptions on the facts presented in Section 5.4, where we stated that DFS might reach a residual sink faster than BFS, and in that case we can expect FlowrouteD to terminate faster than Flowroute.

- The first assumption was confirmed for 100% of the DTNs, with FlowrouteD producing results 2.5 times faster than Flowroute on average. The running time was shorter for 70% of the more linear problems. The second assumption was confirmed in 90% of the DTNs; in 70% of $Ex.$ 1–6, and in 100% of $Ex.$ 7–12 FlowrouteD produced results of siginificanly better quality than Flowroute. The quality in the solutions was better for the more linear problems; in 70% of $Ex.$ 1–6, and in 80% of $Ex.$ 7–12.

- The first assumption was confirmed for 85% of the STDs, with FlowrouteD producing results 1.5 times faster than Flowroute on average. The running time was shorter for 80% of the more linear problems. The second assumption was contradicted for all transportation networks, but FlowrouteD faired comparatively better on the more linear problems; in 70% of $Ex.$ 1–6, and in 80% of $Ex.$ 7–12.

We believe that the reason that FlowrouteD produced results of lower quality than Flowroute for STNs can be explained by FlowrouteD having problems to find residual sinks when DFS is truncated prematurely.

We expected that FlowrouteBS would produce results faster than, but of similar quality as Flowroute. We based these assumptions on the facts presented in Section 5.5 where we state that the MFALD algorithm reroutes flow on many paths simultaneously in the residual network. Rerouting flow along many paths simultaneously might lead to that the residual sources are emptied faster, and in that case we can expect that FlowrouteBS terminates faster than Flowroute. However, unforseen difficulties were encountered when executing the original FlowrouteBS which terminated prematurely for

many networks. Therefore, we used the modified FlowrouteBS described in Section 7.

- The first assumption was contradicted in 100% of the DTNs, with FlowrouteBS producing results 25 times slower than Flowroute on average. FlowrouteBS faired comparatively better for 55% of the more linear problems; in 40% of $Ex.$ 1–6, and in 70% of $Ex.$ 7–12. The second assumption was contradicted by 100% of the DTNs. FlowrouteBS produced results of significantly better quality in 100% of the DTNs. The quality in the solutions was better for the more linear problems; in 80% of $Ex.$ 1–6 ($Ex.$ 5 excluded), and in 85% of $Ex.$ 7–12. Further, FlowrouteBS terminated prematurely in 85% of the DTNs.

- The first assumption was contradicted in 100% of the STNs, with FlowrouteBS producing results 20 times slower than Flowroute on average. FlowrouteBS faired comparatively better for 70% of the more nonlinear problems; in 85% of $Ex.$ 1–6, and in 50% of $Ex.$ 7–12. The second assumption was contradicted by 100% of the STNs. FlowrouteBS produced results of significantly better quality in 100% of the STNs. The quality in the solutions was better for 70% of the more linear problems. Further, FlowrouteBS terminated prematurely in 10% of the STNs.

We suspected that FlowrouteBS would produce results slower than Flowroute, as the running time of the MFALD algorithm is $O(|V|^3)$. However, we hoped that the experimental results would be different in reality. We did not forsee that FlowrouteBS would produce results of significanly better quality than Flowroute. One hypothesis is that the ALD created in FlowrouteBS always contain the minimum length paths from the supersource to the supersink, and that using fewer arcs to reroute flow is favourable in quality terms. Further, we did not forsee the difficulties with premature termination of the original FlowrouteBS. We found that the supersink of the (on average) third to fifth ALD created by the heuristic could not be reached from the supersource, which is cruical for the MFALD algorithm, and that this is one reason for the failure of the heuristic. Moreover, the MFALD algorithm is a modification of the Ford–Fulkerson algorithm that is known to terminate for integer data only. We use non integer data, and this could be another explanation for the failure of the original FlowrouteBS. However, more research must be conducted in order to explain these unforseen events.

We expected that Shortest PathL would produce results in similar running time as, but of similar or better quality than Shortest Path. These as-

sumptions were based on the facts presented in Section 5.7 where we motivated that using the derivative of the Lagrangian cost function instead of the linearized cost function of the primal problem in the shortest path search produces results closer to the optimal solution when the conjugate ascent method is stopped at a lower accuracy. Further, unforseen difficulties were experienced using the Bellman–Ford algorithm for the implementation of Shortest Path and Shortest PathL. Bellman–Ford ran for such a long time when applied to the larger DTNs, that reliable results of the running time could not be obtained for $Ex.$ 5,6, 11 and 12.

- The first assumption was contradicted in 100% of the DTNs, with Shortest PathL producing results 1.3 times slower than Shortest Path on average. The running time was not affected by the linearity of the problem on average, but Shortest PathL faired comparatively better for 75% of the more linear problems in $Ex.$ 1–4, and for 75% of the more nonlinear problems in $Ex.$ 7–10. The second assumption was confirmed in 85% of the DTNs, with the results for $\delta \leq 0.1$ having better quality in 80% of $Ex.$ 1–6, and in 50% of $Ex.$ 7–12; for $\delta \leq 0.001$ having better quality in 30% of $Ex.$ 1–6, and in 70% of $Ex.$ 7–12. The quality in the solutions was better for 90% of the more linear problems; 80% of $Ex.$ 1–6, and in 100% of $Ex.$ 7–12.

- The first assumption was contradicted in 100% of the STNs, with Shortest PathL producing results 1.3 times slower than Shortest Path on average. Shortest PathL faired comparatively better for 70% of the more nonlinear problems; in 100% of $Ex.$ 1–4, and in 25% of $Ex.$ 7–10. The second assumption was confirmed in 95% of the STNs, with the results for $\delta \leq 0.1$ beeing better in 85% of $Ex.$ 1–6, and in 100% of $Ex.$ 7–12; for $\delta \leq 0.001$ beeing better in 100% of $Ex.$ 1–6, and in 70% of $Ex.$ 7–12. The quality in the solutions was better for 70% of the more linear problems; 30% of $Ex.$ 1–6, and in 85% of $Ex.$ 7–12.

We believe that the longer running times in Shortest PathL are explained by the extra computations needed to calculate the derivative of the Lagrangian cost function.

9 Conclusion

This thesis presents results for five Lagrangian heuristics used in combination with the conjugate gradient method and the Helgason step size rule [11] applied to the dual of the strictly convex quadratic minimum cost network flow problem. Two new modifications of the Lagrangian heuristic Flowroute, FlowrouteBS and FlowrouteD, and one new modification of the Lagrangian heuristic Shortest Path, Shortest PathL have been studied. Tests on transportation networks with strictly quadratic costs generated according to the system by Ohuchi and Kaji [18] have been performed by first running the conjugate gradient method with an exact line search to obtain near-optimal dual solutions. Then the Lagrangian heuristics were applied to the dual solutions to obtain primal feasibility. The quality of the solutions were then evaluated, i.e. how close the solutions were compared to the optimal solution, and how good the quality of the solutions were compared to the solutions obtained from the dual scheme. The running times for the different Lagrangian heuristics were also evaluated.

We conclude that FlowrouteD performs better than Flowroute in time, and produces results of better quality than FLowroute for dense networks. We also conclude that FlowrouteBS performs worse than Flowroute in time regardless of the sparsity of the network, and that FlowrouteBS produces results of significantly better quality than Flowroute regardless of the sparsity of the network. The overall performance of FlowrouteD and FlowrouteBS is better for networks with a more linear cost function, and with a total capacity close to the total demand.

We conclude that Shortest PathL performs worse than Shortest Path in time regardless of the sparsity of the network. We also conclude that Shortest PathL produces results of better quality than Shortest Path when the conjugate ascent method is stopped at a lower accuracy. The overall performance of Shortest PathL is better for networks with a more linear cost function, but it is hard to draw any conclusions regarding the influence of the relationship between capacity and demand.

We conclude that the best results for transportation networks are given by the Lagrangian heuristic FlowrouteBS when considering the overall performance. The results given by FlowrouteBS have for most problems similar quality to the Shortest Path heuristics, and are obtained in reasonable time. However, if the quality of the solution is more important, our results show that Shortest PathL produces results of best quality. If the running time of the solution is more important, our results show that FlowrouteD is the best choice for dense networks, and Flowroute for sparse networks.

The tests in this thesis have been restricted to transportation networks,

but it would be interesting to compare the performance of the different Lagrangian heuristics on transshipment networks as well. Further, with the use of the Bellman–Ford algorithm in Shortest PathL, the presented running times are not at all competative with FlowrouteBS, but it would be interesting to improve the implementation for Shortest PathL, and compare the running times between FlowrouteBS and Shortest PathL. It would also be intersting to compare the performance between the fastest Lagrangian heuristics Flowroute or FlowrouteDS and any of the algorithms commonly used to solve strictly convex quadratic minimum cost network flow problems. Further, more research needs to be conducted regarding FlowrouteBS in order to explain its premature termination, and the reason for the exceptional good quality in its solutions.

References

[1] R. A. AHUJA et al., *Network Flows*, Prentice Hall, Inc., New Jersey, 1993.

[2] N. ANDRÉASSON, A. EVGRAFOV AND M. PATRIKSSON, *An Introduction to Continous Optimization: Foundations and Fundamental Algorithms*, Studentlitteratur, Lund, first ed., 2005.

[3] M. S. BAZARAA, H. D. SHERALI AND C. M. SHETTY, *Nonlinear Programming - Theory and Algorithms*, John Wiley and Sons, Inc., second ed., 1993.

[4] D. P. BERTSEKAS, *Nonlinear Programming*, Athena scientific, Belmont, Mass., second ed., 1999.

[5] N. L. BIGGS, *Discrete Mathematics*, Oxford University Press Inc., New York, second ed., 2002.

[6] T. H. CORMEN et al., *Introduction to Algorithms*, McGrawHill Book Company, England, second ed, 2001.

[7] N. D. CURET, *On the dual coordinate ascent approach for nonlinear networks*, Computers Ops. Res. vol 20 No.2 pp. 133-140, 1993.

[8] E. PÄRT–ENANDER AND A. SJÖBERG, *Användarhandledning för MATLAB 6*, Uppsala Universitet, 2001.

[9] J. R. EVANS, *Optimization Algorithms for Networks and Graphs*, Marcel Dekker, Inc., New York, second ed., 1992.

[10] M. T. HEATH, *Scientific Computing: An Introductory Survey*, McGraw–Hill, Singapore, 1997.

[11] R. HELGASON et al., *A polynomially bounded algorithm for a singly constrained quadratic program*, Mathematical Programming 18, pp. 338-343, 1980.

[12] D. KLINGMAN et al., *NETGEN: A program for generating large-scale capacitated assignment, transportation, and minimum cost flow network problems*, Management Science, Vol. 20, pp. 814-821, 1974.

[13] T. LARSSON AND M. PATRIKSSON, *Global optimality conditions for discrete and nonconvex optimization - With applications to Lagrangian heuristics and column generation*, Operations Research, Forthcoming.

[14] Z. LIU, *A Lagrangian dual scheme for structured linear programs with applications and extensions*, Dissertation No. 283, Linköping university, Linköping, 1992.

[15] J. MARKLUND, *A study of Lagrangian Heuristics for Convex Network Flow Problems*, Master of science thesis, Linköping university, Linköping, 1993.

[16] A. MIGDALAS AND M. GÖTHE-LUNDGREN, *Kombinatorisk optimering - problem och algoritmer*, Department of Mathematics, Linköping university, Linköping, 1994.

[17] S. G. NASH AND A. SOFER, *Linear and Nonlinear Programming*, MacGraw–Hill, Singapore, 1996.

[18] A. OHUCHI AND I. KAJI, *Lagrangian dual coordinate wise maximization algorithm for network transportation problems with quadratic costs*, Networks, Vol. 14 pp. 515-530, 1984.

[19] M. PATRIKSSON, *The book*, Forthcoming.

[20] R. SEDGEWICK, *Algorithms*, Addison–Wesley Publishing Company, Inc., second. ed., 1988.

[21] U. BILTING AND J. SKANSHOLM, *Vägen till C*, Studentlitteratur, Lund, third ed., 2000.

[22] J. VENTURA, *Computational development of a Lagrangian dual approach for quadratic networks*, Networks, Vol.21 (1991), pp. 469-485, 1991.

[23] D. R. WILKINS, *Getting Started with LaTex*, http://www.maths.tcd.ie/ dwilkins/LaTeXPrimer/, second ed., 1995.

A Computer code

A.1 Main program

A.1.1 OK.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Vertex.h"
#include "Arc.h"
#include "Network.h"
#include "NetworkUtil.h"
#include "Algorithms.h"
#include "Misc.h"
#include "Heuristics.h"
#include "Constants.h"
#include "TestGraphs.h"
#include "Alpha.h"

int main(int args, char *argv[]){

  Network *n;
  char graph_type, test_type;
  int no_sinks, no_sources, a, b, heur_type;
  unsigned int seed;
  double h, delta;
  long double opt_value;
  double opt_time;
  Result *proj_value, *dual_opt_value;

  dual_opt_value = (Result*)calloc(1, sizeof(Result));
  proj_value = (Result*)calloc(1, sizeof(Result));
  // Reading problem data
  // S for sparse network, D for dense network
  sscanf(argv[1], "%c", &graph_type);
  // Number of sources
  sscanf(argv[2],"%d", &no_sources);
  // Number of sinks
  sscanf(argv[3], "%d", &no_sinks);
  // Average amount of of flow on arcs, h
  sscanf(argv[4], "%lf", &h);
  // Upper limit of linear coefficient, a
  sscanf(argv[5], "%d", &a);
  // Upper limit of quadratic coefficient, b
  sscanf(argv[6], "%d", &b);
  // Seed to random number generator
  sscanf(argv[7], "%u", &seed);
  // Tolerance when to stop dual solver, delta
  sscanf(argv[8], "%lf", &delta);
  // O for finding the optimal dual value, H for test with heuristic
  sscanf(argv[9], "%c", &test_type);
  // 0-FLOWROUTE, 1-FLOWROUTEBS, 2-FLOWROUTED, 3-SHORTESTPATH, 4-SHORTESTPATHL
  sscanf(argv[10], "%d", &heur_type);
  // The optimal value for the current problem
  sscanf(argv[11], "%Lf", &opt_value);
  // The average CPU time of one dual subsolution
  sscanf(argv[12], "%lf", &opt_time);

  // Create graph type
  printf("%s\n", "CREATING NETWORK");
  if(graph_type--'D')
    n  = TestGraphs_GenTestBD(no_sources, no_sinks, h, 13502460, a, b);
  else
    n  = TestGraphs_GenTestBS(no_sources, no_sinks, h, 13502460, a, b);
  // Run desired test
  printf("%s\n", "WORKING...");
  if(test_type--'O'){
    Algorithms_CG(n, dual_opt_value);
    printf("Optimal value:                          %.16f\n",
           dual_opt_value->value);
    printf("Average CPU time for one dual iteration: %.16f\n",
           dual_opt_value->time);
  }
  else{
    Algorithms_ConjugateGradient(n, heur_type, delta, opt_value, proj_value);
    printf("Projected value:        %.16f\n", proj_value->value);
    printf("Quality of solution:    %.16Lf\n", 100*(
                                        (proj_value->value - opt_value)
                                        /opt_value));
```

```
    printf("CPU time for heuristic: %.16f\n", proj_value->time);
    printf("Time of solution:       %.16f\n",100*(proj_value->time/opt_time));
  }
  printf("%s\n", "DONE!");
  Network_Destroy(n);
  free(proj_value);
  free(dual_opt_value);
  return 0;
}
```

A.2 Algorithmic related code

A.2.1 Algorithms.c

```
#include "Algorithms.h"
#include "NetworkUtil.h"
#include "Queue.h"
#include "Stack.h"
#include "Constants.h"
#include "Heuristics.h"
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <float.h>
#include <time.h>
#include <stdio.h>

/*****************************************************************************
* PUBLIC FUNCTIONS
*****************************************************************************

/** Uses the Helgason et al. step length rule to determine the optimal
    stepsize
 *  among a vector of stepsizes for the dual line search problem of a
    network.
 *  @parm   n         network
 *  @param  alpha     vector with candidate stepsizes
 *  @param  d         vector of direction
 *  @return alpha_opt
 **/
double Algorithms_Helgason(Network *n, Alpha *alpha, long double
                           *direction){

  Vertex *vp;
  Arc *arc;
  //l-index_left, r-index_right, m-index_middle
  int l, r, m, i, j, k;
  //L-h(alpha_l), R-h(alpha_r), C-h(alpha_m), c-sum_i(demand_i*direction_i)
  double L, R, C, c, alpha_opt, a_ij, b_ij, price_i, price_j, direction_i,
         direction_j;
  QuadFun cost;
  bool done = false;
  bool first = true;

  L = R = C = l = m = c = alpha_opt = 0;
  r = (Network_NoArcs(n)*2 - (NetworkUtil_NoMarked(n)*2));
  while(!done){
    if((r-l)==1){
      break;
    }
    else{
      m = floor(0.5*(l+r));
      for(i=0; i<Network_NoVertices(n); i++){
        vp = &Network_Vertices(n)[i];
        price_i = Vertex_Price(vp);
        direction_i = direction[i];
        // calculate sum_i(demand_i*direction_i)
        if(first){
          c += Vertex_Demand(vp) * direction_i;
        }
        if(Vertex_NoOut(vp) != 0){
          for(k=0; k<Vertex_NoOut(vp); k++){
            j = Vertex_Out(vp)[k];
            arc = &(n->arcs[i][j]);
            cost = Arc_Cost(arc);
            a_ij = QuadFun_a(&cost);
            b_ij = QuadFun_b(&cost);
            price_j = Vertex_Price(&n->vertices[j]);
            direction_j = direction[j];
            assert(b_ij != 0);
            // calculate sums at left_index, right_index, middle_index
            if(first){
              L += (direction_i - direction_j)*(Algorithms_Mid(Arc_Lower(arc),
                (-(a_ij + price_i - price_j +((direction_i - direction_j)*
                   Alpha_Value(&alpha[l]))) / (2*b_ij)),
                Arc_Upper(arc)));
              R += (direction_i - direction_j)*(Algorithms_Mid(Arc_Lower(arc),
                (-(a_ij + price_i - price_j +((direction_i - direction_j)*
                   Alpha_Value(&alpha[r]))) / (2*b_ij)),
                Arc_Upper(arc)));
            }
            C += (direction_i - direction_j)*(Algorithms_Mid(Arc_Lower(arc),
```

```
                (-(a_ij + price_i - price_j +((direction_i - direction_j)*
                   Alpha_Value(&alpha[m]))) / (2*b_ij)),
                   Arc_Upper(arc)));
            }
          }
        }
        if(first){
          first = false;
        }
      }
      if(C--c){
        alpha_opt = Alpha_Value(&alpha[m]);
        done = true;
      }
      else if(C>c){
        l = m;
        L = C;
      }
      else{
        r = m;
        R = C;
      }
      C = 0;
    }
    if(!done){
      alpha_opt = Alpha_Value(&alpha[l]) + (((Alpha_Value(&alpha[r]) -
                  Alpha_Value(&alpha[l])) * (c - L)) / (R - L));
    }
    return(alpha_opt);
  }

  /** Solves the dual problem starting from the current price values,
   *  for each vertex, sets the prices that corresponds to the solution.
   *  Returns the value of the dual function at the current price and flow
      values.
   *  @param  n          network
   *  @param  heuristic  0-Flowroute, 1-FlowrouteBS, 2-FlowrouteD,
   *                     3-ShortestPath, 4-ShortestPathL
   *  @param  delta        tolerance when to stop dual method
   *  @param  opt_val      optimum value to problem
   *  @return result     primal_value and CPU time for choosen heuristic
   **/
  void Algorithms_ConjugateGradient(Network *n, int heuristic, double delta,
                                    long double opt_val, Result *result){

    double alpha, cpu_time_used;
    long double *dual_grad_old, *dual_grad_new, *direction, dual_grad_norm,
                dual_grad_square, dual_value, primal_value;
    Alpha *all_alpha, *all_alpha_new;
    clock_t start, end;
    int k = 0;
    bool done = false;
    bool first = true;
    Network *copy;
    double stop = opt_val*(100-delta)/100;

    printf("%s %d %s %f\n","CONJUGATE GRADIENT with heuristic ", heuristic, ",
           delta = ", delta);
    dual_value = alpha = primal_value = cpu_time_used = 0;
    dual_grad_old = (long double*)calloc(Network_NoVertices(n),
                    sizeof(long double));
    direction = (long double*)calloc(Network_NoVertices(n),
                sizeof(long double));
    all_alpha = (Alpha*)calloc(((Network_NoArcs(n)*2) + 1), sizeof(Alpha));
    all_alpha_new = (Alpha*)calloc(((Network_NoArcs(n)*2) + 1), sizeof(Alpha));
    start = end = clock();
    while(!done && (k++ < MAX_ITERATIONS)){
      // calculate the solution to the dual subproblem, w(v)
      dual_value = NetworkUtil_SolveDualSub(n, NULL, true);
      dual_grad_new = NetworkUtil_DualGrad(n);
      dual_grad_square = NetworkUtil_DualGradNormSquare(n);
      dual_grad_norm = sqrt(dual_grad_square);
      // termination criteria
      if(stop <= dual_value){
        done = true;
        // adjust to primal feasiblility
        switch(heuristic){
        case 0:
          copy = Network_ResidualCopy(n, false);
          start = clock();
          primal_value = Heuristics_Flowroute(copy);
          end = clock();
          break;
```

```
      case 1:
        copy = Network_ResidualCopy(n, true);
        start = clock();
        primal_value = Heuristics_FlowrouteBS(copy);
        end = clock();
        break;
      case 2:
        copy = Network_ResidualCopy(n, false);
        start = clock();
        primal_value = Heuristics_FlowrouteD(copy);
        end = clock();
        break;
      case 3:
        copy = Network_ResidualCopy(n, false);
        start = clock();
        primal_value = Heuristics_ShortestPath(copy);
        end = clock();
        break;
      case 4:
        copy = Network_ResidualCopy(n, false);
        start = clock();
        primal_value = Heuristics_ShortestPathL(copy);
        end = clock();
        break;
      default:
        copy = Network_ResidualCopy(n, false);
        primal_value = -1;
        break;
      }
      //printf("f(x)=%.16Lf\n", primal_value);
      result->value = primal_value;
      Network_Destroy(copy);
    }
    else{
      // ascent direction should be set to gradient every given iteration
      if((k%RESTART) == 0)
        direction = dual_grad_new;
      else
        Algorithms_FindDirection(dual_grad_new, dual_grad_old, direction,
                                 dual_grad_square, Network_NoVertices(n));
      // calculate the stepsize by Helgason et al.
      alpha =  Algorithms_CalcHelgasonAlpha(n, all_alpha, all_alpha_new,
                                            direction, first);
      if(first)
        first = false;
      // set the new price vector v_new = v + alpha*d
      NetworkUtil_UpdatePrice(n, alpha, direction);
      Algorithms_ArrayCopy(dual_grad_old, dual_grad_new, Network_NoVertices(n));
    }
  }
  printf("w(v)=%.16Lf\n", dual_value);
  cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
  result->time = cpu_time_used;
  free(all_alpha_new);
  free(all_alpha);
  free(dual_grad_old);
  free(direction);
}

/** Solves the dual problem starting from the current price values,
 *  for each vertex, sets the prices that corresponds to the solution.
 *  Returns the value of the dual function at the current price and flow
    values.
 *  @param  n          network
 *  @return result     dual_value and average time for one dual subsolution
 **/
void Algorithms_CG(Network *n, Result *result){

  double alpha, cpu_time_used;
  long double *dual_grad_old, *dual_grad_new, *direction, numerator, denominator,
              dual_grad_norm, dual_grad_square, dual_value;
   Alpha *all_alpha, *all_alpha_new;
  clock_t start, end;
  int k = 0;
  bool done = false;
  bool first = true;

  numerator = denominator = dual_value = alpha = cpu_time_used = 0;
  dual_grad_old = (long double*)calloc(Network_NoVertices(n), sizeof(long double));
  direction = (long double*)calloc(Network_NoVertices(n), sizeof(long double));
  all_alpha = (Alpha*)calloc(((Network_NoArcs(n)*2) + 1), sizeof(Alpha));
  all_alpha_new = (Alpha*)calloc(((Network_NoArcs(n)*2) + 1), sizeof(Alpha));
  while(!done && (k++ < MAX_ITERATIONS)){
```

```
    // calculate the solution to the dual subproblem, w(v)
    start - clock();
    dual_value - NetworkUtil_SolveDualSub(n, NULL, true);
    end - clock();
    dual_grad_new - NetworkUtil_DualGrad(n);

    dual_grad_square - NetworkUtil_DualGradNormSquare(n);
    dual_grad_norm - sqrt(NetworkUtil_DualGradNormSquare(n));
    // termination criteria
    numerator - dual_grad_norm;
    denominator - sqrt(NetworkUtil_DemandNormSquare(n));
    if((numerator/denominator) <- OPT_EPSILON)
      done - true;
    else{
      // ascent direction should be set to gradient every given iteration
       if((k%RESTART) -- 0)
         direction - dual_grad_new;
       else
         Algorithms_FindDirection(dual_grad_new, dual_grad_old, direction,
                                  dual_grad_square, Network_NoVertices(n));
      // calculate the stepsize by Helgason et al.
      alpha -  Algorithms_CalcHelgasonAlpha(n, all_alpha, all_alpha_new,
                                            direction, first);
      if(first)
        first - false;
      // set the new price vector v_new - v + alpha*d
      NetworkUtil_UpdatePrice(n, alpha, direction);
      Algorithms_ArrayCopy(dual_grad_old, dual_grad_new, Network_NoVertices(n));
    }
    //printf("grad w(v)/grad b - %.16Lf\n", (numerator/denominator));
    //printf("w(v)-%.16Lf\n", dual_value);
    cpu_time_used +- ((double) (end - start)) / CLOCKS_PER_SEC;
  }
  printf("%d %s\n", k, "iterations");printf("grad w(v)/grad b - %.16Lf\n",
          (numerator/denominator));
  free(all_alpha_new);
  free(all_alpha);
  free(dual_grad_old);
  free(direction);
  result->value - dual_value;
  result->time - cpu_time_used/k;
}

/** Uses the breadth first search (BFS) strategy to find paths
 *  in a network from a given source to any reachable sink.Only arcs
 *  that are not saturated are considered.
 *  Assumes all the vertices are unmarked and have from - -1 initially.
 *  @param  n network
 *  @param  s source
 **/
void Algorithms_BFS(Network *n, Vertex *s){

  Queue *queue;
  Vertex *vertices;
  int *ip, j, vertex_index, source_index, no_sinks;

  no_sinks - 0;
  queue - Queue_Create(Network_NoVertices(n));
  vertices - Network_Vertices(n);
  source_index - Vertex_Index(s);
  Vertex_SetMarked(s, true);
  Vertex_SetFrom(s, source_index);
  Queue_Add(queue, source_index);
  while(!Queue_Empty(queue)){
    vertex_index - Queue_First(queue);
    if(Vertex_IsSink(&vertices[vertex_index])){
      no_sinks++;
      if(no_sinks -- Network_NoSinks(n)){
        break;
      }
    }
    if(Vertex_NoOut(&vertices[vertex_index]) > 0){
      ip - Vertex_Out(&vertices[vertex_index]);
      for(j-0; j<Vertex_NoOut(&vertices[vertex_index]); j++){
        if(!Vertex_Marked(&vertices[*ip]) &&
           !Arc_Saturated(&(n->arcs[vertex_index][*ip]))){
          Vertex_SetMarked(&vertices[*ip], true);
          Vertex_SetFrom(&vertices[*ip], vertex_index);
          Queue_Add(queue, Vertex_Index(&vertices[*ip]));
        }
        ip++;
      }
    }
```

```
  }
  Queue_Destroy(queue);
}

/** Uses the multipel breadth first search (BS) strategy to find paths
 *  in a network from a supersource to a supersink.Only arcs that are not
 *  saturated is considered.
 *  Assumes the network has a supersource, ss and supersink added
 *  at indices  no_vertices-2, no_vertices-1 respectively.
 *  Assumes all the vertices are unmarked and have from - level - -1
    initially.
 *  @param  n network
 **/
void Algorithms_BS(Network *n){

  Queue *queue;
  Vertex *vertices, *ss;
  int *ip, j, vertex_index, source_index;

  queue - Queue_Create(Network_NoVertices(n));
  vertices - Network_Vertices(n);
  ss - &(Network_Vertices(n)[Network_NoVertices(n)-2]);
  source_index - Vertex_Index(ss);
  Vertex_SetMarked(ss, true);
  Vertex_SetLevel(ss, 0);
  Queue_Add(queue, source_index);
  while(!Queue_Empty(queue)){
    vertex_index - Queue_First(queue);
    if(Vertex_NoOut(&vertices[vertex_index]) > 0){
      ip - Vertex_Out(&vertices[vertex_index]);
      for(j-0; j<Vertex_NoOut(&vertices[vertex_index]); j++){
        if(!Vertex_Marked(&vertices[*ip]) &&
           !Arc_Saturated(&(n->arcs[vertex_index][*ip]))){
          Vertex_SetMarked(&vertices[*ip], true);
          Vertex_SetLevel(&vertices[*ip],
                          (Vertex_Level(&vertices[vertex_index]) + 1));
          Vertex_SetPre(&vertices[*ip], vertex_index);
          Queue_Add(queue, Vertex_Index(&vertices[*ip]));
        }
        else{
          if(Vertex_Level(&vertices[*ip])--
                           (Vertex_Level(&vertices[vertex_index]) + 1)){
            if(!Arc_Saturated(&(n->arcs[vertex_index][*ip])))
              Vertex_SetPre(&vertices[*ip], vertex_index);
          }
        }
        ip++;
      }
    }
  }
  Queue_Destroy(queue);
}

/** Uses the deapth first search (DFS) strategy to find a path
 *  in a network from a given source to a reachable sink. Only arcs
 *  that are not saturated is considered.
 *  Assumes all the vertices are unmarked and have from - -1 initially.
 *  @param  n network
 *  @param  s source
 *  @return sink_index
 **/
void Algorithms_DFS(Network *n, Vertex *s){

  Stack *stack;
  Vertex *vertices;
  int *ip, j, vertex_index, source_index, no_sinks;

  no_sinks - 0;
  stack - Stack_Create(Network_NoVertices(n));
  vertices - Network_Vertices(n);
  vertex_index - source_index - Vertex_Index(s);
  Vertex_SetMarked(s, true);
  Vertex_SetFrom(s, source_index);
  Stack_Push(stack, source_index);
  while(!Stack_Empty(stack)){
    vertex_index - Stack_Top(stack);
    if(Vertex_IsSink(&vertices[vertex_index]))
      break;
    if(Vertex_NoOut(&vertices[vertex_index]) > 0){
      ip - Vertex_Out(&vertices[vertex_index]);
      for(j-0; j<Vertex_NoOut(&vertices[vertex_index]); j++){
        if(!Vertex_Marked(&vertices[*ip]) &&
           !Arc_Saturated(&(n->arcs[vertex_index][*ip]))){
```

```
          Vertex_SetMarked(&vertices[*ip], true);
          Vertex_SetFrom(&vertices[*ip], vertex_index);
          Stack_Push(stack, *ip);
        }
        ip++;
      }
    }
  }
  Stack_Destroy(stack);
}

/** Finds the shortest paths in a network from a source to all other
    vertices.
 *  The cost functions associated with the arcs may have negative costs.
 *  Only arcs that are not saturated are considered.
 *  If lagrange-false, use linearized cost function of the primal problem,
 *  otherwise use linearized cost of Lagrange function.
 *  Returns true if paths can be found, false otherwise.
.*  Assumes that all the vertices have from - -1 and cost -
    DBL_MAX initially.
 *  @param  n        network
 *  @param  s        source vertex from which the search origins
 *  @param  lagrange false if linearized cost of primal problem should be
                     used
 *                   true if linearized cost of Lagrange function should
                     be used
 *  @return true     if no negative cycles are present
 *          false    otherwise
 **/
bool Algorithms_BellmanFord(Network *n, Vertex *s, bool lagrange){

  Vertex *vertex;
  int i, j, *j_p, iteration;
  long double cost_j, cost_i2j;

  Vertex_SetFrom(s, Vertex_Index(s));
  Vertex_SetCost(s, 0);
  //iterate over all non saturated arcs |V|-1 times
  for(iteration-0; iteration<Network_NoVertices(n)-1; iteration++){
     for(i-0; i<Network_NoVertices(n); i++){
       vertex - &Network_Vertices(n)[i];
       if(Vertex_NoOut(vertex) != 0){
           j_p - Vertex_Out(vertex);
           for(j-0; j<Vertex_NoOut(vertex); j++){
             if(!Arc_Saturated(&(Network_Arcs(n)[i][*j_p]))){
               if(!lagrange)
                 Algorithms_RelaxArc(n, &(Network_Arcs(n)[i][*j_p]), false);
               else
                 Algorithms_RelaxArc(n, &(Network_Arcs(n)[i][*j_p]), true);
             }
             j_p++;
           }
       }
     }
  }
  // check for negative cycle
  for(i-0; i<Network_NoVertices(n); i++){
    vertex - &Network_Vertices(n)[i];
    if(Vertex_NoOut(vertex) != 0){
        j_p - Vertex_Out(vertex);
        for(j-0; j<Vertex_NoOut(vertex); j++){
          if(!Arc_Saturated(&(Network_Arcs(n)[i][*j_p]))){
            cost_j - Vertex_Cost(&Network_Vertices(n)[*j_p]);
            if(!lagrange)
              cost_i2j - Vertex_Cost(vertex)+
               Arc_CurrLinearizedCost(&(Network_Arcs(n)[i][*j_p]))+TOLERANCE;
            else
              cost_i2j - Vertex_Cost(vertex)+
               Arc_CurrLinearizedLagrangeCost(&(Network_Arcs(n)[i][*j_p]),
               Vertex_Price(vertex),
               Vertex_Price(&(Network_Vertices(n)[*j_p])))+TOLERANCE;
            if(cost_j > cost_i2j){
              fputs("Bellman-Ford detected negative cycles.\n", stderr);
              exit(99);
              return(false);
            }
          }
          j_p++;
        }
    }
  }
  return(true);
}
```

```
/** Traces the reversed path from the given sink to the given source in
    the network.
 *  Simultaneously checks how much flow that can be pushed on the path,
    and
 *  updates the network with this value,(max_push) and the length of the
    path (path_length).
 *  Assumes that a graph search algorithm has been run.
 *  @param n            network
 *  @param sink_index
 *  @param source_index
 *  @param path_r       vertex indicies from sink to source
 **/
void Algorithms_TracePath(Network *n, int sink_index, int source_index,
                          int *path_r){

  int path_length, i, j;
  long double max_push;

  path_length = 0;
  max_push = -(Vertex_Demand(&Network_Vertices(n)[sink_index]));
  path_r[path_length++] = sink_index;
  j = sink_index;
  i = Vertex_From(&Network_Vertices(n)[j]);
  if(Arc_ResidualFlow(&(Network_Arcs(n)[i][j])) < max_push){
    max_push = Arc_ResidualFlow(&(Network_Arcs(n)[i][j]));
  }
  j = i;
  if(j != source_index){
    while(j != source_index){
      path_r[path_length++] = j;
      i = Vertex_From(&Network_Vertices(n)[j]);
      if(Arc_ResidualFlow(&(Network_Arcs(n)[i][j])) < max_push){
        max_push = Arc_ResidualFlow(&(Network_Arcs(n)[i][j]));
      }
      j = i;
    }
  }
  path_r[path_length++] = j;
  if(Vertex_Demand(&Network_Vertices(n)[j]) < max_push){
    max_push = Vertex_Demand(&Network_Vertices(n)[j]);
  }
  NetworkUtil_SetPathLength(n, path_length);
  NetworkUtil_SetMaxPush(n, max_push);
}

/** Updates the flows on the arcs on path with max_push value for network.
 *  Also updates the demand for the source and sink vertex on path.
 *  Assumes that the path is in order sink -> source, and that there is
    only
 *  one source and one sink on the path.
 *  @param n    network
 *  @param path
 **/
void Algorithms_UpdateFlowOnPath(Network *n, int *path){

  int i, row, col;
  Arc *arc_r;

  row = col = 0;
  if(NetworkUtil_MaxPush(n) > 0){
    for(i=NetworkUtil_PathLength(n)-2; i>=0; i--){
      col = path[i];
      row = path[i+1];
      arc_r = &(Network_Arcs(n)[row][col]);
      Arc_SetFlow(arc_r, (Arc_Flow(arc_r) + (NetworkUtil_MaxPush(n))));
    }
    Vertex_ChangeDemand(&(Network_Vertices(n)
                        [path[NetworkUtil_PathLength(n)-1]]),
                        NetworkUtil_MaxPush(n));
    Vertex_ChangeDemand(&(Network_Vertices(n)[path[0]]),
                        NetworkUtil_MaxPush(n));
  }
}

/** Uses the polynomial maximum flow algorithm for ALD's by Migdalas
 *  to reroute flow from the supersource to the supersink in the network.
 *  Simultaneously updates the demands of the supersource and supersink.
 *  Assumes the supersource has supply>0 initially.
 *  @param original original network in which flow also should be rerouted
 *  @param ald      network which holds the ald
 */
void Algorithms_MFALD(Network *original, Network *ald){
```

```
  int min_index;
  long double min_cap;
  Vertex *start;
  Network *ald_copy;
  bool done - false;

  Algorithms_SetCapacities(ald);
  ald_copy - Network_MFALDCopy(ald);
  min_index - Algorithms_SetCapacities(ald_copy);
  while(!done){
    if(min_index < 0){//Network_Print(ald);
      done - true;
      break;
    }
    start - &Network_Vertices(ald_copy)[min_index];
    min_cap - Vertex_Cap(start);
    Vertex_SetCap(&Network_Vertices(ald)[min_index], 0);
    Vertex_SetMarked(&Network_Vertices(ald)[min_index], false);
    // push flow to the supersink
    Algorithms_PushNPull(original, ald, ald_copy, start, min_cap, true);
    // pull flow to the supersource
    Algorithms_PushNPull(original, ald, ald_copy, start, min_cap, false);
    Network_Destroy(ald_copy);
    Algorithms_SetCapacities(ald);
    ald_copy - Network_MFALDCopy(ald);
    min_index - Algorithms_SetCapacities(ald_copy);
    //Network_Print(ald_copy);
    // continue or terminate depending on capacity of supersource
     if(Vertex_Cap(&Network_Vertices(ald_copy)[Network_NoVertices(ald)-2])<
        -0)
      done - true;
  }
  Network_Destroy(ald_copy);
}

/** Downloaded from http://linux.wku.edu/~lamonml/algor/sort/shell.html
 *  (homepage of Michael Lamont), modified to take Alpha objects.
 *  Shell sort sorts a given array of numbers in incrementing order.
 *  @param numbers    numbers to be sorted (Alpha_Value(&numbers[i]))
 *  @param array_size size of the array to be sorted
 **/
void Algorithms_ShellSort(Alpha numbers[], int array_size)
{
  int i, j, increment;
  Alpha *temp;

  temp - Alpha_Create(0, 0, 0, false);
  increment - 3;
  while (increment > 0)
  {
    for (i-0; i < array_size; i++)
    {
      j - i;
      temp->value - (&numbers[i])->value;
      temp->i - (&numbers[i])->i;
      temp->j - (&numbers[i])->j;
      temp->lower - (&numbers[i])->lower;
      while ((j >- increment) &&
               (Alpha_Value(&numbers[j-increment]) > Alpha_Value(temp)))
      {
        (&numbers[j])->value - (&numbers[j - increment])->value;
        (&numbers[j])->i - (&numbers[j - increment])->i;
        (&numbers[j])->j - (&numbers[j - increment])->j;
        (&numbers[j])->lower - (&numbers[j - increment])->lower;
        j - j - increment;
      }
      (&numbers[j])->value - temp->value;
      (&numbers[j])->i - temp->i;
      (&numbers[j])->j - temp->j;
      (&numbers[j])->lower - temp->lower;
    }
    if (increment/2 != 0)
      increment - increment/2;
    else if (increment == 1)
      increment - 0;
    else
      increment - 1;
  }
  free(temp);
}

/** Uses the Polak-Ribiere formula to compute the ascent direction, i.e.
```

```
 *  d_k+1 - dual_grad_k+1 + beta_k*d_k, where
 *  beta_k - (dual_grad_k+1(dual_grad_k+1-dual_grad_k)) /
             dual_grad_k^Tdual_grad_k.
 *  @param  new_gradient        dual_grad_k+1
 *  @param  old_gradient        dual_grad_k
 *  @param  old_direction       d_k
 *  @param  old_gradient_square dual_grad_k^Tdual_grad_k.
 *  @param  no_vertices
 *  @return new_direction       d_k+1
 **/
void Algorithms_FindDirection(long double *new_gradient, long double
                              *old_gradient, long double *old_direction,
                              long double old_gradient_square, int
                              no_vertices){

  double beta;

  assert(old_gradient_square != 0);
  beta = (Algorithms_ScalarProd(new_gradient, new_gradient, old_gradient,
          no_vertices)/old_gradient_square);
  Algorithms_AddArrays(new_gradient, old_direction, beta, old_direction,
                       no_vertices);
}

/** Copies the elements in the the old array to the new array
 *  assumes that the new array can hold at least size # of elements
 *  @param copy     the array to hold the copy
 *  @param original the array that holds the elements to be copied
 *  @param size     the size of the original array
 **/
void Algorithms_ArrayCopy(long double *copy, long double *original,
                          int size){

  int i;
  long double *d_c, *d_o;

  d_c = copy;
  d_o = original;
  for(i=0; i<size; i++){
    *d_c++ = *d_o++;
  }
}

/* Returns the middle value of three given values
 * @param  v_1
 * @param  v_2
 * @param  v_2
 * @return the middle value of (v_1, v_2, v_3)
 */
double Algorithms_Mid(double v_1, double v_2,double v_3){

  if(v_3<=v_2){
    if(v_3<=v_1){
      if(v_1<=v_2){
        //v_3<v_1<v_2
        return(v_1);
      }
      else{
        //v_3<v_2<v_1
        return(v_2);
      }
    }
    else{
      //v_1<v_3<v_2
      return(v_3);
    }
  }
  else{
    if(v_1<=v_3){
      if(v_1<=v_2){
        //v_1<v_2<v_3
        return(v_2);
      }
      else{
        //v_2<v_1<v_3
        return(v_1);
      }
    }
    else{
      //v_2<v_3<v_1
        return(v_3);
    }
  }
```

```
}

/*****************************************************************************
* PRIVATE FUNCTIONS
*****************************************************************************

/** Sets the capacities for the unmarked vertices of the network, and
 *  unmarkes those vertices that have a capacity equal to zero;
 *  Returns the array index for the vertex with minimum capacity, capacity
    > 0.
 *  @param  ald       network
 *  @return min_index
 **/
int Algorithms_SetCapacities(Network *ald){

  int i;
  long double in, out;
  Vertex *supersource, *supersink;
  int min_index = -1;
  long double min_cap = DBL_MAX;

  // calculate capacities for all marked vertices except the super-source
     and super-sink
  for(i=0; i<Network_NoVertices(ald)-2; i++){
    if(Vertex_Marked(&Network_Vertices(ald)[i])){
      in = NetworkUtil_DeltaFlow(ald, &Network_Vertices(ald)[i], true);
      out = NetworkUtil_DeltaFlow(ald, &Network_Vertices(ald)[i], false);
      if(in < out)
        Vertex_SetCap(&Network_Vertices(ald)[i], in);
      else
        Vertex_SetCap(&Network_Vertices(ald)[i], out);
      // unmark those vertices with a capacity equal to zero
      if(Vertex_Cap (&Network_Vertices(ald)[i]) == 0)
        Vertex_SetMarked(&Network_Vertices(ald)[i], false);
      if((Vertex_Marked(&Network_Vertices(ald)[i])) && (min_cap >
         Vertex_Cap(&Network_Vertices(ald)[i]))){
        min_cap = Vertex_Cap(&Network_Vertices(ald)[i]);
        min_index = i;
      }
    }
  }
  // calculate capacity for super-source
  supersource = &Network_Vertices(ald)[Network_NoVertices(ald)-2];
  Vertex_SetCap(supersource, NetworkUtil_DeltaFlow(ald, supersource, false));
  if((Vertex_Cap(supersource)>0) && (min_cap > Vertex_Cap(supersource))){
    min_cap = Vertex_Cap(supersource);
    min_index = Network_NoVertices(ald)-2;
  }
  supersink = &Network_Vertices(ald)[Network_NoVertices(ald)-1];
  Vertex_SetCap(supersink, NetworkUtil_DeltaFlow(ald, supersink, true));
  if((Vertex_Cap(supersink)>0) && (min_cap > Vertex_Cap(supersink))){
    min_cap = Vertex_Cap(supersink);
    min_index = Network_NoVertices(ald)-1;
  }
  return(min_index);
}

/* Calculates the scalar product between two arrays of same length,
 * where the product is taken as a1*(a2-a3).
 * @param  a1  array 1
 * @param  a2  array 2
 * @param  a3  array 3
 * @param  no  number of elements in array
 * @return sum
 */
double Algorithms_ScalarProd(long double *a1, long double *a2,
                             long double *a3, int no){

  long double sum;
  int i;

  sum = 0;
  for(i=0; i<no; i++){
    sum += a1[i]*a2[i] - a1[i]*a3[i];
  }
  return sum;
}

/* Adds two arrays elementwise and stores the result in result.
 * Multiplies array 2 with scalar before addition.
 * Assumes that both arrays are of same length.
 * @param  a1     array 1
 * @param  a2     array 2
```

```
 * @param  scalar
 * @param  result
 * @param  no     number of elements in array
 */
void Algorithms_AddArrays(long double *a1, long double *a2, double scalar,
                          long double *result, int no){

  int i;

  for(i=0; i<no; i++){
    result[i] = a1[i] + (scalar*a2[i]);
  }
}

/** Pushes (push=true) or pulls (push=false) flow from a given start vertex
 *  to the sink or source of an ald produced by a BS or DS search of a
    network.
 *  Simultaneousely  updates the flows in the original network. The demands
 *  of the supersource of the original network and ald are also updated if
 *  the push or pull procedure was successful.
 *  A copy of the ald is used to hold the current predecessor and out
    information.
 *  @param original   original network
 *  @param ald        original ald
 *  @param copy       copy of ald
 *  @param start      vertex
 *  @param flow       amount of flow to be pushed or pulled
 *  @param push       true or false
  **/
void Algorithms_PushNPull(Network *original, Network *ald, Network
                    *ald_copy, Vertex *start, long double flow, bool push){

  Queue *queue;
  Vertex *o_source, *a_source, *o_sink, *a_sink;
  int vertex_index, *neighbours;
  bool successful = false;

  neighbours = NULL;
  queue = Queue_Create(Network_NoVertices(original)*
                       Network_NoVertices(original));
  Queue_Add(queue, Vertex_Index(start));
  o_source = &Network_Vertices(original)[Network_NoVertices(original)-2];
  o_sink = &Network_Vertices(original)[Network_NoVertices(original)-1];
  while(!Queue_Empty(queue)){
    vertex_index = Queue_First(queue);
    if(vertex_index != Vertex_Index(start)){
      Vertex_SetCap(&Network_Vertices(ald)[vertex_index],
                (Vertex_Cap(&Network_Vertices(ald)[vertex_index])-flow));
      if(Vertex_Cap(&Network_Vertices(ald)[vertex_index]) == 0)
        Vertex_SetMarked(&Network_Vertices(ald)[vertex_index], false);
    }
    // pull flow towards the source
    if(push==false){
      if(Vertex_NoPre(&Network_Vertices(ald_copy)[vertex_index]) > 0){
        successful = true;
        neighbours = Vertex_Predecessors(
                               &Network_Vertices(ald_copy)[vertex_index]);
        Algorithms_AddFlowOnAdjacent(original, ald, flow, neighbours,
                              Vertex_NoPre(&Network_Vertices(ald_copy)
                              [vertex_index]), vertex_index, queue, false);
      }
    }
    // push flow towards the sink
    else{
      if(Vertex_NoOut(&Network_Vertices(ald_copy)[vertex_index]) > 0){

        successful = true;
        neighbours = Vertex_Out(&Network_Vertices(ald_copy)[vertex_index]);
        Algorithms_AddFlowOnAdjacent(original, ald, flow, neighbours,
                  Vertex_NoPre(&Network_Vertices(ald_copy)[vertex_index]),
                  vertex_index, queue , true);
      }
    }
  }
  // adjust the demands of the supersource of the original network, and ald
  if(push==false && successful){
    Vertex_SetDemand(o_source, (Vertex_Demand(o_source)-flow));
    a_source = &Network_Vertices(ald)[Network_NoVertices(ald)-2];
    Vertex_SetDemand(a_source, (Vertex_Demand(a_source)-flow));
  }
  if(push==true && successful){
    Vertex_SetDemand(o_sink, (Vertex_Demand(o_sink)+flow));
    a_sink = &Network_Vertices(ald)[Network_NoVertices(ald)-1];
```

```
    Vertex_SetDemand(a_sink, (Vertex_Demand(a_sink)+flow));
  }
  Queue_Destroy(queue);
}

/** Adds amount flow to the arcs defined by an index and its neighbours
    in an ald.
 *  If push is true, the neighbours are taken from the current vertex
    outgoing arcs,
 *  if false from the vertex predecessors.
 *  Assumes that the arcs has enough capacity to add amount flow between
    them.
 *  @param original   original network
 *  @param ald
 *  @param flow       amount of flow that should be added
 *  @param neighbours array that holds the indices of the vertices
                      adjacent to index
 *  @param index      index of the current vertex
 *  @param queue      holds indicies of those vertices with involved arcs
 *  @param push       true or false
 **/
void Algorithms_AddFlowOnAdjacent(Network *original, Network *ald,
                                  long double flow, int *neighbours,
                                  int no_neighbours, int index,
                                  Queue *queue , bool push){

  Arc *arc_a,*arc_o;// arc_a-ald arc, arc_o-original arc
  long double curr_flow = flow;
  int iterator = 0;

  while(curr_flow>0 && iterator<no_neighbours){
    // use predecessor arcs
    if(!push){
      arc_a = &Network_Arcs(ald)[*neighbours][index];
      arc_o = &Network_Arcs(original)[*neighbours][index];
    }
    // use outgoing arcs
    else{
      arc_a = &Network_Arcs(ald)[index][*neighbours];
      arc_o = &Network_Arcs(original)[index][*neighbours];
    }
    // Arc_Print(arc_o);Arc_Print(arc_a);
    if(Arc_ResidualFlow(arc_a) >= curr_flow){
      Arc_ChangeFlow(arc_a, curr_flow);
      Arc_ChangeFlow(arc_o, curr_flow);
      curr_flow = 0;
    }
    else{
      curr_flow -= Arc_ResidualFlow(arc_a);
      Arc_SetFlow(arc_a, Arc_Upper(arc_a));
      Arc_SetFlow(arc_o, Arc_Upper(arc_o));
    }
    Queue_Add(queue, *neighbours++);
    iterator++;
  }
}

/** Reduces the cost to an vertex  j in a shortest path search if
 *  the current cost of the vertex is higher than the sum of a
 *  current neighbour vertex i and the arc (i,j) between them.
 *  To avoid negative cycles by computer arithmetic, a small tolerance is
 *  added to the flow of each arc.
 *  @param n        network that holds the array of vertices in the network
 *  @param arc      arc (i,j)
 *  @param lagrange false if linearized cost of primal problem should be
                    used
 *                  true if linearized cost of Lagrange function should be
                    used
 **/
void Algorithms_RelaxArc(Network *n, Arc *arc, bool lagrange){

  int i = Arc_From(arc);
  int j = Arc_To(arc);
  long double tmp_cost = DBL_MAX;

  if(!lagrange)
    tmp_cost = Vertex_Cost(&Network_Vertices(n)[i]) +
               Arc_CurrLinearizedCost(arc) + TOLERANCE;
  else
    tmp_cost = Vertex_Cost(&Network_Vertices(n)[i]) +
               Arc_CurrLinearizedLagrangeCost(arc,
               Vertex_Price(&Network_Vertices(n)[i]),
               Vertex_Price(&Network_Vertices(n)[j])) + TOLERANCE;
```

```
  if(Vertex_Cost(&Network_Vertices(n)[j]) > tmp_cost){
    Vertex_SetCost(&Network_Vertices(n)[j], tmp_cost);
    Vertex_SetFrom(&Network_Vertices(n)[j], i);
  }
}

/** Calculates the step size according to Helgason et al.
 *  If first-true, the candidate stepsizes are set in all_alpha,
 *  otherwise all_alpha from previous iteration is sorted, now traverse
    sorted all_alpha
 *  and replace current values with the new values that for each alpha is
    found
 *  in the corresponding arc, store the new alphas in all_alpha_new with
    elements
 *  equal to -1 in all_alpha removed.
**/
double Algorithms_CalcHelgasonAlpha(Network *n, Alpha *all_alpha,
                                    Alpha *all_alpha_new, long double
                                    *direction, bool first){

  double alpha;
  int old_alpha_length;

  // set the candidate alphas in all_alpha
  if(first){
    NetworkUtil_CalcAlphas(n, all_alpha, direction, true);
    Algorithms_ShellSort(all_alpha, ((Network_NoArcs(n)*2) + 1 -
                                     (NetworkUtil_NoMarked(n)*2)));
    alpha = Algorithms_Helgason(n, all_alpha, direction);
  }
  // traverse sorted all_alpha and replace values,
  // sort all_alpha_new, and apply Helgason
  else{
    old_alpha_length = NetworkUtil_NoAlpha(n);
    NetworkUtil_CalcAlphas(n, all_alpha, direction, false);
    Algorithms_ReplaceAlphas(n, all_alpha, all_alpha_new, old_alpha_length);
    Algorithms_ShellSort(all_alpha_new, NetworkUtil_NoAlpha(n));
    alpha = Algorithms_Helgason(n, all_alpha_new, direction);
  }
  if(!first)
    Algorithms_AlphaArrayCopy(all_alpha, all_alpha_new,
                              NetworkUtil_NoAlpha(n));
  return(alpha);
}

/** Takes an array of sorted alpha values with length number of elements,
    and
 *  replaces each value with the corresponding value of the arc in the
    network.
 *  Then removes all elements from the array that have a value equal to -1.
 *  @param n          network
 *  @param alpha_old  array of sorted alpha values from previous iteration
 *  @param alpha_new  array of to hold (un)sorted alpha values from
                      current iteration
 *  @param length     number of sorted elements in alpha_old
 **/

void Algorithms_ReplaceAlphas(Network *n, Alpha *alpha_old,
                              Alpha *alpha_new, int length){

  int i, from, to, j;
  Arc *arc;

  j-0;
  for(i-0; i<length; i++){
    from - Alpha_From(&alpha_old[i]);
    to - Alpha_To(&alpha_old[i]);
    arc - &Network_Arcs(n)[from][to];
    if(Alpha_Lower(&alpha_old[i]))
      Alpha_SetValue(&alpha_old[i], Arc_AlphaL(arc));
    else
      Alpha_SetValue(&alpha_old[i], Arc_AlphaU(arc));
  }
  for(i-0; i<Network_NoArcs(n)*2 + 1; i++){
    if(Alpha_Value(&alpha_old[i])!--1)
      alpha_new[j++] - alpha_old[i];
  }
}

/** Copies the elements in the the old array to the new array
 *  assumes that the new array can hold at least size # of elements
 *  @param copy     the array to hold the copy
 *  @param original the array that holds the elements to be copied
```

```
 *  @param size     the size of the original array
 **/
void Algorithms_AlphaArrayCopy(Alpha *copy, Alpha *original, int size){

  int i;
  Alpha *a_c, *a_o;

  a_c = copy;
  a_o = original;
  for(i=0; i<size; i++){
    *a_c++ = *a_o++;
  }
}
```

A.2.2 Alpha.c

```
#include <assert.h>
#include <stdlib.h>
#include "Alpha.h"

/*typedef struct{
  double value; // the current value of alpha
  int i;        // index of vertex at arcs tail which alpha is associated with
  int j;        // index of vertex at arcs head which alpha is associated with
  bool lower;   // true if alpha is evaluated at the lower limit of arc (i,j),
                   false otherwise
  }Alpha;*/

/**************************************************************************
* PUBLIC FUNCTIONS
**************************************************************************/

/** Allocates memory for an alpha.
 *  @param  value value of alpha
 *  @param  i     index of vertex at arcs tail which alpha is associated with
 *  @param  j     index of vertex at arcs head which alpha is associated with
 *  @param  lower true if alpha evaluated at lower limit, false otherwise
 *  @return alpha
 **/
Alpha* Alpha_Create(double value, int from, int to, bool lower){

  Alpha *alpha = (Alpha *)calloc(1, sizeof(Alpha));
  alpha->value = value;
  alpha->i = from;
  alpha->j = to;
  alpha->lower = lower;
  return(alpha);
}

/** Returns memory for an alpha.
 *  @param aalpha
 **/
void Alpha_Destroy(Alpha *alpha){

  assert(alpha!=NULL);
  free(alpha);
}

/** Gets the value for alpha.
 *  @param  alpha
 *  @return value
 **/
double Alpha_Value(Alpha *alpha){

  assert(alpha!=NULL);
  return(alpha->value);
}

/** Sets the value for alpha.
 *  @param alpha
 *  @param value
 **/
void Alpha_SetValue(Alpha *alpha, double value){

  assert(alpha!=NULL);
  alpha->value = value;
}

/** Gets the value for index at arcs tail to which alpha is associated.
 *  @param  alpha
 *  @return i
 **/
int Alpha_From(Alpha *alpha){

  assert(alpha!=NULL);
  return(alpha->i);
}

/** Sets the value for index at arcs tail to which alpha is associated.
 *  @param alpha
 *  @param i
 **/
void Alpha_SetFrom(Alpha *alpha, int i){

  assert(alpha!=NULL);
  alpha->i = i;
}
```

```
/** Gets the value for index at arcs head to which alpha is associated.
 *  @param  alpha
 *  @return j
 **/
int Alpha_To(Alpha *alpha){

  assert(alpha!=NULL);
  return(alpha->j);
}

/** Sets the value for index at arcs head to which alpha is associated.
 *  @param alpha
 *  @param j
 **/
void Alpha_SetTo(Alpha *alpha, int j){

  assert(alpha!=NULL);
  alpha->j = j;
}

/* Gets the value for lower. True if alpha evaluated at lower limit,
   false otherwise.
 * @param  alpha
 * @return true or false
 */
bool Alpha_Lower(Alpha *alpha){

  assert(alpha!=NULL);
  return(alpha->lower);
}

/* Sets the value for lower of alpha.
 * @param alpha
 * @param lower true if alpha evaluated at lower limit, false otherwise
 */
void Alpha_SetLower(Alpha *alpha, bool lower){

  assert(alpha!=NULL);
  alpha->lower = lower;
}
```

A.2.3 Constants.h

```
#ifndef _Constants_h
#define _Constants_h
#include <stdlib.h>


#define CORR_EPSILON 0.0000000000000001 //-16

#define OPT_EPSILON 0.000000000001 //-12

#define EPSILON 0.00000001 //-8

#define TOLERANCE MIN 1000

#define MAX_ITERATIONS 200000

#define RESTART 100

typedef struct{
  double value;      // value returned by process
  double time;       // CPU time for process
}Result;

#endif
```

A.2.4 Heuristics.c

```
#include "Heuristics.h"
#include "Algorithms.h"
#include "NetworkUtil.h"
#include "Queue.h"
#include "Constants.h"
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <float.h>

/*****************************************************************************
* PUBLIC FUNCTIONS
*****************************************************************************/

/** Uses the Flowroute method with the BFS search strategy to adjust the
 *  current flows of the network to primal feasibility.
 *  Returns the primal objective value after adjustment.
 *  Assumes there is a feasible solution.
 *  @param n      network
 *  @return value sum f_ij(x_ij)
 **/
long double Heuristics_Flowroute(Network *n){

  int i, j, *path;
  Vertex *source, *sink;

  path = (int*)calloc(Network_NoVertices(n), sizeof(int));
  for(i=0; i<Network_NoSources(n); i++){
    source = &(Network_Vertices(n)[Network_Sources(n)[i]]);
    Algorithms_BFS(n, source);
    for(j=0; j<Network_NoSinks(n); j++){
      sink = &(Network_Vertices(n)[Network_Sinks(n)[j]]);
      if(Vertex_Marked(sink)){
        Algorithms_TracePath(n, Vertex_Index(sink), Vertex_Index(source), path);
        Algorithms_UpdateFlowOnPath(n, path);
      }
    }
    NetworkUtil_UnMark(n);
  }
  free(path);
  return(NetworkUtil_PrimalValue(n));
}

/** Uses the Flowroute method with the DFS search strategy to adjust the
 *  current flows of the network to primal feasibility.
 *  Returns the primal objective value after adjustment.
 *  Assumes there is a feasible solution.
 *  @param n      network
 *  @return value sum f_ij(x_ij)
 **/
long double Heuristics_FlowrouteD(Network *n){

  int i, j, *path;
  Vertex *source, *sink;
  bool done = false;

  path = (int*)calloc(Network_NoVertices(n), sizeof(int));
  for(i=0; i<Network_NoSources(n); i++){
    source = &(Network_Vertices(n)[Network_Sources(n)[i]]);
    while(!done){
      Algorithms_DFS(n, source);
      for(j=0; j<Network_NoSinks(n); j++){
        sink = &(Network_Vertices(n)[Network_Sinks(n)[j]]);
        if(Vertex_Marked(sink)){
          Algorithms_TracePath(n, Vertex_Index(sink), Vertex_Index(source),
path);
          Algorithms_UpdateFlowOnPath(n, path);
          if(Vertex_Demand(source) < OPT_EPSILON){
            done = true;
            break;
          }
        }
      }
      NetworkUtil_UnMark(n);
    }
    done = false;
  }
  free(path);
  return(NetworkUtil_PrimalValue(n));
}
```

```
/** Uses the Flowroute method with the BS search strategy, and the
 *  MFALD method to adjust the current flows of the network to primal
    feasibility.
 *  If the super-source has supply left after FlowrouteBS has been run,
 *  Flowroute empties the remaining supply.
 *  Returns the primal objective value after adjustment.
 *  Assumes that network has a supersource and a supersink, and that
 *  there is a feasible solution.
 *  @param copy   network
 *  @return value sum f_ij(x_ij)
 **/
long double Heuristics_FlowrouteBS(Network *copy){

  Network *ald;
  int i, source_index, sink_index;
  Arc arc;
  bool residual = false;

  while(Vertex_Demand(&Network_Vertices(copy)[Network_NoVertices(copy)-2]) >
        OPT_EPSILON){
    Algorithms_BS(copy);
    if(!Vertex_Marked(&Network_Vertices(copy)[Network_NoVertices(copy)-1])){
      residual = true;
      break;
    }
    ald = Network_ALDCopy(copy);
    Algorithms_MFALD(copy, ald);
    Network_Destroy(ald);
    NetworkUtil_UnMark(copy);
    }
  if(residual){
    copy->no_sources -= 1;
    copy->no_sinks -= 1;
    for(i=0; i<Network_NoSources(copy)-1; i++){
      source_index = Network_Sources(copy)[i];
      arc = Network_Arcs(copy)[Network_NoVertices(copy)-2][source_index];
      Vertex_SetDemand(&Network_Vertices(copy)[source_index],
     (Vertex_Demand(&Network_Vertices(copy)[source_index])-Arc_Flow(&arc)));
    }
    for(i=0; i<Network_NoSinks(copy)-1; i++){
      sink_index = Network_Sinks(copy)[i];
      arc = Network_Arcs(copy)[sink_index][Network_NoVertices(copy)-1];
      Vertex_SetDemand(&Network_Vertices(copy)[sink_index],
        (Vertex_Demand(&Network_Vertices(copy)[sink_index])+Arc_Flow(&arc)));
    }
    return(Heuristics_Flowroute(copy));
  }
  else
    return(NetworkUtil_PrimalValue(copy));
}

/** Uses the Shortest Path method to adjust the current flows of the network
 *  to primal feasibility.
 *  Assumes there is a feasible solution, and that all reachable sinks are
 *  unmarked initially.
 *  @param n network
 *  @return value sum f_ij(x_ij)
 **/
long double Heuristics_ShortestPath(Network *n){

  int i, j, *path, min_index;
  Vertex *source, *sink;
  bool ok = false;

  path = (int*)calloc(Network_NoVertices(n), sizeof(int));
  for(i=0; i<Network_NoSources(n); i++){
    source = &(Network_Vertices(n)[Network_Sources(n)[i]]);
    if((ok = Algorithms_BellmanFord(n, source, false))){
      for(j=0; j<Network_NoSinks(n); j++){
        min_index = Heuristics_FindMinSink(Network_Vertices(n),
                                           Network_Sinks(n), Network_NoSinks(n));
        sink = &Network_Vertices(n)[min_index];
        Vertex_SetMarked(sink, true);
        Algorithms_TracePath(n, Vertex_Index(sink), Vertex_Index(source),
                             path);
        Algorithms_UpdateFlowOnPath(n, path);
      }
      NetworkUtil_UnMark(n);
    }
    else
      break;
  }
  free(path);
```

```
  if(ok)
    return(NetworkUtil_PrimalValue(n));
  else
    return(-1);
}

/** Uses the Shortest Path method used for the linearized Lagrange function wrt x
 *  to adjust the current flows of the network to primal feasibility.
 *  Assumes there is a feasible solution.
 *  @param n network
 *  @return value sum f_ij(x_ij)
 **/
long double Heuristics_ShortestPathL(Network *n){

  int i, j, *path, min_index;
  Vertex *source, *sink;
  bool ok = false;

  path = (int*)calloc(Network_NoVertices(n), sizeof(int));
  for(i=0; i<Network_NoSources(n); i++){
    source = &(Network_Vertices(n)[Network_Sources(n)[i]]);
    if((ok = Algorithms_BellmanFord(n, source, true))){
      for(j=0; j<Network_NoSinks(n); j++){
        min_index = Heuristics_FindMinSink(Network_Vertices(n),
                                           Network_Sinks(n), Network_NoSinks(n));
        sink = &Network_Vertices(n)[min_index];
        Vertex_SetMarked(sink, true);
        Algorithms_TracePath(n, Vertex_Index(sink), Vertex_Index(source),
                             path);
        Algorithms_UpdateFlowOnPath(n, path);
      }
      NetworkUtil_UnMark(n);
    }
    else
      break;
  }
  free(path);
  if(ok)
    return(NetworkUtil_PrimalValue(n));
  else
    return(-1);
}

/*****************************************************************************
* PRIVATE FUNCTIONS
*****************************************************************************/

/** Finds the sink with the minimum cost among the set of reachable sinks
 *  that are unmarked from a source, and returns that sinks index.
 *  @param  vertices
 *  @param  sinks
 *  @param  no_sinks size of the array holding sinks
 *  @return min_index
 **/
int  Heuristics_FindMinSink(Vertex *vertices, int *sinks, int no_sinks){

  int i, min_index;
  long double min_cost;
  Vertex *sink;

  min_cost = DBL_MAX;
  min_index = -1;
  for(i=0; i<no_sinks; i++){
    sink = &vertices[sinks[i]];
    if((Vertex_Cost(sink) != DBL_MAX) && (!Vertex_Marked(sink))){
      if(min_cost > Vertex_Cost(sink)){
        min_cost = Vertex_Cost(sink);
        min_index = Vertex_Index(sink);
      }
    }
  }
  return(min_index);
}
```

A.3 Data structure related code

A.3.1 Arc.c

```
#include "Arc.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "Constants.h"

/*typedef struct{
  int from;              // index of vertex at arcs tail
  int to;                // index of vertex at arcs head
  double lower;          // lower limit for flow on arc, i.e. l_ij
  double upper;          // upper limit for flow on arc, i.e. u_ij
  QuadFun cost;          // f_ij(x_ij) = c_ij + a_ij*x_ij + b_ij*x_ij^2
  long double flow;      // flow on arc, i.e. x_ij
  bool residual;         // false if arc original, true if arc residual
  bool marked;           // true if arc removed from Helgason, false otherwise
  double alpha_l;        // the value for alpha evaluated at the lower limit
                            of arc
  double alpha_u;        // the value for alpha evaluated at the upper limit
                            of arc
  }Arc; */

/*************************************************************************
* PUBLIC FUNCTIONS
*************************************************************************/

/** Allocates memory for an arc.
 *  @param  from  vertex index for the tail of the arc
 *  @param  to    vertex index for the head of the arc
 *  @param  lower the lower flow-limit of the arc
 *  @param  upper the upper flow-limit of the arc
 *  @return arc
 **/
Arc *Arc_Create(int from, int to, double lower, double upper, QuadFun *cost){

  Arc *arc;

  arc = (Arc *)calloc(1, sizeof(Arc));
  arc->from = from;
  arc->to = to;
  arc->lower = lower;
  arc->upper = upper;
  (&arc->cost)->a = QuadFun_a(cost);
  (&arc->cost)->b = QuadFun_b(cost);
  (&arc->cost)->c = QuadFun_c(cost);
  arc->flow = 0; // 1 for testing purposes, otherwise 0 initially
  arc->residual = arc->marked = false;
  arc->alpha_l = arc->alpha_u = -1;
  QuadFun_Destroy(cost);
  return arc;
}

/** Returns memory for an arc.
 *  @param arc
 **/
void Arc_Destroy(Arc *arc){

  free(arc);
}

/* Sets the lower limit of an arc.
 * @param arc
 * @param lower
 */
void Arc_SetLower(Arc *arc, double lower){

  assert(arc != NULL);
  arc->lower = lower;
}

/** Gets the lower limit of an arc.
 *  @param  arc
 *  @return lower
 **/
double Arc_Lower(Arc *arc){

  assert(arc != NULL);
  return arc->lower;
}
```

```
/* Sets the upper limit of an arc.
 * @param arc
 * @param upper
 */
void Arc_SetUpper(Arc *arc, double upper){

  assert(arc != NULL);
  arc->upper = upper;
}

/** Gets the upper limit of an arc.
 *  @param  arc
 *  @return upper
 **/
double Arc_Upper(Arc *arc){

   assert(arc != NULL);
   return arc->upper;
}

/** Gets the cost function to traverse an arc.
 *  @param  arc
 *  @return cost
 **/
QuadFun Arc_Cost(Arc *arc){

   assert(arc != NULL);
   //Arc_Print(arc);
   return arc->cost;
}

/** Gets the flow for an arc.
 *  @param  arc
 *  @return flow
 **/
long double Arc_Flow(Arc *arc){

  assert(arc != NULL);
  return arc->flow;
}

/** Sets the flow on arc to new_flow.
 *  @param arc
 *  @param new_flow
 **/
void Arc_SetFlow(Arc *arc, long double new_flow){

  assert(arc != NULL);
  arc->flow = new_flow;
}

/** Returns true if the arc has flow (i.e. x_ij != 0),
 *  otherwise false.
 *  @param  arc
 *  @return true or false
 **/
bool Arc_HasFlow(Arc *arc){

  assert(arc != NULL);
  return(!(Arc_Flow(arc) == 0));
}

/** Changes the flow for an arc by given amount if the change is within limits,
 *  otherwise nothing happens.
 *  @param arc
 *  @param amount
 **/
void Arc_ChangeFlow(Arc *arc, long double amount){

  long double new;

  assert(arc != NULL);
  new = (arc->flow) + amount;
  if(Arc_ValidChange(new, arc->upper, arc->lower)){
     arc->flow = new;
  }
}

/** Returns the value of how much flow can be added to the arc based on
 *  the current flow and the upper limit of the arc.
 *  @param arc
 *  @return value
 **/
```

```
long double Arc_ResidualFlow(Arc *arc){

  assert(arc != NULL);
  return(arc->upper - arc->flow);
}

/** Returns true if the arc is saturated, false otherwise.
 *  @param arc
 *  @return true or false
 **/
bool Arc_Saturated(Arc *arc){

  assert(arc != NULL);
  return((arc->upper - arc->flow) -- 0);//< OPT_EPSILON);
}

/** Gets the value for residual on arc.
 *  Returns true if arc is an residual arc, otherwise false.
 *  @param  arc
 *  @return true or false
**/
bool Arc_Residual(Arc *arc){

  assert(arc != NULL);
  return(arc->residual);
}

/** Sets the value for residual on arc to residual.
 *  @param arc
 *  @param remove
**/
void Arc_SetResidual(Arc *arc, bool residual){

  assert(arc != NULL);
  arc->residual = residual;
}

/** The cost in copy is set to minus the cost in original.
 *  @param copy
 *  @param original
 **/
void Arc_SetResidualCost(Arc *copy, Arc *original){

  QuadFun original_cost ;

  assert(copy != NULL);
  assert(original != NULL);
  original_cost = Arc_Cost(original);
  (&copy->cost)->a = -(QuadFun_a(&original_cost));
  (&copy->cost)->b = -(QuadFun_b(&original_cost));
  (&copy->cost)->c = -(QuadFun_c(&original_cost));
}

/** Gets the value for marked on arc.
 *  Returns true if arc has d_i-d_j-0, otherwise false.
 *  @param  arc
 *  @return true or false
**/
bool Arc_Marked(Arc *arc){

  assert(arc != NULL);
  return(arc->marked);
}

/** Sets the value for marked on arc to marked.
 *  @param arc
 *  @param mark
**/
void Arc_SetMarked(Arc *arc, bool mark){

  assert(arc != NULL);
  arc->marked = mark;
}

/** Calculates the cost for the current flow.
 *  @param  arc
 *  @return sum
 **/
long double Arc_CurrCost (Arc *arc){

  QuadFun f;

  assert(arc != NULL);
```

```
  f = Arc_Cost(arc);
  return (QuadFun_Value(&f, Arc_Flow(arc)));
}

/** Calculates the cost for the current linearized flow.
 *  @param  arc
 *  @return sum
 **/
long double Arc_CurrLinearizedCost (Arc *arc){

  QuadFun cost;

  assert(arc != NULL);
  cost = Arc_Cost(arc);
  return ((2*QuadFun_b(&cost)*Arc_Flow(arc)) + QuadFun_a(&cost));
}

/** Calculates the cost for the current linearized Lagrange function
 *  evaluated for the current flow.
 *  @param  arc
 *  @param  price_i dual variable associated with vertex at arcs tail
 *  @param  price_j dual variable associated with vertex at arcs head
 *  @return sum
 **/
long double Arc_CurrLinearizedLagrangeCost (Arc *arc, long double price_i,
                                            long double price_j){

  QuadFun cost;

  assert(arc != NULL);
  cost = Arc_Cost(arc);
  return ((2*QuadFun_b(&cost)*Arc_Flow(arc)) + QuadFun_a(&cost) +
          price_i - price_j);
}

/** Sets the index of the vertex at the tail of the arc.
 *  @param arc
 *  @param from
 **/
void Arc_SetFrom(Arc *arc, int from){

  assert(arc != NULL);
  arc->from = from;
}

/** Returns the index of the vertex at the tail of the arc.
 *  @param  arc
 *  @return from
 **/
int Arc_From(Arc *arc){

  assert(arc != NULL);
  return arc->from;
}

/** Sets the index of the vertex at the head of the arc.
 *  @param arc
 *  @param to
 **/
void Arc_SetTo(Arc *arc, int to){

  assert(arc != NULL);
  arc->to = to;
}

/** Returns the index of the vertex at the head of the arc.
 *  @param  arc
 *  @return head
 **/
int Arc_To(Arc *arc){

  assert(arc != NULL);
  return arc->to;
}

/* Sets the value for alpha_l of the arc.
 * @param arc
 * @param alpha_l
 */
void Arc_SetAlphaL(Arc *arc, double alpha_l){

 assert(arc != NULL);
 arc->alpha_l = alpha_l;
```

```
}

/* Returns the value for alpha_l of the arc.
 * @param  arc
 * @return alpha_l
 */
double Arc_AlphaL(Arc *arc){

  assert(arc != NULL);
  return(arc->alpha_l);
}

/* Sets the value for alpha_u of the arc.
 * @param arc
 * @param alpha_u
 */
void Arc_SetAlphaU(Arc *arc, double alpha_u){

 assert(arc != NULL);
 arc->alpha_u = alpha_u;
}

/* Returns the value for alpha_u of the arc.
 * @param  arc
 * @return alpha_u
 */
double Arc_AlphaU(Arc *arc){

  assert(arc != NULL);
  return(arc->alpha_u);
}

/** Returns true if arc holds the "empty" arc,
 *  otherwise false.
 *  @param  arc
 *  @return true or false
 **/
bool Arc_Empty(Arc *arc){

  assert(arc != NULL);
  return ((arc->from--0) && (arc->to--0) && (arc->upper--0));
}

/** Sets all the values of arc to zero
 *  @param arc
 **/
void Arc_MkEmpty(Arc *arc){

  arc->from = arc->to = 0;
  arc->lower = arc->upper = arc->flow = 0;
  (&arc->cost)->a = (&arc->cost)->b = (&arc->cost)->c = 0;
}

/** String representation of an arc.
 *  @param arc
 **/
void Arc_Print(Arc *arc){

  assert(arc != NULL);
  printf("Arc from vertex %d to vertex %d has:\n", arc->from, arc->to);
  printf("current flow = %Lf\n", arc->flow);
  printf("cost         = %Lf\n", (QuadFun_Value(&(arc->cost), arc->flow)));
  printf("lower limits = %f\n", arc->lower);
  printf("upper limits = %.16f\n", arc->upper);
  printf("residual     = %d\n", arc->residual);
  printf("marked       = %d\n", arc->marked);
  printf("%s", "cost         = ");
  QuadFun_Print(&arc->cost);
}

/*************************************************************************
* PRIVATE FUNCTIONS
*************************************************************************/

/** Returns true if the parameter new is within limits lower to upper,
 *  otherwise false.
 *  @param  new   the parameter to check if within limits
 *  @param  upper upper limit
 *  @param  lower lower limit
 *  @return true or false
 **/
bool Arc_ValidChange(long double new, double upper, double lower){
```

```
  if(new<-upper && new>-lower)
    return true;
  else
    return false;
}
```

A.3.2 Network.c

```
#include "Network.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <float.h>
#include "Arc.h"
#include "Misc.h"
#include "Constants.h"

/*typedef struct{
   int *sources;                    // array with indices of the source-
                                       vertices in network
  int *sinks;                       // array with indices of the sink-
                                       vertices in network
  Vertex *vertices;                 // array with the vertices of network
  Arc **arcs;                       // matrix with the arcs of network
  int max_sources;                  // maximum size of *sources
  int max_sinks;                    // maximum size of *sinks
  int max_vertices;                 // maximum size of *vertices
  int no_arcs;                      // current size of **arcs
  int no_sources;                   // current size of *sources
  int no_sinks;                     // current size of *sinks
  int no_vertices;                  // current size of *vertices
  long double *price;               // array with the dual variables of
                                       the vertices in network, i.e. v
  long double *dual_grad;           // the dual gradient of network,
                                       i.e. grad w(v) - Ax-b
  long double *balance;             // the current balance of each vertex,
                                       i.e. Ax
  long double dual_grad_norm_square; // the norm of the dual gradient of
                                       network, i.e. ||grad w(v)||
  long double demand_norm_square;   // the norm of the demand of the
                                       vertices in network, i.e. ||b||
  int path_length;                  // pathlength of current path from
                                       source to sink in residaul network
  long double max_push;             // maximum amount of flow that can be
                                       pushed on current path
  int min_index;                    // index of vertex with minimum
                                       capacity
  int no_marked;                    // number of marked arcs in network
  int no_alpha;                     // no_arcs*2 + 1 - no_marked*2
  }Network;*/

/*********************************************************************
* PUBLIC FUNCTIONS
*********************************************************************

/** Allocates memory for a network.
 *  @param  max_vertices the maximum number of vertices in the network
 *  @return network
 **/
Network *Network_Create(int max_vertices){

  Network *n;
  int i;

  n = (Network *)calloc(1, sizeof(Network));
  n->max_vertices = max_vertices;
  n->vertices = (Vertex *)calloc(max_vertices, (sizeof(Vertex)));
  n->arcs = (Arc **)calloc(max_vertices, (sizeof(Arc*)));
  for(i=0; i<max_vertices; i++){
    n->arcs[i] = (Arc *)calloc(max_vertices, (sizeof(Arc)));
  }
  n->price = (long double *)calloc(max_vertices, (sizeof(long double)));
  n->dual_grad = (long double *)calloc(max_vertices, (sizeof(long double)));
  n->balance = (long double *)calloc(max_vertices, (sizeof(long double)));
  n->sources = n->sinks = NULL;
  return n;
}

/** Allocates memory for a bipartite graph.
 *  @param  max_sources the number of sources of the bipartite graph
 *  @param  max_sinks    the number of sinks of the bipartite graph
 *  @return graph
 **/
Network *Network_CreateB(int max_sources, int max_sinks){

  Network *b;
  int max_vertices;

  max_vertices = max_sources+max_sinks;
```

```
  b = Network_Create(max_vertices);
  b->max_sources = max_sources;
  b->max_sinks = max_sinks;
  b->sources = (int *)calloc(max_sources, (sizeof(int)));
  b->sinks = (int *)calloc(max_sinks, (sizeof(int)));
  return b;
}

/** Returns memory for a network (or a bipartite graph).
 *  @param network
 **/
void Network_Destroy(Network *network){

  int i;

  for(i=0; i<network->max_vertices; i++){
    if((&network->vertices[i])->in != NULL)
      free((&network->vertices[i])->in);
    if((&network->vertices[i])->out != NULL)
      free((&network->vertices[i])->out);
    if((&network->vertices[i])->predecessors != NULL)
      free((&network->vertices[i])->predecessors);
  }
  free(network->vertices);
  if(network->sources != NULL){
    free(network->sources);
  }
  if(network->sinks != NULL){
    free(network->sinks);
  }
  free(network->price);
  free(network->dual_grad);
  free(network->balance);
  for(i=0; i<network->max_vertices; i++){
    free(network->arcs[i]);
  }
  free(network->arcs);
  free(network);
}

/** Adds a vertex to a network.
 *  Assumes there is enough space in *vertices.
 *  @param vertex
 *  @param network
 **/
void Network_AddVertex(Network *network, Vertex *vertex){

  Vertex *v;

  assert(network != NULL);
  assert(network->no_vertices < network->max_vertices);
  v = &network->vertices[network->no_vertices++];
  v->index = Vertex_Index(vertex);
  v->max_in = Vertex_NoIn(vertex);
  v->max_out = Vertex_NoOut(vertex);
  v->marked = false;
  v->from = v->level = -1;
  v->cost = DBL_MAX;
  Vertex_Destroy(vertex);
}

/** Sets one index in the array of source indices.
 *  If the array is too small, it gets extended to twice its current size.
 *  @param network
 *  @param source_index
 **/
void Network_SetSource(Network *network, int source_index){

  assert(network != NULL);
  if(network->sources != NULL){
    if(network->no_sources >= network->max_sources){
      network->sources = (int*)realloc(network->sources,
                         ((network->max_sources*2)*sizeof(int)));
      network->max_sources *= 2;
    }
  }
  else{
    network->sources = (int*)calloc(1, sizeof(int));
    network->max_sources = 1;
  }
  network->sources[network->no_sources++] = source_index;
}
```

```
/** Sets one index in the array of sink indices.
 *  If the array is too small, it gets extended to twice its current size.
 *  @param network
 *  @param sink_index
 **/
void Network_SetSink(Network *network, int sink_index){

  assert(network != NULL);
  if(network->sinks != NULL){
    if(network->no_sinks >= network->max_sinks){
      network->sinks = (int*)realloc(network->sinks, ((network->max_sinks*
                                    2)*sizeof(int)));
      network->max_sinks *= 2;
    }
  }
  else{
    network->sinks = (int*)calloc(1, sizeof(int));
    network->max_sinks = 1;
  }
  network->sinks[network->no_sinks++] = sink_index;
}

/** Adds an arc to a network, if there is another arc present at the
 *  coordinates where this arc is to be added, the old arc remains.
 *  Sets the incoming and outgoing indexes for the involved vertices.
 *  Returns true if addition successful, otherwise false.
 *  @param  arc
 *  @param  network
 *  @return true or false
 **/
bool Network_AddArc(Network *network, Arc *arc){

  int from, to;
  Arc *a;
  QuadFun cost;

  assert(network != NULL);
  from = Arc_From(arc);
  to = Arc_To(arc);
  a = &network->arcs[from][to];
  cost = Arc_Cost(arc);
  if(Arc_Empty(a)){
    a->from = from;
    a->to = to;
    a->lower = Arc_Lower(arc);
    a->upper = Arc_Upper(arc);
    a->flow = Arc_Flow(arc);
    (&a->cost)->a = QuadFun_a(&cost);
    (&a->cost)->b = QuadFun_b(&cost);
    (&a->cost)->c = QuadFun_c(&cost);
    a->residual = Arc_Residual(arc);
    network->no_arcs++;
    Vertex_SetOut(&network->vertices[from], to);
    Vertex_SetIn(&network->vertices[to], from);
    Arc_Destroy(arc);
    return true;
  }
  else{
    Arc_Destroy(arc);
    return false;
  }
}

/** Gets the number of sources in the network.
 *  @param network
 **/
int Network_NoSources(Network *network){

  assert (network != NULL);
  return network->no_sources;
}

/** Gets the array with sources of the network.
 *  @param  network
 *  @return sources
 **/
int* Network_Sources(Network *network){

  assert (network != NULL);
  return network->sources;
}

/** Gets the number of sinks in the network.
```

```
 *  @param network
 **/
int Network_NoSinks(Network *network){

  assert (network != NULL);
  return network->no_sinks;
}

/** Gets the array with sinks of the network.
 *  @param  network
 *  @return sinks
 **/
int* Network_Sinks(Network *network){

  assert (network != NULL);
  return network->sinks;
}

/** Gets the number of vertices in the network.
 *  @param network
 **/
int Network_NoVertices(Network *network){

  assert (network != NULL);
  return network->no_vertices;
}

/** Gets the array with vertices of the network.
 *  @param  network
 *  @return vetices
 **/
Vertex* Network_Vertices(Network *network){

  assert (network != NULL);
  return network->vertices;
}

/** Gets the number of arcs of the network.
 *  @param  network
 *  @return no_arcs
 **/
int Network_NoArcs(Network *network){

  assert (network != NULL);
  return network->no_arcs;
}

/** Gets the matrix with arcs of the network.
 *  @param  network
 *  @return arcs
 **/
Arc** Network_Arcs(Network *network){

  assert (network != NULL);
  return network->arcs;
}

/** Returns true if an arc is present in the network at given coordinates,
 *  otherwise false.
 *  @param  network
 *  @param  row
 *  @param  col
 *  @retrun true or false
 **/
bool Network_ArcPresentAt(Network *network, int row, int col){

  return(!(Arc_Empty(&network->arcs[row][col])));
}

/** Constructs a copy of a network, resembeling the residual graph.
 *  If the total incoming flows of a vertex is more than the total
    outgoing flows,
 *  the vertex is considered a residual source. If opposite a residual sink.
 *  If super-true, a supersource and a supersink are added to the copy at
    position
 *  [max_vertices-2], [max_vertices-1] in vertices.
 *  Assumes that NU_SolveDualSub has been run for the setting of each
    vertex balance.
 *  @param  network
 *  @param  super
 *  @return copy
 **/
Network *Network_ResidualCopy(Network *network, bool super){
```

```
int i, super_source_index, super_sink_index;
long double curr_demand, total_source_demand, total_sink_demand,
            difference;
Network *copy;

assert(network != NULL);
super_source_index = super_sink_index = 0;
total_source_demand = total_sink_demand = 0;
if(!super){
  if((network->max_sources + network->max_sinks) ==
      network->max_vertices){
    copy = Network_CreateB(network->max_sources, network->max_sinks);
  }
  else{
    copy = Network_Create(network->max_vertices);
  }
}
else{
  copy = Network_Create(network->max_vertices+2);
}
Network_ArcsVerticesCopy(copy, network);
for(i=0; i<network->no_vertices; i++){
  curr_demand = network->balance[Vertex_Index(&network->vertices[i])];
  if(curr_demand != Vertex_Demand(&network->vertices[i])){
    Vertex_SetDemand(&copy->vertices[i],
                     (Vertex_Demand(&network->vertices[i]) -
                     curr_demand));
    if(Vertex_IsSource(&copy->vertices[i])){
      Network_SetSource(copy, Vertex_Index(&copy->vertices[i]));
      total_source_demand += Vertex_Demand(&copy->vertices[i]);
    }
    if(Vertex_IsSink(&copy->vertices[i])){
      Network_SetSink(copy, Vertex_Index(&copy->vertices[i]));
      total_sink_demand += Vertex_Demand(&copy->vertices[i]);
    }
  }
}
if(super){
  //create a supersource and a supersink
  super_source_index = network->max_vertices;
  super_sink_index = network->max_vertices+1;
  Network_AddVertex(copy, Vertex_Create(super_source_index,
                    0, network->max_vertices));
  Network_SetSource(copy, super_source_index);
  Network_AddVertex(copy, Vertex_Create(super_sink_index,
                    network->max_vertices, 0));
  Network_SetSink(copy, super_sink_index);
  // create an arc between the supersource and each source
  for(i=0; i<copy->no_sources-1; i++){
    Network_AddArc(copy, Arc_Create(super_source_index,
                   Vertex_Index(&copy->vertices[copy->sources[i]]),
                   0, Vertex_Demand(&copy->vertices[copy->sources[i]]),
                   (QuadFun_Create(0, 0))));
    Vertex_SetDemand(&copy->vertices[super_source_index],
    (Vertex_Demand(&copy->vertices[super_source_index]) +
    Vertex_Demand(&copy->vertices[copy->sources[i]])));
  }
  // create an arc between each sink and the supersink
  for(i=0; i<copy->no_sinks-1; i++){
    Network_AddArc(copy, Arc_Create(
    Vertex_Index(&copy->vertices[copy->sinks[i]]),
    super_sink_index, 0, -
    Vertex_Demand(&copy->vertices[copy->sinks[i]]),
    (QuadFun_Create(0, 0)) ));
    Vertex_SetDemand(&copy->vertices[super_sink_index],
    (Vertex_Demand(&copy->vertices[super_sink_index]) +
    Vertex_Demand(&copy->vertices[copy->sinks[i]])));
  }
}
// balance network if total_source_demand != total_sink_demand and
   difference not too small
//printf("st %.16Lf,ss %.16Lf\n ",
          Vertex_Demand(&copy->vertices[super_sink_index]),
          Vertex_Demand(&copy->vertices[super_source_index]) );
if(total_source_demand < (-total_sink_demand)){
  difference = (-total_sink_demand)-total_source_demand;
  if(difference > CORR_EPSILON)
    Vertex_SetDemand(&copy->vertices[super_sink_index],
    (Vertex_Demand(&copy->vertices[super_sink_index])+difference));
}
else if (total_source_demand > (-total_sink_demand)){
  difference = total_source_demand+total_sink_demand;
```

```
    if(difference > CORR_EPSILON)
      Vertex_SetDemand(&copy->vertices[super_source_index],
      (Vertex_Demand(&copy->vertices[super_source_index])-difference));
  }
  return copy;
}

/** Constructs an ald of a network based on the information in each
    vertex's
 *  predecessor vector. The ald contains necessary vertex and arc
    information need
 *  in the A_MFALD algorithm only.
 *  No arrays containing sources or sinks, nor any other specifics
    connected
 *  to the network are copied.
 *  Assumes that either A_BS or A_DS has been run.
 *  @param  network network containing information to create an ald
 *  @return ald     network holding the resulting ald
 **/
Network *Network_ALDCopy(Network *network){

  int i, j, *pre_p;
  Vertex *old, *new, *source, *sink;
  Network *ald;

  ald = Network_Create(network->max_vertices);
  ald->no_vertices = network->no_vertices;
 // copy the information of the vertices that are marked
  for(i=0; i<network->no_vertices; i++){
    if(Vertex_Marked(&network->vertices[i])){
      (&ald->vertices[i])->index = Vertex_Index(&network->vertices[i]);
      (&ald->vertices[i])->marked = true;
      (&ald->vertices[i])->level = Vertex_Level(&network->vertices[i]);
      (&ald->vertices[i])->capacity = Vertex_Cap(&network->vertices[i]);
      }
  }
  // copy arcs that are included in the predecessor array of a vertex if
  // the current arc connects to a marked vertex and the arc is not
     saturated,
  // adjust in and out for the involved vertices
  for(i=0; i<network->no_vertices; i++){
    if(Vertex_Marked(&network->vertices[i])){
      new = &ald->vertices[i];
      old = &network->vertices[i];
      if(Vertex_NoPre(old) > 0){
        pre_p = Vertex_Predecessors(old);
        for(j=0; j<Vertex_NoPre(old); j++){
          if(Vertex_Marked(&network->vertices[*pre_p]) &&
            (!Arc_Saturated(&network->arcs[*pre_p][i]))){
            Vertex_SetPre(new, *pre_p);
            // the arc from Vertex *pre_p to new should be copied
            ald->arcs[*pre_p][i] = network->arcs[*pre_p][i];
            Vertex_SetIn(new, *pre_p);
            Vertex_SetOut(&ald->vertices[*pre_p],i);
          }
          pre_p++;
        }
      }
    }
  }
  source = &ald->vertices[ald->no_vertices-2];
  source->demand = (&network->vertices[network->no_vertices-2])->demand;
  source->no_pre = source->no_in = 0;
  sink = &ald->vertices[ald->no_vertices-1];
  sink->demand = (&network->vertices[network->no_vertices-1])->demand;
  sink->no_out = 0;
  return(ald);
}

/** Constructs a copy of a network, resembeling the residual graph used
    in the
 *  polynomial maxflow algorithm by Migdalas.
 *  If the capacity of a vertex is zero, the vertex is removed along with
 *  its incoming and outgoing arcs.
 *  If an arc has a flow equal to its upper capacity, the arc is removed.
 *  Assumes that the vertices with nonzero capacity are marked.
 *  @param  network
 *  @return copy
 **/
Network *Network_MFALDCopy(Network *network){

  int i, j, *out_p, *pre_p;
  Network *copy;
```

```
  Vertex *old, *new;
  //Arc *arc;

  copy = Network_Create(network->max_vertices);
  copy->no_vertices = network->no_vertices;
  // copy the vertices with a capacity > OPT_EPSILON
  for(i=0; i<network->no_vertices; i++){
    if(Vertex_Marked(&network->vertices[i])){
      (&copy->vertices[i])->index = Vertex_Index(&network->vertices[i]);
      (&copy->vertices[i])->marked = true;
      (&copy->vertices[i])->level = Vertex_Level(&network->vertices[i]);
      (&copy->vertices[i])->capacity = Vertex_Cap(&network->vertices[i]);
    }
  }
  (&copy->vertices[copy->no_vertices-2])->demand =
  (&network->vertices[network->no_vertices-2])->demand;
  (&copy->vertices[copy->no_vertices-1])->demand =
  (&network->vertices[network->no_vertices-1])->demand;
  // 1) copy the arcs that has flow < upper capacity of the current arc,
  //    if the arc does not lead to an umarked vertex
  //    adjust in and out for the involved vertices
  // 2) update the predecessor array of each vertex
  for(i=0; i<copy->no_vertices; i++){
    if(Vertex_Marked(&network->vertices[i])){
      new = &copy->vertices[i];
      old = &network->vertices[i];
      if(Vertex_NoPre(old) != 0){
        pre_p = Vertex_Predecessors(old);
        for(j=0; j<Vertex_NoPre(old); j++){
          if((Vertex_Marked(&network->vertices[*pre_p])== true) &&
             (!Arc_Saturated(&network->arcs[*pre_p][i])))
            Vertex_SetPre(new, *pre_p);
          pre_p++;
        }
      }
      if(Vertex_NoOut(old) > 0){
        out_p = Vertex_Out(old);
        for(j=0; j<Vertex_NoOut(old); j++){
          if((Vertex_Marked(&network->vertices[*out_p])==true) &&
             (!Arc_Saturated(&network->arcs[i][*out_p]))){
            copy->arcs[i][*out_p] = network->arcs[i][*out_p];
            Vertex_SetOut(new, *out_p);
            Vertex_SetIn(&copy->vertices[*out_p], i);
          }
          out_p++;
        }
      }
    }
  }
  return(copy);
}

/** String representation of a network.
 *  @param network
 **/
void Network_Print(Network *network){

  int i, j, k, noOut, *ip;
  Vertex *vp;

  assert(network != NULL);
  vp = network->vertices;
  for(i=0; i<network->no_vertices; i++){
    Vertex_Print(vp);
    printf("%s\n", "--------------------------------");
    noOut = Vertex_NoOut(vp);
    if(noOut != 0){
      ip = Vertex_Out(vp);
      for(k=0; k<noOut; k++){
           j = *ip++;
           Arc_Print(&(network->arcs[Vertex_Index(vp)][j]));
           printf("%s", "\n");
      }
    }
    printf("%s\n", "--------------------------------");
    vp++;
  }
}

/*****************************************************************************
* PRIVATE FUNCTIONS
*****************************************************************************
```

```
/** Copies the arcs and the vertices in the original network to the copy
    network.
 *  If arc (ij) in original has flow, it is copied as is to arc (ij)
 *  and to arc (ji) with flow 0 and upper(ji)-flow(ij) in copy.
 *  Otherwise, arc (ij) in original is copied as is to arc (ij) in copy.
 *  @param copy     the network to hold the copy
 *  @param original the matrix that holds the elements to be copied
 **/
void Network_ArcsVerticesCopy(Network *copy, Network *original){

  int i, j, k;
  Vertex *c, *o;

  for(i=0; i<Network_NoVertices(original); i++){
    c = &Network_Vertices(copy)[i];
    o = &Network_Vertices(original)[i];
    c->index = o->index;
    c->price = o->price;
    c->from = o->from;
    c->marked = o->marked;
    c->cost = o->cost;
    copy->no_vertices++;
  }
  for(i=0; i<Network_NoVertices(original); i++){
    if(Vertex_NoOut(&Network_Vertices(original)[i]) != 0){
      for(k=0; k<Vertex_NoOut(&Network_Vertices(original)[i]); k++){
        j = Vertex_Out(&Network_Vertices(original)[i])[k];
        if(Arc_HasFlow(&Network_Arcs(original)[i][j])){
          Network_Arcs(copy)[i][j] = Network_Arcs(original)[i][j];
          Network_SetResidualArc(copy->arcs, original->arcs,
                                 &copy->vertices[j], &copy->vertices[i],
                                 j, i);
        }
        else{
          Network_Arcs(copy)[i][j] = Network_Arcs(original)[i][j];
        }
        Vertex_SetOut(&Network_Vertices(copy)[i],j);
        Vertex_SetIn(&Network_Vertices(copy)[j],i);
      }
    }
  }

}

/** Sets an copy of the arc at position (j, i) at position (i, j)
 *  in a copy of the networks matrix of arcs. Updates the involved
    vertices.
 *  @param copy     the copy to hold the arcs of the copied network
 *  @param original the matrix of arcs in the original network
 *  @param v_i      the vertex at position i in copy of network
 *  @param v_j      the vertex at position j in copy of network
 *  @param i
 *  @param j
 **/
void Network_SetResidualArc(Arc **copy, Arc **original, Vertex *v_i,
                            Vertex *v_j, int i, int j){

  copy[i][j] = original[j][i];
  Arc_SetFrom(&copy[i][j], i);
  Arc_SetTo(&copy[i][j], j);
  Arc_SetResidualCost(&copy[i][j], &original[j][i]);
  Vertex_SetIn(v_j, i);
  Vertex_SetOut(v_i, j);
  Arc_SetFlow(&copy[i][j], 0);
  Arc_SetLower (&copy[i][j], 0);
  Arc_SetUpper(&copy[i][j], Arc_Flow(&original[j][i]));
  Arc_SetResidual(&copy[i][j], true);
  Arc_SetMarked(&copy[i][j], true);
 }
```

A.3.3 NetworkUtil.c

```
#include "NetworkUtil.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <float.h>
#include "Vertex.h"
#include "Arc.h"
#include "Misc.h"
#include "QuadFun.h"
#include "Algorithms.h"

/*********************************************************************
* PUBLIC FUNCTIONS
*********************************************************************/

/** Returns the norm of the squared demand of the vertices in the network,
    i.e. ||b||.
 *  Assumes Network_SolveDualSub has been run.
 *  @param  network
 *  @return demand_norm
 **/
long double NetworkUtil_DemandNormSquare(Network *network){

  assert(network != NULL);
  return network->demand_norm_square;
}

/** Returns an array with the prices of the networks vertices.
 *  Assumes Network_SolveDualSub has been run.
 *  @param  network
 *  @return price
 **/
long double* NetworkUtil_Price(Network *network){

  assert(network!=NULL);
  return network->price;
}

/** Sets the prices of the networks vertices to the values in new_price.
 *  Assumes that price holds same # elements as there are vertices in
    network,
 *  and that Network_SolveDualSub has been run.
 *  @param network
 *  @param new_price
 **/
void NetworkUtil_SetPrice(Network *network, long double *new_price){

  int i, no_vertices;
  Vertex *vp;
  long double *fp;

  assert(network!=NULL);
  no_vertices = network->no_vertices;
  vp = network->vertices;
  fp = new_price;
  for(i=0; i<no_vertices; i++){
    Vertex_SetPrice(vp++, *fp);
    network->price[i] = *fp++;
  }
}

/** Adds scalar*vector to the current prices of the networks vertices.
 *  Assumes that vector holds same # elements as there are vertices in
    network,
 *  and that Network_SolveDualSub has been run.
 *  @param network
 *  @param scalar
 *  @param vector
 **/
void NetworkUtil_UpdatePrice(Network *network, double scalar, long double *vector){

  int i, no_vertices;
  Vertex *vp;
  long double new_price;

  assert(network!=NULL);
  no_vertices = network->no_vertices;
  vp = network->vertices;
  for(i=0; i<no_vertices; i++){
    new_price = Vertex_Price(vp)+(scalar*vector[i]);
    Vertex_SetPrice(vp++, new_price);
```

```
    network->price[i] = new_price;
  }
}

/** Returns an array with the dual gradient of the network.
 *  Assumes Network_SolveDualSub has been run.
 *  @param  network
 *  @return gradient
 **/
long double* NetworkUtil_DualGrad(Network *network){

  assert(network!=NULL);
  return network->dual_grad;
}

/** Returns an array with the balance of the network.
 *  Assumes Network_SolveDualSub has been run.
 *  @param  network
 *  @return balance
 **/
long double* NetworkUtil_Balance(Network *network){

  assert(network!=NULL);
  return network->balance;
}

/** Returns the value of the squared norm of the dual gradient of the
    network.
 *  Assumes Network_SolveDualSub has been run.
 *  @param  network
 *  @return dual_grad_norm
 **/
long double NetworkUtil_DualGradNormSquare(Network *network){

  assert(network!=NULL);
  return network->dual_grad_norm_square;
}

/** Sets the pathlength of current path from source to sink in residual
    network
 *  @param network
 *  @param path_length
 **/
void NetworkUtil_SetPathLength(Network *network, int path_length){

  assert(network != NULL);
  network->path_length = path_length;
}

/** Gets the pathlength of current path from source to sink in residual network
 *  @param  network
 *  @return path_length
 **/
int NetworkUtil_PathLength(Network *network){

  assert(network != NULL);
  return(network->path_length);
}

/** Sets the max_push of current path from source to sink in residual
    network
 * @param network
 * @param max_push
 **/
void NetworkUtil_SetMaxPush(Network *network, long double max_push){

  assert(network != NULL);
  network->max_push = max_push;
}

/** Gets the value for max_push of current path from source to sink in
    residual network
 * @param  network
 * @return max_push
 **/
long double NetworkUtil_MaxPush(Network *network){

  assert(network != NULL);
  return(network->max_push);
}

/** Sets the index of the vertex with minimum capacity in vertices
 * @param network
```

```
 * @param min_index
 **/
void NetworkUtil_SetMinIndex(Network *network, int min_index){
  assert(network != NULL);
  network->min_index = min_index;
}

/** Gets the index of the vertex with minimum capacity in vertices
 * @param network
 * @return min_index
 **/
int NetworkUtil_MinIndex(Network *network){
  assert(network != NULL);
  return(network->min_index);
}

/** Gets the number of marked arcs in arcs.
 * @param network
 * @return no_marked
 **/
int NetworkUtil_NoMarked(Network *network){
  assert(network != NULL);
  return(network->no_marked);
}

/** Gets the current length of the alpha array
 * @param network
 * @return no_alpha
 **/
int NetworkUtil_NoAlpha(Network *network){
  assert(network != NULL);
  return(network->no_alpha);
}

/** Unmarkes all the vertices in the network.
 *  Sets all from values to -1, and all cost values to DBL_MAX.
 *  @param network
 **/
void NetworkUtil_UnMark(Network *network){

  int i, no_vertices;

  assert(network!=NULL);
  no_vertices = network->no_vertices;
  for(i=0; i<no_vertices; i++){
    Vertex_SetMarked(&network->vertices[i], false);
    Vertex_SetFrom(&network->vertices[i], -1);
    Vertex_SetCost(&network->vertices[i], DBL_MAX);
    Vertex_SetLevel(&network->vertices[i], -1);
    (&network->vertices[i])->predecessors = NULL;
    (&network->vertices[i])->no_pre = 0;
  }
}

/** Returns the primal objective function value evaluated at the current
 *  flow values, i.e. sum f_ij(x_ij)
 *  @param  network
 *  @return value
 **/
long double NetworkUtil_PrimalValue(Network *network){

  long double sum;
  int i, j;
  Vertex *vertex;
  Arc *arc;

  assert(network != NULL);
  sum = 0;
  for(i=0; i<Network_NoVertices(network); i++){
    vertex = &Network_Vertices(network)[i];
    if(Vertex_NoOut(vertex) != 0){
      for(j=0; j<Vertex_NoOut(vertex); j++){
        arc = &(Network_Arcs(network)[i][Vertex_Out(vertex)[j]]);
        sum += Arc_CurrCost(arc);
      }
    }
  }
  return sum;
}

/** Returns delta_flow for an vertex.
 *  delta_flow is calculated as sum (u_ij-flow_ij) for all incoming arcs
 *  if in=true.
```

```
 *  If if-false, delta_flow is calculated for all outgoing arcs.
 *  @param  network
 *  @param  vertex
 *  @param  in
 *  @return delta_flow
 */
long double NetworkUtil_DeltaFlow(Network *network, Vertex *vertex,
                                  bool in){

  int i;
  Arc *arc;
  long double sum = 0;

  if(!in){
    if(Vertex_NoOut(vertex) > 0){
      for(i=0; i<Vertex_NoOut(vertex); i++){
        arc = &(Network_Arcs(network)[Vertex_Index(vertex)]
                                     [Vertex_Out(vertex)[i]]);
          if(Vertex_Marked(&Network_Vertices(network)[Vertex_Out(vertex)[i]])
                          && !Arc_Saturated(arc))
            sum += Arc_Upper(arc) - Arc_Flow(arc);
      }
    }
  }
  else{
    if(Vertex_NoIn(vertex) > 0){
      for(i=0; i<Vertex_NoIn(vertex); i++){
        arc = &(Network_Arcs(network)[Vertex_In(vertex)[i]]
                                     [Vertex_Index(vertex)]);
        if(Vertex_Marked(&Network_Vertices(network)[Vertex_In(vertex)[i]])
                         && !Arc_Saturated(arc))
          sum += Arc_Upper(arc) - Arc_Flow(arc);
      }
    }
  }
  return(sum);
}

/** Solves the dual subproblem, and returns the value of the dual function.
 *  If current-true, uses the current price values of the vertices
 *  otherwise uses the prices provided in new_price.
 *  Sets the flows that corresponds to the solution for each arc,
 *  the gradient of the dual function, the norm of the demand,
 *  and the norm of the dual gradient.
 *  Assumes new_price holds same # elements as there are vertices in
    network.
 *  @param  network
 *  @param  new_price
 *  @param  current
 *  @return dual_value
 **/
long double NetworkUtil_SolveDualSub(Network *network, long double
                                     *new_price, bool current){

  int i, j, k;
  Vertex *vp;
  Arc *arc;
  QuadFun cost;
  long double curr_flow, dual_value, sumout, sumin, imbalance,
              demand_norm_square, dual_grad_norm_square;

  assert(network!=NULL);
  curr_flow = dual_value = sumout = sumin = imbalance =
           demand_norm_square = dual_grad_norm_square = 0;
  for(i=0; i<network->no_vertices; i++){
    vp = &network->vertices[i];
    // sets or gets flows for outgoing arcs of vertex i
    if(Vertex_NoOut(vp) != 0){
      for(k=0; k<Vertex_NoOut(vp); k++){
        j = Vertex_Out(vp)[k];
        arc = &(network->arcs[i][j]);
        cost = Arc_Cost(arc);
        if(j>=i){
          if(current){
            if(!Arc_Marked(arc))
              curr_flow = NetworkUtil_CalcFlow(arc, Vertex_Price(vp),
                                  Vertex_Price(&network->vertices[j]));
               else{
               curr_flow = Algorithms_Mid(Arc_Lower(arc),
                                  ( -(QuadFun_a(&cost)+ Vertex_Price(vp),
                                   Vertex_Price(&network->vertices[j])) /
                                  (2*QuadFun_b(&cost)) ), Arc_Upper(arc));
               Arc_SetMarked(arc, false);}
```

```
        }
        else
          curr_flow = NetworkUtil_CalcFlow(arc,
                          new_price[Vertex_Index(vp)],
                          new_price[Vertex_Index(&network->vertices[j])]);
        Arc_SetFlow(arc, curr_flow);
      }
      else
        curr_flow = Arc_Flow(arc);
      if(current){
        dual_value += QuadFun_DualValue(&cost, curr_flow,
                                        Vertex_Price(vp),
                                        Vertex_Price(&network->vertices[j]));
      }
      else{
        dual_value += QuadFun_DualValue(&cost, curr_flow,
                          new_price[Vertex_Index(vp)],
                          new_price[Vertex_Index(&network->vertices[j])]);
      }
      sumout += curr_flow;
    }
  }
  // set flows for incoming arcs of vertex i
  if(Vertex_NoIn(vp) != 0){
    for(k=0; k<Vertex_NoIn(vp); k++){
      j = Vertex_In(vp)[k];
      arc = &(network->arcs[j][i]);
      if(j>i){
        if(current){
          curr_flow = NetworkUtil_CalcFlow(arc,
                                        Vertex_Price(&network->vertices[j]),
                                        Vertex_Price(vp));
        }
        else{
          curr_flow = NetworkUtil_CalcFlow(arc,
                           new_price[Vertex_Index(&network->vertices[j])],
                           new_price[Vertex_Index(vp)]);
        }
        Arc_SetFlow(arc, curr_flow);
      }
      else{
        curr_flow = Arc_Flow(arc);
      }
      sumin += curr_flow;
    }
  }
  if(current){
    dual_value -= Vertex_Price(vp)*Vertex_Demand(vp);
  }
  else{
    dual_value -= new_price[Vertex_Index(vp)]*Vertex_Demand(vp);
  }
  imbalance = sumout - sumin - Vertex_Demand(vp);
  network->dual_grad[Vertex_Index(vp)] = imbalance;
  network->balance[Vertex_Index(vp)] = sumout - sumin;
  dual_grad_norm_square += imbalance*imbalance;
  sumout = sumin = 0;
  demand_norm_square += Vertex_Demand(vp)*Vertex_Demand(vp);
 }
 network->dual_grad_norm_square = dual_grad_norm_square;
 network->demand_norm_square = demand_norm_square;
 return dual_value;
}

/** Calculates all stepsizes alpha according to the solution of KKT
 *  for the dual line search problem (max w(v + alpha*d)) of the network.
 *  Assumes Network_SolveDualSub has been run, and that all arcs are
    unmarked.
 *  @param network
 *  @param alpha   array to hold the stepsizes
 *  @param d       array of directions d
 *  @param first   true if the alphas should be set in array,
                   false if in arc
 **/
void NetworkUtil_CalcAlphas(Network *network, Alpha *alpha, long double
                            *d, bool first){

  Vertex *vp;
  Arc *arc;
  int i, j, k, m;
  double a_ij, b_ij, v_i, v_j, d_i, d_j, value_u, value_l;
  QuadFun cost;
```

```
  m  = 0;
  (&alpha[m++])->value = 0;
  network->no_marked = 0;
  for(i=0; i<network->no_vertices; i++){
    vp = &network->vertices[i];
    v_i = Vertex_Price(vp);
    d_i = d[i];
    if(Vertex_NoOut(vp) != 0){
      for(k=0; k<Vertex_NoOut(vp); k++){
        j = Vertex_Out(vp)[k];
        arc = &(network->arcs[i][j]);
        cost = Arc_Cost(arc);
        a_ij = QuadFun_a(&cost);
        b_ij = QuadFun_b(&cost);
        v_j = Vertex_Price(&network->vertices[j]);
        d_j = d[j];
        if((d_i = d_j) == 0){
          Arc_SetMarked(arc, true);
          Arc_SetAlphaU(arc, -1);
          Arc_SetAlphaL(arc, -1);
          network->no_marked++;
        }
        else{
          value_u = ((-(a_ij + v_i - v_j + (2 * b_ij * Arc_Upper(arc)))) /
                     (d_i - d_j));
          value_l = ((-(a_ij + v_i - v_j + (2 * b_ij * Arc_Lower(arc)))) /
                     (d_i - d_j));
          // the values should be set to alpha directly
          if(first){
            (&alpha[m])->value = value_u;
            (&alpha[m])->i = i;
            (&alpha[m])->j = j;
            (&alpha[m++])->lower = false;
            (&alpha[m])->value = value_l;
            (&alpha[m])->i = i;
            (&alpha[m])->j = j;
            (&alpha[m++])->lower = true;
          }
          // if the current arc had alpha_l--alpha_u--1, in previous
             iteration,
          // i.e. d_i-d_j--0, value_l and value-u should be included in
             alpha
          // at position no_alpha_old+1, no_alpha_old+2
          else{
            if(Arc_AlphaU(arc) == -1){
              (&alpha[network->no_alpha])->value = value_u;
              (&alpha[network->no_alpha])->i = i;
              (&alpha[network->no_alpha])->j = j;
              (&alpha[network->no_alpha++])->lower = false;
              (&alpha[network->no_alpha])->value = value_l;
              (&alpha[network->no_alpha])->i = i;
              (&alpha[network->no_alpha])->j = j;
              (&alpha[network->no_alpha++])->lower = true;
            }
          }
          // the values should be set to the involved arc directly
          Arc_SetAlphaU(arc, value_u);
          Arc_SetAlphaL(arc, value_l);
        }
      }
    }
  }
  network->no_alpha = (network->no_vertices*2) + 1 - (network->no_marked*2);
}

/*************************************************************************
* PRIVATE FUNCTIONS
*************************************************************************

/** Returns a flow value for arc.
 *  If dual_diff for arc evaluated at lower > 0, returns lower.
 *  If dual_diff for arc evaluated at upper < 0, returns upper.
 *  Otherwise returns the value found when dual_diff = 0.
 *  @param  arc        the arc under consideration
 *  @param  v_i        the price for the arcs tail
 *  @param  v_j        the price for the arcs head
 *  @return flow value
 **/
long double NetworkUtil_CalcFlow(Arc *arc, long double v_i, long double v_j){

  QuadFun *dual_diff, cost;
  long double value;
```

```
  assert(arc != NULL);
  cost = Arc_Cost(arc);
  dual_diff = QuadFun_Create (0,0);
  QuadFun_DualDiff(&cost, dual_diff, v_i, v_j);
  // dual_diff > 0?
  if(NetworkUtil_CheckLimits(dual_diff, Arc_Lower(arc), true))
    value = Arc_Lower(arc);
  // dual_diff < 0?
  else if(NetworkUtil_CheckLimits(dual_diff, Arc_Upper(arc), false))
    value = Arc_Upper(arc);
  // dual_diff = 0
  else
    value = -(QuadFun_a(&cost) + v_i - v_j) / (2*QuadFun_b(&cost));
  QuadFun_Destroy(dual_diff);
  return(value);
}

/** A check for an arcs dual functions differential value at its limits.
 *  If lower-true, the differential at the lower limit is checked, and
 *  true is returned if the differential > 0.
 *  If lower-false, the differential at the upper limit is checked, and
 *  true is returned if the differential < 0.
 *  @param  dual_diff
 *  @param  x         lower/upper value of flow
 *  @return true or false
 **/
bool NetworkUtil_CheckLimits(QuadFun *dual_diff, long double x, bool lower){

  bool ok = false;

  if(lower){
    if(QuadFun_Value(dual_diff, x)>0)
      ok = true;
  }
  else{
    if(QuadFun_Value(dual_diff, x)<0)
      ok = true;
  }
  return ok;
}
```

A.3.4 QuadFun.c

```
#include "QuadFun.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/*typedef struct{
  double c; // constant coefficient
  double a; // linear coefficient
  double b; // quadratic coefficient
  }QuadFun;*/

/*****************************************************************************
* PUBLIC FUNCTIONS
*****************************************************************************/

/* Allocates memory for a QuadFun.
 * @param  a       linear coefficient
 * @param  b       quadratic coefficient
 * @return quadfun
 */
QuadFun* QuadFun_Create(double a, double b){

  QuadFun *fun;

  fun = (QuadFun *)calloc(1, sizeof(QuadFun));
  fun->a = a;
  fun->b = b;
  fun->c = 0;
  return fun;
}

/* Returns memory for a QuadFun.
 * @param fun
 */
void QuadFun_Destroy(QuadFun *fun){

  free(fun);
}

/* Returns the constant coefficient.
 * @param  fun
 * @return c
 */
double QuadFun_c(QuadFun *fun){

  assert(fun != NULL);
  return fun->c;
}

/* Returns the linear coefficient.
 * @param  fun
 * @return a
 */
double QuadFun_a(QuadFun *fun){

  assert(fun != NULL);
  return fun->a;
}

/* Returns the quadratic coefficient.
 * @param  fun
 * @return b
 */
double QuadFun_b(QuadFun *fun){

  assert(fun != NULL);
  return fun->b;
}

/* Returns the differential of the quadratic function,
 * i.e. f'_ij(x_ij) = a_ij + 2*b_ij*x_ij.
 * @param fun
 * @param dfun
 */
void QuadFun_Diff(QuadFun *fun, QuadFun *dfun){

  dfun->a = 2*QuadFun_b(fun);
  dfun->c = fun->a;
}

/* Returns the value of the quadratic function evaluated at x_ij, i.e.
   f_ij(x_ij).
```

```
 * @param  fun
 * @param  x_ij
 * @return value
 */
long double QuadFun_Value(QuadFun*fun, long double x_ij){

  assert(fun != NULL);
  return ((fun->c)+((fun->a)*x_ij)+((fun->b)*x_ij*x_ij));
}

/* Returns the value for the of x_ij depending part of the dual function
   evaluated at x_ij,
 * i.e. f_ij(x_ij)+(v_i-v_j)*x_ij.
 * @param  fun
 * @param  x_ij
 * @param  v_i price of vertex at arcs tail
 * @param  v_j price of vertex at arcs head
 * @return value
 */
long double QuadFun_DualValue(QuadFun*fun, long double x_ij, long double
                              v_i, long double v_j){

  assert(fun != NULL);
  return (QuadFun_Value(fun, x_ij)+((v_i-v_j)*x_ij));
  }

/* Sets the differential in x_ij of the dual function,
 * i.e. 2*b_ij*x_ij + (a_ij + v_i - v_j).
 * @param  fun       original function
 * @param  dual_diff differential of fun in x_ij
 * @param  v_i       price of vertex at arcs tail
 * @param  v_j       price of vertex at arcs head
 */
void QuadFun_DualDiff(QuadFun *fun, QuadFun *dual_diff, long double v_i,
                      long double v_j){

  assert(fun != NULL);
  assert(dual_diff != NULL);
  dual_diff->a = 2 * (QuadFun_b(fun));
  dual_diff->b = 0;
  dual_diff->c = QuadFun_a(fun) + v_i - v_j;
}

/* Returns true if the quadfun holds the "empty function"
 * otherwise false.
 * @param  fun
 * @return true or false
 */
bool QuadFun_Empty(QuadFun*fun){

  assert(fun != NULL);
  return ((fun->a--0) && (fun->b--0));
}

/* String representation of a quadfun.
 * @param fun
 */
void QuadFun_Print(QuadFun *fun){

  assert(fun != NULL);
  printf("c = %f a = %f  b = %f\n", fun->c, fun->a, fun->b);
}
```

A.3.5 Vertex.c

```
/* Body for Vertex */

#include "Vertex.h"
#include "Misc.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <float.h>

/*typedef struct{
  int index;            // index of vertex
  long double demand;   // demand for vertex, i.e. b_i
                        // demand>0 -> source,
                        // demand<0 -> sink,
                        // demand-0 -> transports
  int *in;              // array with indices of those vertices connected
                           to vertex by incoming arcs
  int max_in;           // maximum size of *in
  int no_in;            // current size of *in
  int *out;             // array with indices of those vertices connected
                           to vertex by outgoing arcs
  int max_out;          // maximum size of *out
  int no_out;           // current size of *out
  long double price;    // dual variable for vertex, i.e. v_i
   // for graph searching reasons...
  int from;             // index of vertex preceding this in a path search
  bool marked;          // true if reached in a path search, false other-
                           wise
                        // true if included in MFALDCopy, false otherwise
  long double cost;     // current cost to reach this vertex in a shortest
                           path search
  int level;            // number of level at which this vertex can be
                           found in a BS search
  int *predecessors;    // indicies of verticies preceding this in a BS
                           search
  int max_pre;          // maximum size of *predecessors
  int no_pre;           // current size of *predecessors
  long double capacity; // capacity for vertex used in the MaxFlowALD
                           algorithm
  }Vertex; */

/************************************************************************
* PUBLIC FUNCTIONS
************************************************************************/

/** Allocates memory for a vertex.
 *  @param  index  the index of the vertex
 *  @param  demand the demand of the vertex
 *  @return vertex
 **/
Vertex *Vertex_Create(int index, int max_in, int max_out){

  Vertex *v;

  v = (Vertex *)calloc(1, sizeof(Vertex));
  v->index = index;
  v->max_in = max_in;
  if(max_in > 0)
    v->in = (int *)calloc(max_in, (sizeof(int)));
  else
    v->in = NULL;
  v->max_out = max_out;
  if(max_out > 0)
    v->out = (int *)calloc(max_out, (sizeof(int)));
  else
    v->out = NULL;
  v->from = v->level = -1;
  v->predecessors = NULL;
  v->marked = false;
  v->cost = DBL_MAX; // 0 for testing purposes, DBL_MAX otherwise
  return v;
}

/** Returns memory for a vertex.
 *  @param vertex
 **/
void Vertex_Destroy(Vertex *vertex){

  if(vertex->in != NULL)
    free(vertex->in);
  if(vertex->out != NULL)
    free(vertex->out);
```

```
  if(vertex->predecessors != NULL)
    free(vertex->predecessors);
  free(vertex);
}

/** Gets the index for a vertex.
 *  @param  vertex
 *  @return index
 **/
int Vertex_Index(Vertex *vertex){

  assert(vertex != NULL);
  return vertex->index;
}

/** Sets the demand for a vertex.
 *  @param vertex
 *  @param demand
 **/
void Vertex_SetDemand(Vertex *vertex, long double demand){

   assert(vertex != NULL);
   vertex->demand = demand;
}

/** Gets the demand for a vertex.
 *  @param  vertex
 *  @return demand
 **/
long double Vertex_Demand(Vertex *vertex){

  assert(vertex != NULL);
  return vertex->demand;
}

/** Changes the demand for a vertex by a given amount.
 *  Assumes that the amount is is within limits, i.e.
 *  if the vertex is a source, demand stays >=0.
 *  if the vertex is a sink, demand stays <=0.
 *  @param vertex
 *  @param amount
 **/
void Vertex_ChangeDemand(Vertex *vertex, long double amount){

  assert(vertex != NULL);
  if((vertex->demand) != 0){
    vertex->demand = Vertex_ValidChange((vertex->demand), amount);
  }
}

/** Returns true if the vertex is a source,
 *  otherwise false.
 *  @param  vertex
 *  @return true or false
 **/
bool Vertex_IsSource(Vertex *vertex){

  assert(vertex != NULL);
  return(Vertex_Demand(vertex)>0);
}

/** Returns true if the vertex is a sink,
 *  otherwise false.
 *  @param  vertex
 *  @return true or false
 **/
bool Vertex_IsSink(Vertex *vertex){

  assert(vertex != NULL);
  return(Vertex_Demand(vertex)<0);
}

/** Sets one index, assumes index not already present, in the array with
 *  indices of those vertices connected to a vertex by incoming arcs.
 *  If the array is too small, it gets extended to twice its current size.
 *  @param in
 **/
void Vertex_SetIn(Vertex *vertex, int in){

  assert(vertex != NULL);
  if(vertex->in != NULL){
    if(vertex->no_in >= vertex->max_in){
      vertex->in = (int*)realloc(vertex->in, ((vertex->max_in*2)*sizeof(int)));
```

```
      vertex->max_in *= 2;
    }
  }
  else{
    vertex->in = (int*)calloc(1, sizeof(int));
    vertex->max_in = 1;
  }
  vertex->in[(vertex->no_in)++] = in;
}

/** Gets the array with indices of those vertices connected to a vertex
 *  by incoming arcs.
 *  @param  vertex
 *  @return in
 **/
int* Vertex_In(Vertex *vertex){

  assert(vertex != NULL);
  return vertex->in;
}

/** Gets the number of incoming arcs.
 *  @param  vertex
 *  @return max_in
 **/
int Vertex_NoIn(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->no_in);
}

/** Sets one index, assumes index not already present, in the array with
 *  indices of those vertices connected to a vertex by outgoing arcs.
 *  If the array is too small, it gets extended to twice its current size.
 *  @param out
 **/
void Vertex_SetOut(Vertex *vertex, int out){

  assert(vertex != NULL);
  if(vertex->out != NULL){
    if(vertex->no_out >= vertex->max_out){
      vertex->out = (int*)realloc(vertex->out, ((vertex->max_out*2)*
                    sizeof(int)));
      vertex->max_out *= 2;
    }
  }
  else{
    vertex->out = (int*)calloc(1, sizeof(int));
    vertex->max_out = 1;
  }
  vertex->out[vertex->no_out++] = out;
}

/** Gets the array with indices of those vertices connected to a vertex
 *  by outgoing arcs.
 *  @param vertex
 *  @return out
 **/
int* Vertex_Out(Vertex *vertex){

  assert(vertex != NULL);
  return vertex->out;
}

/** Gets the number of outgoing arcs.
 *  @param vertex
 *  @return no_out
 **/
int Vertex_NoOut(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->no_out);
}

/** Sets the value for price.
 *  @param vertex
 *  @param price
 **/
void Vertex_SetPrice(Vertex *vertex, long double price){

  assert(vertex != NULL);
  vertex->price = price;
}
```

```
/** Gets the value for price.
 *  @param vertex
 *  @return price
 **/
long double Vertex_Price(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->price);
}

/** Sets the value for marked.
 *  @param vertex
 *  @param marked
 **/
void Vertex_SetMarked(Vertex *vertex, bool marked){

  assert(vertex != NULL);
  vertex->marked = marked;
}

/** Gets the value for marked.
 *  @param  vertex
 *  @return marked
 **/
bool Vertex_Marked(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->marked);
}

/** Sets the value for from.
 *  @param vertex
 *  @param from
 **/
void Vertex_SetFrom(Vertex *vertex, int from){

  assert(vertex != NULL);
  vertex->from = from;
}

/** Gets the value for from.
 *  @param  vertex
 *  @return from
 **/
int Vertex_From(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->from);
}

/** Sets one index, assumes index not already present, in the array with
 *  indices of those vertices visited before this vertex in a BS search.
 *  If the array is too small, it gets extended to twice its current size.
 *  @param in
 **/
void Vertex_SetPre(Vertex *vertex, int in){

  assert(vertex != NULL);
  if(vertex->predecessors != NULL){
    if(vertex->no_pre >= vertex->max_pre){
      vertex->predecessors = (int*)realloc(vertex->predecessors,
                             ((vertex->max_pre*2)* sizeof(int)));
      vertex->max_pre *= 2;
    }
  }
  else{
    vertex->predecessors = (int*)calloc(1, sizeof(int));
    vertex->max_pre = 1;
  }
  vertex->predecessors[(vertex->no_pre)++] = in;
}

/** Gets the array with indices of those vertices visited before this
 *  vertex in a BS search.
 *  @param  vertex
 *  @return predecessors
 **/
int* Vertex_Predecessors(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->predecessors);
}
```

```
/** Gets the number of predecessors.
 *  @param  vertex
 *  @return no_pre
 **/
int Vertex_NoPre(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->no_pre);
}

/** Sets the value for level.
 *  @param vertex
 *  @param level
 **/
void Vertex_SetLevel(Vertex *vertex, int level){

  assert(vertex != NULL);
  vertex->level = level;
}

/** Gets the value for level.
 *  @param  vertex
 *  @return level
 **/
int Vertex_Level(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->level);
}

/** Sets the value for cost
 *  @param vertex
 *  @param cost
 **/
void Vertex_SetCost(Vertex *vertex, long double cost){

  assert(vertex != NULL);
  vertex->cost = cost;
}

/** Gets the value for cost.
 *  @param  vertex
 *  @return cost
 **/
long double Vertex_Cost(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->cost);
}

/** Sets the value for capacity
 *  @param vertex
 *  @param cap
 **/
void Vertex_SetCap(Vertex *vertex, long double cap){

  assert(vertex != NULL);
  //if(Vertex_Index(vertex) == 100)
  // printf("old cap %Lf, new cap%Lf\n",vertex->capacity, cap );
  vertex->capacity = cap;
}

/** Gets the value for capacity.
 *  @param  vertex
 *  @return capacity
 **/
long double Vertex_Cap(Vertex *vertex){

  assert(vertex != NULL);
  return(vertex->capacity);
}

/** Changes the value for capacity with amount
 *  @param vertex
 *  @param amount
 **/
void Vertex_ChangeCap(Vertex *vertex, long double amount){

  assert(vertex != NULL);
  Vertex_Print(vertex);
  assert(amount <= vertex->capacity);
  vertex->capacity -= amount;
```

```
}

/** Returns true if two vertices have the same index
 *  @param  v1 vertex 1
 *  @param  v2 vertex 2
 *  @return true or false
 **/
bool Vertex_IsEqual(Vertex *v1, Vertex *v2){

  assert(v1 != NULL);
  assert(v2 != NULL);
  return(Vertex_Index(v1) == Vertex_Index(v2));
}

/** String representation of a vertex
 *  @param vertex
 **/
void Vertex_Print(Vertex *vertex){

  int i;

  assert(vertex != NULL);
   printf("Vertex %d has demand: %Lf\n", vertex->index, vertex->demand);
   printf("price               : %Lf\n", vertex->price);
   printf("marked              : %d\n", vertex->marked);
   printf("from                : %d\n", vertex->from);
   printf("cost                : %Lf\n", vertex->cost);
   printf("level               : %d\n", vertex->level);
   printf("capacity            : %.16Lf\n", vertex->capacity);
   if (vertex->no_in>0){
    printf("is head of arc from vertex: ");
    for (i=0; i<vertex->no_in; i++){
      printf("%d ", vertex->in[i]);
    }
    printf("%s", "\n");
  }
  if (vertex->no_out>0){
   printf("is tail of arc to vertex  : ");
   for (i=0; i<vertex->no_out; i++){
     printf("%d ", vertex->out[i]);
   }
   printf("%s", "\n");
  }
  if (vertex->no_pre>0){
    printf("has predecessors          : ");
    for (i=0; i<vertex->no_pre; i++){
      printf("%d ", vertex->predecessors[i]);
    }
    printf("%s", "\n");
  }
}

/*************************************************************************
* PRIVATE FUNCTIONS
*************************************************************************/

/** Returns old - amount if old > 0,
 *  otherwise old + amount.
 *  @param  old
 *  @param  change
 *  @return value
 **/
long double Vertex_ValidChange(long double old, long double amount){

  if(old>0){ // this is a source
    return old - amount;
  }
  else{ // old<0, this is a sink
    return old + amount;
  }
}
```

A.4 Miscellaneous code

A.4.1 Stack.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include "Stack.h"


/*typedef struct{
  int *indices; // the elements in the stack
  int no;       // maxsize of the stack
  int top;      // current array-index of stacks front
  }Stack;*/

/*****************************************************************************
* PUBLIC FUNCTIONS
*****************************************************************************/

/** Allocates memory for a stack.
 *  @param  no    number of elements in the stack
 *  @return stack
 **/
Stack *Stack_Create(int no){

  Stack *stack;

  stack = (Stack *)calloc(1, sizeof(Stack));
  stack->indices = (int *)calloc(no, sizeof(int));
  stack->no = no;
  stack->top = -1;
  return stack;
}

/** Returns memory for a stack.
 *  @param stack
 **/
void Stack_Destroy(Stack *s){

  free(s->indices);
  free(s);
}

/** Pushes an index on top of the stack, if index not already present.
 *  @param v index
 *  @param s stack
 **/
void Stack_PushOnto(Stack *s, int v){

  assert (s != NULL);
  assert(s->top < s->no);
  if(!Stack_Empty(s)){
    if(!Stack_Contain(s, v)){
      s->indices[++s->top] = v;
    }
  }
  else{
    s->indices[++s->top] = v;
  }
}

/** Pushes an index on top of the stack.
 *  Assumes index not present.
 *  @param v index
 *  @param s stack
 **/
void Stack_Push(Stack *s, int v){

  assert (s != NULL);
  assert(s->top < s->no);
  s->indices[++s->top] = v;
}

/** Returns the first element of the stack.
 *  @param s stack
 **/
int Stack_Top(Stack *s){

  assert (s != NULL);
  assert(s->top >= 0);
  return(s->indices[s->top--]);
```

```
}

/** Returns true if the stack is empty.
 *  @param s stack
 **/
bool Stack_Empty(Stack *s){

   assert (s != NULL);
   return(s->top < 0);
}

/** Returns true if the stack is full.
 *  @param s stack
 **/
bool Stack_Full(Stack *s){

   assert (s != NULL);
   return(s->top == s->no-1);
}

/** Prints a string representation
 *  in order bottom to top of the stack.
 *  @param s stack
 **/
 void Stack_Print(Stack *s){

   int i;

    assert (s != NULL);
    printf("%s", "[");
   for(i=0; i<=s->top; i++){
     printf("%d ",s->indices[i]);
   }
   printf("%s", "]");
 }

/** Returns true if the stack contains index, false otherwise.
 *  Assumes that the stack is nonempty.
 *  @param  s stack
 *  @param  v index
 *  @return true or false
 **/
bool Stack_Contain(Stack *s, int v){

  int i;

  for(i=0; i<=s->top; i++){
    if((s->indices[i]) ==  v){
      return true;
    }
  }
  return false;
}
```

A.4.2 Queue.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include "Queue.h"

/*typedef struct{
  int *indices; // the elements in the queue
  int no;       // maxsize of the queue
  int front;    // current array-index of queues front
  int back;     // current array-index of queues back
  int in_queue; // current number of elements in the queue
  }Queue;*/

/*****************************************************************************
* PUBLIC FUNCTIONS
*****************************************************************************

 * @param  no number of elements in the queue
 * @return queue
 */
Queue* Queue_Create(int no){
  Queue *queue;
  queue = (Queue *)malloc(sizeof(Queue));
  queue->indices = (int *)malloc(no*sizeof(int));
  queue->no = no;
  queue->front = queue->back = queue->in_queue = 0;
  return queue;
}

/* Returns memory for a queue.
 * @param queue
 */
void Queue_Destroy(Queue *q){
  free(q->indices);
  free(q);
}

/* Adds an index last in the queue, if index not already present.
 * @param v index
 * @param q queue
 */
void Queue_AddLast(Queue *q, int v){
  assert (q != NULL);
  assert(q->back <= q->no);
  if(!(Queue_Contain(q, v))){
    q->indices[q->back++] = v;
    q->in_queue++;
  }
}

/* Adds an index last in the queue.
 * Assumes index not present.
 * @param v index
 * @param q queue
 */
void Queue_Add(Queue *q, int v){
  assert (q != NULL);
  assert(q->back <= q->no);
  q->indices[q->back++] = v;
  q->in_queue++;
}

/* Returns the first index in the queue.
 * @param q queue
 */
int Queue_First(Queue *q){
  assert (q != NULL);
  assert(q->in_queue > 0);
  q->in_queue--;
  return(q->indices[q->front++]);
}

/* Returns true if the queue is empty.
 * @param q queue
 */
bool Queue_Empty(Queue *q){
  return(q->in_queue == 0);
}

/* Returns true if the queue is full.
 * @param q queue
```

```
 */
bool Queue_Full(Queue *q){
  return((q->back-1) -- q->no);
}

/* Prints a string representation of the queue.
 * @param q queue
 */
 void Queue_Print(Queue *q){
   int i;
   printf("%s", "[");
   for(i-q->front; i<q->back; i++){
     printf("%d ",q->indices[i]);
   }
   printf("%s", "]");
 }

/* Returns true if the queue contains index, false otherwise.
 * @param  q queue
 * @param  v index
 * @return true or false
 */
bool Queue_Contain(Queue *q, int v){
  int i;
  if(!(Queue_Empty(q))){
    for(i-q->front; i<q->back; i++){
      if((q->indices[i]) --  v){
        return true;
      }
    }
    return false;
  }
  else{
    return false;
  }
}
```

A.5 Test problem generator

A.5.1 TestGraphs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "TestGraphs.h"

/*****************************************************************************
* PRIVATE FUNCTIONS
*****************************************************************************/

/** By random, creates a dense bipartite graph with type 1 or type 2
 *  arc costs(Ohuchi&Kaji)
 *  @param  noSources number of sources in the network
 *  @param  noSinks   number of sinks in the network
 *  @param  h         factor for avarage flow on arcs in network
 *  @param  seed      seed for the random values
 *  @param  max_a     ceiling for the linear coefficient in the cost function
 *  @param  max_b     ceiling for the quadratic coefficient in the cost
                      function
 *  @return network
 **/
Network* TestGraphs_GenTestBD(int no_sources, int no_sinks, double h,
                              unsigned int seed, int max_a, int max_b){

  int  i, j, no_vertices;
  double demand;
  Vertex *vertex;
  Arc *arc;
  Network *g;

  no_vertices = no_sources+no_sinks;
  g = Network_CreateB(no_sources, no_sinks);
  srand(seed);

  // create sources and sinks and add them to network
  for(i=0; i<(no_vertices); i++){
    if(i<no_sources){
      // create sources an set them in sources
      vertex= Vertex_Create(i, 0, no_sinks);
      Network_SetSource(g, i);
    }
    else{
      // create sinks and set them in sinks
      vertex= Vertex_Create(i, no_sources, 0);
      Network_SetSink(g, i);
    }
    Network_AddVertex(g, vertex);
  }
  // create arcs, randomize arc limits and costs, add arcs to network
  for(i=0; i<no_sources; i++){
      for(j=0; j<no_sinks; j++){
        arc =  TestGraphs_CreateArc(i, (j+no_sources), max_a, max_b);
        Network_AddArc(g, arc);
      }
  }
  // set supply for sources, set demand for sinks
  for(i=0; i<no_vertices; i++){
    vertex = &Network_Vertices(g)[i];
    if(i<no_sources){
      demand = TestGraphs_CalcDemand(vertex, g, h, true);
      Vertex_SetDemand(vertex, demand);
    }
    else{
      demand = TestGraphs_CalcDemand(vertex, g, h, false);
      Vertex_SetDemand(vertex, -demand);
    }
  }
  return g;
}

/** By random, creates a sparse bipartite graph with type 1 or type 2
 *  arc costs(Ohuchi&Kaji), ca 10% of the arcs exist.
 *  @param  noSources the number of sources in the network
 *  @param  noSinks   the number of sinks in the network
 *  @param  h         factor for avarage flow on arcs in network
 *  @param  seed      the seed for the random values
 *  @param  max_a     the ceiling for the linear coefficient in the cost
                      function
 *  @param  max_b     the ceiling for the quadratic coefficient in the cost
                      function
```

```
 *  @return network
 **/
Network* TestGraphs_GenTestBS(int no_sources, int no_sinks, double h,
                              unsigned int seed, int max_a, int max_b){

  int  i, j, no_vertices, density, no_arcs;
  double demand;
  bool mkConnection;
  Vertex *vertex;
  Arc *arc;
  Network *g;
  bool done = false;

  no_vertices = no_sources+no_sinks;
  density = (int)floor(0.1*no_sinks);
  no_arcs = 0;
  g = Network_CreateB(no_sources, no_sinks);
  srand(seed);
  // create sources and sinks and add them to network
  for(i=0; i<no_vertices; i++){
    if(i<no_sources){
      // create sources
      vertex = Vertex_Create(i, 0, no_sinks);
      Network_SetSource(g, i);
    }
    else{
      // create sinks
      vertex = Vertex_Create(i, no_sources, 0);
      Network_SetSink(g, i);
    }
    Network_AddVertex(g, vertex);
  }
  // for each source create arcs to 10% of the sinks by random,
  // randomize arc limits and costs, add arcs to network
  for(i=0; i<no_sources; i++){
    while(!done){
      for(j=0; j<no_sinks; j++){
        mkConnection = false;
        mkConnection = TestGraphs_MkConnection(rand());
        if(mkConnection){
          arc =  TestGraphs_CreateArc(i, (j+no_sources), max_a, max_b);
          if(Network_AddArc(g, arc)){
            no_arcs++;
            if(no_arcs>-density){
              no_arcs = 0;
              done = true;
              break;
            }
          }
        }
      }
    }
    done = false;
  }
  // set supply for sources, set demand for sinks
  for(i=0; i<(no_vertices); i++){
    vertex = &Network_Vertices(g)[i];
    if(i<no_sources){
      demand = TestGraphs_CalcDemand(vertex, g, h, true);
      Vertex_SetDemand(vertex, demand);
    }
    else{
      demand = TestGraphs_CalcDemand(vertex, g, h, false);
      Vertex_SetDemand(vertex, -demand);
    }
  }
  return g;
}

/*****************************************************************************
* PRIVATE FUNCTIONS
*****************************************************************************/

/** Returns an arc with randomized arc limits and costs
 *  @param  i     index for the tail of the arc
 *  @param  j     index for the head of the arc
 *  @param  max_a ceiling for the linear coefficient of the cost function
 *  @param  max_b ceiling for the quadratic coefficient of the cost function
 *  @return arc
 */
Arc* TestGraphs_CreateArc(int i, int j, int max_a, int max_b){

  double lower, upper;
```

```
  QuadFun *cost;


  cost = QuadFun_Create(0,0);
  lower = TestGraphs_LowerLimit(rand());
  upper = TestGraphs_UpperLimit(((int)lower), rand());
  TestGraphs_Cost(max_a, rand(), max_b, rand(), cost);
  return(Arc_Create(i, j, lower, upper, cost));
}

/** Returns the demand for vertex (sum(arc->lower) +
                                  sum(arc->upper - arc->lower)*h).
 *  If out=true, the demand is set for a source and equals total value of
    outgoing arcs.
 *  If out=false, the demand is set for a sink and equals total value of
    incoming arcs.
 *  @param  vertex
 *  @param  n      network that the vertex belongs to
 *  @param  h      factor for avarage flow on arcs in network
 * @ return demand
 */
double TestGraphs_CalcDemand(Vertex *vertex, Network *n, double h, bool out){

  Arc *arc;
  double sum = 0;
  int i;

   // demand for sources
  if(out){
    for(i=0; i<Vertex_NoOut(vertex); i++){
      arc = &Network_Arcs(n)[Vertex_Index(vertex)][Vertex_Out(vertex)[i]];
      sum += Arc_Lower(arc) + ((Arc_Upper(arc)-Arc_Lower(arc))*h);
    }
  }
  // demand for sinks
  else{
    for(i=0; i<Vertex_NoIn(vertex); i++){
      arc = &Network_Arcs(n)[Vertex_In(vertex)[i]][Vertex_Index(vertex)];
      sum += (Arc_Lower(arc) + ((Arc_Upper(arc)-Arc_Lower(arc))*h));
    }
  }
  return(sum);
}

/** Sets the linear (a0 and quadratic (b) coefficient of cost.
 *  @param  max1  maximum value for the linear coefficient
 *  @param  rand1 int to decide the value of the linear coefficient,
 *                in the range [0<-a<-max1]
 *  @param  max2  maximum value for the quadratic coefficient
 *  @param  rand2 int to decide the value of the quadratic coefficient
 *                in the range [0<b<-max2]
 *  @param  cost QuadFun to hold the coefficients a and b
 */
void TestGraphs_Cost(int max1, int rand1, int max2, int rand2, QuadFun *cost){

  double a, b;
  double factor = 3.05185; // correction to obtain upper limit as
                           RAND_MAX-32767

  if(max1==10 && max2==1){
    a = (((double)rand1)/10000)*factor;
    b = (((double)rand2)/100000)*factor;
  }
  else{
    a = (((double)rand1)/100000)*factor;
    b = (((double)rand2)/10000)*factor;
  }
  cost->a = a;
  cost->b = b;
}

/* Based on the rand value, returns true if an arc should be created between
 * two vertices.
 * Returns true if rand <= max (= 0.1*RAND_MAX), false otherwise.
 * @param rand
 * @return true or false
 */
bool TestGraphs_MkConnection(int rand){

  double max = RAND_MAX*0.1;

  return (rand <= max);
}
```