Sensitivity Analysis of Traffic Equilibria

Magnus Josefsson

Abstract

This Master's thesis concerns the analysis of the sensitivity of traffic networks with autonomous, cost-minimizing users.

The Wardrop User Equilibrium Principle is used to model travellers' route choices in a traffic network. Basic theory for solving the resulting traffic assignment problems is discussed, and several algorithms are implemented and tested. In particular, Disaggregate Simplicial Decomposition (DSD)-type algorithms are studied. Algorithm performance is compared to previously published results, and specific performance issues are mentioned. It is found that DSD-type algorithms can produce good solutions even to large problems, and that for large networks, an implementation using a Newton-type search direction generation generally outperforms one using a simpler gradient projection-based method. DSD-type algorithms are found to be vastly better than the still often used Frank-Wolfe algorithm.

Reasons are given why network flow solutions are not always differentiable. Examples are presented of when a popular method of calculating traffic flow gradients breaks down. The need for a better method of obtaining sensitivity information is discussed. A method is then proposed utilizing solutions to optimization problems similar to the well known traffic assignment problems. The limitations of the method are discussed, and an implementation is given. Numerical results are presented for a well known problem.

In a further application, the bilevel optimization problem of network design/capacity enhancement is studied. The proposed method of sensitivity analysis is utilized as part of an algorithm for solving this problem. An implementation of the algorithm is tested thoroughly on well known networks, and the results are compared to previously published results. It is found that the method is able to produce better solutions than those previously published. The problem of non-uniqueness of local optima of the network design problem is discussed, and some ideas of possible ways of dealing with this are mentioned briefly. Some implementation mechanisms for further improving algorithm performance are discussed.

Sammanfattning

Detta examensarbete berör känslighetsanalys för trafiknätverk med autonoma, kostnadsminimerande användare.

Wardrop's användarjämviktsprincip används för att modellera resenärers ruttval i ett trafiknätverk. Vi diskuterar grundläggande teori för lösandet av de resulterande trafikproblemen, och implementerar och testar flera algoritmer. Framförallt studerar vi algoritmer av Disaggregate Simplicial Decomposition (DSD)-typ. Algoritmprestanda jämfrs med tidigare publicerade resultat, och specifika prestanda-relaterade observationer nämns. Vi finner att algoritmer av DSD-typ kan ge bra lösningar även till stora problem. Vidare konstaterar vi att för stora nätverk så överträffar en implementering som använder sig av en sökriktningsgenerering av Newton-typ implementeringar som använder en enklare gradientprojektionbaserad metod. DSD-algoritmer finnes vara avsevärt mycket bättre än den ännu ofta använda Frank-Wolfe-algoritmen.

Anledningar diskuteras till varför jämviktslösningar inte alltid är differentierbara. Vidare presenterar vi exempel på när en populär metod för att beräkna trafikflödesgradienter inte fungerar. Behovet för en bättre metod för att erhålla känslighetsinformation diskuteras. Vi föreslår sedan en metod som utnyttjar lösningen till ett optimeringsproblem liknande det för det välkända TAP trafikproblemet. Vi diskuterar begränsningarna till den här metoden och presenterar en implementering. Numeriska resultat ges för ett välkänt problem.

I en ytterligare applikation studeras det två-nivåoptimeringsproblem som ges av nätverksdesignproblemet/kapacitetsförbättringsproblemet för ett trafiknätverk. Den tidigare föreslagna metoden för känslighetsanalys begagnas som del av en algoritm för att lösa detta problem. En implementering av algoritmen testas noggrant på välkända nätverk, och resultaten jämförs med tidigare publicerade data. Vi finner att denna metod lyckas hitta noggrannare lösningar än de som tidigare publicerats. Vi diskuterar sedan icke-unikhetsproblemet för lokala optima i nätverksdesignproblemet och nämner idéer på möjliga sätt att lyckas hantera detta. Några implementeringsmekansimer för att vidare förbättra algoritmprestanda diskuteras.

Acknowledgments

I would like to thank my supervisor, professor Michael Patriksson, for his theory, ideas, comments, encouragement, proofreading, and many hours of discussions.

Contents

1 Introduction **7**

2 The Traffic Model **8**
2.1 The Wardrop condition for user equilibrium 8
2.2 TAP: mathematical program for user equilibrium 9
2.3 Properties of solutions to TAP 12
2.4 Example: Braess' network 13

3 Sensitivity Analysis **17**
3.1 The Tobin-Friesz method 17
3.1.1 Characteristics of Tobin and Friesz' method 17
3.1.2 Counterexample 1 – Non-differentiable equilibrium 18
3.1.3 Counterexample 2 – Differentiable equilibrium for which the Tobin-Friesz method fails 20
3.2 A different approach 21
3.2.1 Example 1 – Braess network 27
3.2.2 Example 2 – Sensitivity problem with non-unique solutions 28
3.2.3 Example 3 – Sensitivity problem with no solution 29

4 Network Design **31**
4.1 Overview 31
4.2 Bilevel programming 31
4.3 Solving the network design problem 32

5 Algorithms for Traffic Assignment Problems **33**
5.1 The Frank-Wolfe algorithm 33
5.1.1 Line search for the Frank-Wolfe algorithm 34
5.1.2 Problems with the Frank-Wolfe algorithm 34
5.1.3 Extensions of the Frank-Wolfe algorithm 35
5.2 The Disaggregate Simplicial Decomposition algorithm 35
5.2.1 Column generation 35
5.2.2 Restricted master problems 35
5.2.3 The algorithm 36

6 Implementation **37**
6.1 DSD algorithm 37
6.1.1 Data input 37
6.1.2 Data types 38
6.1.3 Settings and constants 38
6.1.4 Main algorithm 39
6.1.5 Route adding algorithm 39
6.1.6 Algorithms for restricted master problems 40

6.2 Sensitivity analysis . . . 44
6.2.1 Data input . . . 45
6.2.2 Data types . . . 45
6.2.3 Main algorithm . . . 45
6.2.4 Route generating algorithm . . . 46
6.2.5 Algorithm for the restricted master problem . . . 46
6.3 Network Design . . . 47
6.3.1 Data input . . . 48
6.3.2 Main algorithm . . . 48
6.3.3 Line search method . . . 48
6.3.4 Termination criteria . . . 49
6.3.5 Performance improvements . . . 49

7 Results - Algorithm Performance of TAP Solvers 50
7.1 Results of tests on well known networks . . . 50
7.1.1 Nguyen/Dupuis network . . . 51
7.1.2 Steenbrink network . . . 53
7.1.3 Harker and Friesz network . . . 53
7.1.4 Hull network . . . 55
7.1.5 Sioux Falls network . . . 59
7.1.6 Barcelona network . . . 62
7.1.7 Winnipeg network . . . 62
7.2 Algorithm performance and convergence issues . . . 67
7.2.1 Feasibility Errors . . . 67
7.2.2 Settings and constants . . . 67
7.2.3 CPU-time Consumption . . . 68
7.2.4 Line Search Tolerance . . . 69
7.2.5 Considerations for solving TAP for sensitivity analysis . . . 69

8 Results - Sensitivity analysis 71
8.1 Numerical example . . . 71
8.2 Impact of termination coefficient . . . 71
8.3 Impact of accuracy of original equilibrium solution . . . 74

9 Results - Network Design 76
9.1 Comparisons of results for well known networks . . . 76
9.1.1 Braess network . . . 76
9.1.2 Harker and Friesz network . . . 79
9.1.3 Sioux Falls network . . . 86
9.2 Performance and convergence issues . . . 91
9.2.1 Further discussion of non-differentiability . . . 91
9.2.2 Non-uniqueness of local optima . . . 92
9.2.3 Impact of DSD algorithm and sensitivity algorithm tolerances . . . 92

9.2.4 Choice of Armijo parameters 93
9.2.5 Mechanisms for improving performance 93

10 Conclusions **95**
10.1 Algorithms performance for TAP 95
10.2 Sensitivity analysis 95
10.3 Network design 95

References **97**

Appendix **98**

A Program Code **98**
A.1 Shell program / user interface 98
A.2 DSD algorithm TAP solver 105
A.3 Sensitivity analysis solver 113
A.4 Frank-Wolfe/PARTAN TAP Solvers 120
A.5 Network design solver 124

B Network/solution illustrating tool **128**

1 Introduction

Traffic theory and network flow-based traffic models have, particularly in the last few decades, become very important areas of mathematical research. Greater computational power now allow us to study and solve real life-scale problems accurately, and there is a great potential for benefit from the sound application of available theory in the form of well designed tools for traffic analysis and design.

User Equilibrium models (i.e. models based on Wardrop's First Principle) have become the standard for studying traffic networks with autonomous users, such as automobile road networks. Although the process of solving such problems (i.e. calculating network traffic flows from cost and demand functions) is now well understood, algorithms of relatively poor performance are still often used, and there are other aspects that are not so well studied. One very important such aspect of is that of sensitivity, i.e. the influence on the user equilibrium solutions of perturbations in the cost and demand functions. Sensitivity information is important for several reasons. Since network data such as cost functions are usually only approximations of real-life quantities, knowledge of sensitivities may indicate how well the solution to a model approximates the real-life network it is based on. Also, sensitivity analysis has many further applications, in particular that of network design. Network design involves determining values of controllable variables, such as link capacities, that will optimize the resulting network traffic flows according to some chosen criteria.

In this Master's thesis, we shall study the sensitivity analysis of traffic equilibria, with a focus on implementations of methods for small and real-life scale problems. We shall start by discussing the basic theory of the User Equilibrium model and its solutions, and then proceed to propose and implement a method for sensitivity analysis that will provide directional derivatives of network flows for fixed demand traffic assignment problems. The sensitivities studied will be those with respect to link capacities (i.e. we will not study demand sensitivities). Finally, we shall apply our method of sensitivity analysis in order to produce a network design algorithm for calculating optimum capacity enhancements for fixed demand networks. For each stage of the report, we will supply numerical results as well as comparisons with previously published data.

2 The Traffic Model

2.1 The Wardrop condition for user equilibrium

The basis for any traffic model is an assumption about the behavior of individual travelers. There are several different such assumptions that can be made depending on the application (and thus the nature of the traffic network), but the one that we shall make is the following: Each traveler has perfect knowledge of the costs (travel times) of all routes in the network and chooses his route as to minimize his own traveling cost. It it easily seen that, if the traffic network is in equilibrium (i.e. steady state), this assumption implies the following condition (Wardrop 1952):

> The journey times on all routes actually used are equal, and less than those which would be experienced by a single vehicle on any unused route.

This condition is known as Wardrop's first principle (or the Wardrop condition for user equilibrium).

Let us now look at Wardrop's first principle in notation. Let our network consist of a set of nodes, $\mathcal{N}$, connected by a set of directed links, $\mathcal{L}$. Further let there be a demand for transportation between certain pairs of nodes in the network. For any such pair p and q, let the demand be denoted by d_{pq}, $d_{pq} > 0$. The pair (p, q) is then referred to as an *OD-pair*, where p is known as the origin and q as the destination. Let $\mathcal{R}_{pq}$ be the set of cycle-free routes between p and q, and let $h_{pqr} \geq 0$ and c_{pqr} respectively denote the flow (i.e. traffic) on and cost of route $r, r \in \mathcal{R}_{pq}$. Let $\mathcal{C} \subseteq \mathcal{N} \times \mathcal{N}$ be the set of OD-pairs in the network. Now we can write Wardrop's first principle as

$$\begin{aligned} h_{pqr} > 0 &\Longrightarrow c_{pqr} = \pi_{pq}, \quad \forall r \in \mathcal{R}_{pq}, \quad \forall (p,q) \in \mathcal{C} \\ h_{pqr} = 0 &\Longrightarrow c_{pqr} \geq \pi_{pq}, \quad \forall r \in \mathcal{R}_{pq}, \quad \forall (p,q) \in \mathcal{C}, \end{aligned} \tag{1}$$

where π_{pq} is the cost of the cheapest route for the OD-pair (p, q). Of course, if Wardrop's condition is satisfied, the cost of the cheapest route is also the equilibrium route cost for the OD-pair. Now, for the flow to be *demand-feasible*, we also require that the total flow between each OD-pair to be equal to the demand, i.e.

$$\sum_{r \in \mathcal{R}_{pq}} h_{pqr} = d_{pq}, \quad \forall (p,q) \in \mathcal{C}. \tag{2}$$

In order for demand-feasibility to be possible, we shall henceforth make the natural assumption that the networks are always strongly connected with respect to the OD-pairs, i.e. there is at least one route from the origin to the destination for every OD-pair.

Throughout this project we will treat the transportation demands d_{pq} of a network as constant, i.e. we assume that we have *fixed demand*. In the more general case of *elastic demand*, the transportation demand associated with each OD-pair is a function of the cost of travel between that OD-pair. The reason we will consider the fixed demand situation is that it is slightly simpler while it still allows us to investigate most aspects of sensitivity analysis – aspects that will be relevant also to elastic cases. In fact, there are ways of modifying a network model for an elastic demand traffic assignment problem so that it can be treated as a fixed demand problem (Patriksson 1994, p.41).

The usefulness of a traffic equilibrium model based on Wardrop's first principle relies on the underlying assumptions. While it seems reasonable that each traveler attempts to minimize his travel cost, the assumption that all travelers have perfect information is not very realistic. In fact, one may question that the network will arrive at a steady state at all. In nearly any real-life traffic network, the demand and hence the congestion will fluctuate during each working day. This means that to reach a steady state (meaning in this particular context that flows are given by a function that depends only on the time of day), all travelers must learn to anticipate exactly how much traffic there will be on the different routes for every time of day. One way to address this problem is to use a non-deterministic, or *stochastic*, user equilibrium, in which the users' guesses about route costs are modeled using a random term. However, such a model still requires an approximation of the size of the random term. While stochastic equilibrium models have their uses, the deterministic user equilibrium principle is the one usually used in practice for the type of problems we will study.

We should mention that there is another frequently used traffic model based on another assumption – that of *system equilibrium*. In a system equilibrium model, it is assumed that all travelers behave so as to minimize the average travel cost of all travelers. However, this is obviously not applicable to any real-life traffic network with autonomous users (travelers) and is therefore not suitable for our uses.

2.2 TAP: mathematical program for user equilibrium

We now wish to express the user equilibrium model as a mathematical program that we can solve to obtain the equilibrium solution. We must start by introducing some more notation. Let v_l be the flow on the link $l \in \mathcal{L}$ and v_{lpq} be the flow on the same link due to the traffic between p and q (i.e. v_{lpq} is the *commodity link flow* for the commodity pq). These quantities are related to the route flows by

$$v_{lpq} = \sum_{r \in \mathcal{R}_{pq}} \delta_{pqrl} h_{pqr}, \quad \forall (p,q) \in \mathcal{C}, \ \ \forall l \in \mathcal{L}, \tag{3}$$

and

$$v_l = \sum_{(p,q) \in \mathcal{C}} v_{lpq}, \quad \forall l \in \mathcal{L}. \tag{4}$$

where

$$\delta_{pqrl} = \begin{cases} 1, & \text{link } l \text{ is on route } r \in R_{pq}, \\ 0, & \text{otherwise}, \end{cases} \qquad \forall l \in \mathcal{L},\ \forall r \in R_{pq},\ \forall (p,q) \in \mathcal{C}. \tag{5}$$

Furthermore, let t_l be the travel cost function associated with the link $l \in \mathcal{L}$. In order to make the model simple, we must make some assumptions about the travel cost functions. We assume that travel costs are *separable*. What this means is that each link travel cost function is a function solely of the flow on that link, i.e. $t_l(v) = t_l(v_l)$. We also assume that travel costs are *additive*, meaning that the total travel cost of a route is the sum of the travel costs of all the links on the route. Finally, we also assume that each t_l is positive, continuously differentiable, and strictly increasing.

Assumption
- The traffic network is strongly connected w.r.t. the OD-pairs
- The demand for transportation is fixed and non-negative for each OD-pair
- Travel costs are additive and separable
- Travel cost functions are positive and continuously differentiable
- Travel cost functions are strictly increasing

Table 1: Assumptions about the network model.

While assuming additive travel costs may seem quite reasonable in most cases, the assumption of separability is less realistic. In most real-life traffic networks, links (roads) will cross (e.g. at traffic lights) and the flow (traffic) on one link will influence the cost of the other. Nevertheless, it is necessary to assume separable travel costs in order to obtain a workable model. There are ways to modify the representation of a separable network model in order to incorporate non-separable elements (see e.g. Patriksson 1994).

Before we continue to the main result of this section, we will summarize our assumptions about the model in Table 1. Henceforth, these assumptions will be held to be true for every traffic network model.

Theorem 1 Consider a traffic network model subject to the following optimization problem:

$$\underset{v}{\text{minimize}}\ T(v) = \sum_{l \in \mathcal{L}} \int_0^{v_l} t_l(s)ds, \tag{6}$$

subject to

$$\begin{array}{rcll} h_{pqr} & \geq & 0, & \forall r \in \mathcal{R}_{pq} \quad \forall (p,q) \in \mathcal{C} \\ \sum_{r \in R_{pq}} h_{pqr} & = & d_{pq}, & \forall (p,q) \in \mathcal{C} \\ \sum_{(p,q) \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}} \delta_{pqrl} h_{pqr} & = & v_l, & \forall l \in \mathcal{L} \end{array} \tag{7}$$

A set of link flows v satisfy first order optimality conditions to this problem if and only if they also satisfy Wardrop's first principle and demand feasibility (equations (1) and (2)).

Note to Theorem 1 We may express the feasible set (7) equivalently as

$$\begin{array}{rcl} h & \geq & 0, \\ \Lambda h & = & d, \\ \Delta h & = & v, \end{array} \tag{8}$$

where Δ and Λ are the route-link and route-OD-pair incidence matrices, respectively. I.e., element (l, r) of Δ is 1 if route r uses link l, zero otherwise. Similarly, element (pq, r) of Δ is one if route r is a route for the OD-pair pq, zero otherwise.

Proof Let D be the set of link flows v such that there exist route flows h so that (7) is satisfied. Then v^* is an optimal solution to (6),(7) if and only if

$$\iff \sum_{l\in\mathcal{L}} \int_0^{v_l^*} t_l(s)ds \leq \sum_{l\in\mathcal{L}} \int_0^{v_l} t_l(s)ds \quad \forall v \in D \tag{9}$$

$$\iff \sum_{l\in\mathcal{L}} \int_{v_l^*}^{v_l} t_l(s)ds \geq 0 \quad \forall v \in D \tag{10}$$

$$\iff \text{(all } t_l \text{ incr.)} \sum_{l\in\mathcal{L}} t_l(v_l^*)(v_l - v_l^*) \geq 0 \quad \forall v \in D \tag{11}$$

$$\iff \sum_{l\in\mathcal{L}} t_l(v_l^*)v_l^* \geq \sum_{l\in\mathcal{L}} t_l(v_l^*)v_l \quad \forall v \in D \tag{12}$$

Now let h^* denote a route flow solution such that h^* and v^* together satisfy (7). The above inequalities are further equivalent to (h^*, v^*) solving the linear programming problem

$$\underset{h,v}{\text{minimize}} \sum_{l\in\mathcal{L}} t_l(v_l^*)v_l \quad \text{subject to (7)} \tag{13}$$

Note that the condition $v \geq 0$ is implicit in (7). The dual of this problem is (denoting the dual variables by u and w):

$$\underset{u,w}{\text{maximize}} \sum_{(p,q)\in\mathcal{C}} d_{pq}u_{pq} \tag{14}$$

subject to

$$\begin{array}{rcll} u_{pq} - \sum_{l\in\mathcal{L}} \delta_{pqrl}w_l & \leq & 0, & \forall r \in \mathcal{R}_{pq}, \quad \forall (p,q) \in \mathcal{C} \\ w_l & \leq & t_l, & \forall l \in \mathcal{L} \end{array} \tag{15}$$

The complementarity conditions between the two problems are

$$\begin{array}{lcll} h_{pqr} > 0 & \Longrightarrow & u_{pq} - \sum_{l\in\mathcal{L}} \delta_{pqrl}w_l = 0, & \forall r \in \mathcal{R}_{pq}, \quad \forall (p,q) \in \mathcal{C} \\ v_l > 0 & \Longrightarrow & w_l = t, & \forall l \in \mathcal{L} \end{array} \tag{16}$$

Now, at least one route for each OD-pair must be used, and all routes along any unused link must be unused. Hence u_{pq} is in fact the cost of the cheapest route for the OD-pair pq, π_{pq}, and the complementarity conditions plus feasibility constraints are actually equivalent to

$$\begin{array}{lcll} h_{pqr} > 0 & \Longrightarrow & \pi_{pq} = \sum_{l\in\mathcal{L}} \delta_{pqrl}t_l, & \forall r \in \mathcal{R}_{pq}, \quad \forall (p,q) \in \mathcal{C} \\ & & \pi_{pq} \leq \sum_{l\in\mathcal{L}} \delta_{pqrl}t_l, & \forall r \in \mathcal{R}_{pq}, \quad \forall (p,q) \in \mathcal{C} \end{array} \tag{17}$$

plus the feasibility constraints for the link flow solution. (17) is equivalent to (1), so a set of link flows is an optimal solution to (6),(7) if and only if they satisfy Wardrop's condition and feasibility. □

The optimization problem (6),(7) of Theorem 1 above is known as the *Traffic Assignment Problem* (TAP). The representation of the traffic network used in this formulation of TAP is known as the *link-route representation*, since the feasibility constraints are written in terms of route flows. There is an alternative way of writing the constraints: Feasibility can be ensured by making sure that the net commodity flows (i.e. the sum of the commodity link flows to the node minus the sum of the commodity link flows from the node, for each commodity) to each node have the correct values. The net commodity flow to each node must be zero if that node is neither the origin or the destination of the commodity, equal to the demand if it is the origin, and equal to minus the demand if it is the destination. This way of writing the constraints is known as the *link-node representation*. While TAP can be formulated and solved using either representation, it should be noted that the feasible sets of flows of the two formulations are in fact not equal. This is because the link-node representation allows cyclical flows, whereas the link-route representation does not (as only simple routes are considered). However, the two formulations of TAP generated by the different representations have the same sets of solutions. This is because cycle flows cannot exist at equilibrium as it would always be advantageous for a user traveling in a cycle to skip the cycle (since the link cost functions are positive). We shall mostly make use of the link-route representation.

2.3 Properties of solutions to TAP

Now that Theorem 1 has established that the sets of link flows that solve TAP are exactly those that satisfy Wardrop's first principle, it is natural to ask under what conditions TAP has an optimal solution. The following theorem provides the answer.

Theorem 2 Consider a traffic network model in accordance with the assumptions in Table 1. There exists at least one optimal solution to the link-route representation of TAP.

Proof The network is strongly connected, so we can always find a route for each OD-pair to which we can assign all the demand, thus yielding a feasible solution. Thus the feasible set of TAP is non-empty. The set is also closed, as it is polyhedral. It is also bounded, since link flows are non-negative and bounded above by the sum of the OD-pair demands. Now, the link travel cost functions are continuous, so the objective function of TAP is as well. Hence, by Weierstrass' Theorem, there exists a minimum value of the objective function of TAP on the feasible set and thus an optimal solution to TAP. □

So TAP always has a solution, and hence solving TAP will always provide us with a user equilibrium for any of our traffic network models. Note that although the theorem considers the link-route formulation, the result is true also for the link-node formulation. The reason for this is that the solution to the link-route formulation of TAP will always be a feasible solution in the link-node formulation (see Flow decomposition theorem, ref). Additionally, since any feasible solution to the link-node formulation that is not feasible in the link-route formulation must contain cycles and thus have a higher objective function value than the corresponding cycles-removed link-route-feasible solution, this solution must always be optimal.

Theorem 3 Consider a traffic network model in accordance with the assumptions in Table 1. The optimal equilibrium link flows are unique.

Proof Assume that there are two different optimal link flow solutions, v^a and v^b. Let h^a and h^b be route flow solutions consistent with $v = v^a$ and $v = v^b$, respectively, i.e.,

$$v_l^a = \sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqrl} h_{pqr}^a \quad \text{and} \quad v_l^b = \sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqrl} h_{pqr}^b, \quad \forall l \in \mathcal{L}. \tag{18}$$

Let $h^c = \frac{1}{2}(h^a + h^b)$. Since both h^a and h^b are positive and demand-feasible, h^c is as well. So h^c is a feasible route flow solution. Let v^c denote the set of link flows corresponding to the route flow solution h^c, i.e.

$$v_l^c = \sum_{(p,q)\in\mathcal{C}} \sum_{l\in\mathcal{L}} \delta_{pqrl} h_{pqr}^b, \quad \forall l \in \mathcal{L}. \tag{19}$$

Since h^c is feasible, v^c is as well. We have

$$T(v^c) = T\left(\sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqr} h_{pqr}^c\right) = T\left(\sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqr} \frac{1}{2}(h_{pqr}^a + h_{pqr}^b)\right) \tag{20}$$

$$= T\left(\frac{1}{2} \sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqr} h_{pqr}^a + \frac{1}{2} \sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqr} h_{pqr}^b\right) = T\left(\frac{1}{2}(v^a + v^b)\right) \tag{21}$$

But since t_l is strictly increasing $\forall l \in \mathcal{L}$, $T(v)$ is a strictly convex function. So,

$$T(v^c) = T\left(\frac{1}{2}(v^a + v^b)\right) < T\left(\frac{1}{2}v^a\right) + T\left(\frac{1}{2}v^b\right) = T(v^a) = T(v^b). \tag{22}$$

Hence neither v^a nor v^b can be optimal. By contradiction, the optimal link flow solution is unique. □

Although strictly increasing travel cost functions guarantee unique equilibrium link flows, we should point out that neither *commodity* link flows nor route flows need be unique. To see this, consider the network in Figure 1. Let there be two OD-pairs, $(1, 5)$ and $(1, 6)$, with demand 1 each. Let the equilibrium link flow be 1 on each of the links in the network. Now, one optimal route flow solution is clearly a flow of 1 unit along the route $[2, 3, 5]$ and another flow of 1 unit along the route $[1, 4, 6]$. Then the commodity link flows for the OD-pair $(1, 5)$ are 1 on links 2, 3, and 5, and 0 on the other links. Similarly, the commodity link flows for the OD-pair $(1, 6)$ are 1 on the links 1, 4, and 6, and 0 on the rest. However, we could instead of the above mentioned route flows have flows of 1 along the routes $[1, 4, 5]$ and $[2, 3, 6]$, which also add up to total link flows of 1 on each link and thus constitute an optimal solution. Also the commodity link flows would clearly be different in this case. So neither route flows nor commodity link flows are generally unique.

2.4 Example: Braess' network

Consider the network in Figure 2, and let

$$t_1 = 10v_1, \quad t_2 = v_2 + 50, \quad t_3 = v_3 + a, \quad t_4 = v_4 + 50, \quad \text{and} \quad t_5 = 10v_5, \tag{23}$$

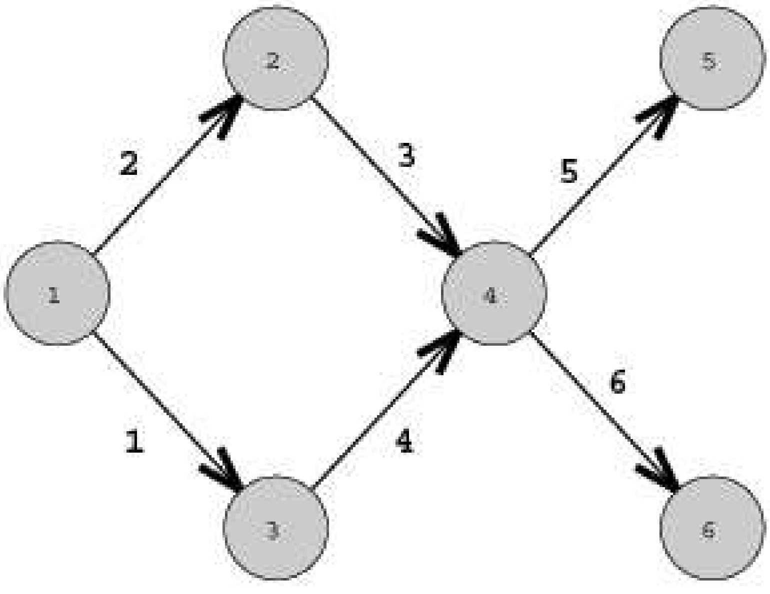


Figure 1: A simple network

where $a > 0$ is a constant. Let there be one OD-pair, $(1, 4)$ with demand 6.

We will now calculate the equilibrium solution to the traffic assignment problem for this network. Although in this case it is relatively easy to work out the equilibrium flows directly from Wardrop's principle, we shall still use the TAP integral. We have,

$$T(v) = \sum_{l \in \mathcal{L}} \int_0^{v_l} t_l(s)\, ds \tag{24}$$

$$= \int_0^{v_1} 10s\, ds + \int_0^{v_2} s + 50\, ds + \int_0^{v_3} s + a\, ds + \int_0^{v_4} s + 50\, ds + \int_0^{v_5} 10s\, ds \tag{25}$$

$$= 5(v_1^2 + v_5^2) + 50(v_2 + v_4) + \frac{1}{2}(v_2^2 + v_3^2 + v_4^2) + av_3. \tag{26}$$

Let us rewrite this in terms of route flows. By symmetry, the flows on routes $1 : [2, 5]$ and $2 : [1, 4]$ must be equal. Let h_0 be the flow on each of these routes, i.e. $h_0 = h_1 = h_2$. To satisfy demand-feasibility, the flow on the last route $3 : [1, 3, 5]$ must then be $h_3 = 6 - 2h_0$. Hence we have,

$$T(h_0) = 5(2(6 - h_0)^2) + 50(2h_0) + \frac{1}{2}(2h_0^2 + (6 - 2h_0)^2) + a(6 - 2h_0) \tag{27}$$

$$= 13h_0^2 - (32 + 2a)h_0 + 378 + a, \qquad \text{where } 0 \le h_0 \le 3. \tag{28}$$

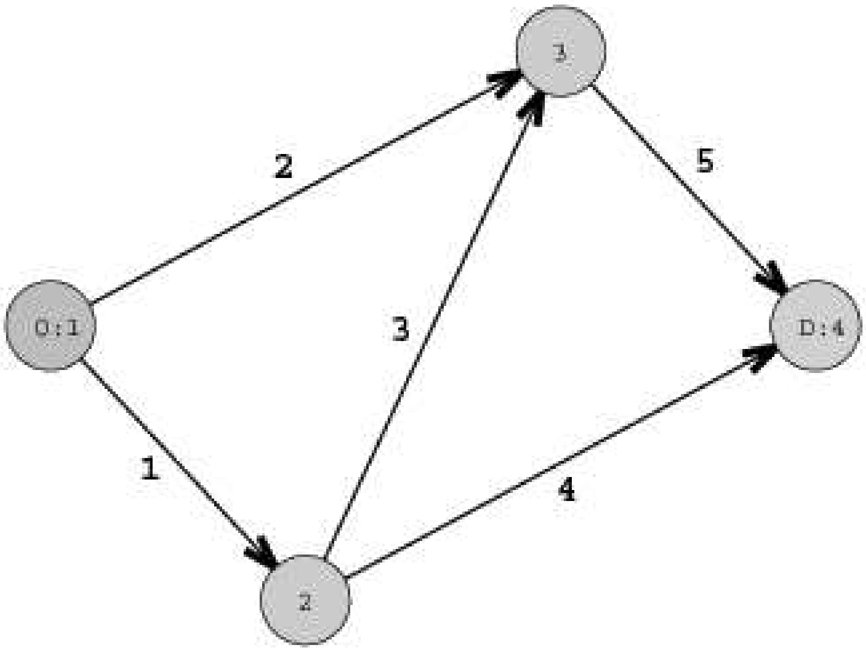


Figure 2: Braess' network

The above quadratic equation has a minimum at $h_0 = \frac{16+a}{13}$. The equilibrium route flows are hence

$$\begin{array}{llll} h_1 = h_2 = \frac{16+a}{13}, & h_3 = \frac{46-2a}{13}, & \text{if} & a \leq 23 \\ h_1 = h_2 = 3, & h_3 = 0, & \text{if} & a > 23. \end{array} \tag{29}$$

Thus, the equilibrium link flows are

$$\begin{array}{lllll} v_1 = v_5 = \frac{62-a}{13}, & v_2 = v_4 = \frac{16+a}{13}, & v_3 = \frac{46-2a}{13}, & \text{if} & a \leq 23 \\ v_1 = v_5 = 3, & v_2 = v_4 = 3, & v_3 = 0, & \text{if} & a > 23. \end{array} \tag{30}$$

Equilibrium OD-pair cost can be calculated

$$\pi_{14} = t_1(v_1) + t_4(v_4) \tag{31}$$

$$\Longrightarrow \begin{array}{llll} \pi_{14} = 10\frac{62-a}{13} + \frac{16+a}{13} + 50 & = & \frac{1286-9a}{13}, & a \leq 23 \\ \pi_{14} = 10(3) + 3 + 50 & = & 83, & a > 23. \end{array} \tag{32}$$

We notice here that the equilibrium travel cost for this network is actually higher the cheaper link 3 is. This counter-intuitive result is known as *Braess' Paradox*.

3 Sensitivity Analysis

Sensitivity analysis of traffic equilibria is the analysis of the behavior of the equilibrium solution (i.e. flows, costs) to a traffic problem in response to a small change in some parameters of the problem. In other words, we would like the means to be able to calculate some kind of *derivative* of the equilibrium solution with respect to problem parameters. One important application of such a calculation would be to use it in a bilevel optimization scheme in order to 'tweak' network parameters according to our preferences. This is referred to as *network design* and is discussed in detail in Section 4.

Another motivation for performing sensitivity analysis is the fact that network models are nearly always only approximations of real-life networks. Sensitivity analysis is therefore useful in order to tell us how large the errors due to the uncertainties in the modeling of the network (i.e. in the network parameters) might be.

One conclusion that we can draw about sensitivity analysis of traffic equilibria without the need for a technical analysis is that traffic equilibria can generally not be expected to be differentiable. This is easily seen by considering an equilibrium solution where there is an unused route with the same cost as the used routes for the same OD-pair. Such an equilibrium solution is called a *non-strictly complementary* solution. In such a case, changing a parameter to increase the cost of the unused route would clearly not influence the equilibrium solution. On the other hand, disturbing the network to reduce the cost of the unused route would tend towards an equilibrium solution in which this route was used. Thus, in this case, the equilibrium solution would not be differentiable.

We shall start this section by examining an older method for sensitivity analysis that is still often used. We shall then go on to present the different approach that we shall utilize in constructing an efficient and reliable implementation of sensitivity analysis.

3.1 The Tobin-Friesz method

Tobin and Friesz (1988) proposed a quite direct method of performing sensitivity analysis for equilibrium network flows. This method gives as the output a gradient vector for the equilibrium link flows. Of course, the equilibrium link flows are not necessarily differentiable, which means that the method cannot possibly work for all cases. Furthermore, as we shall see by looking at some examples, there are additional serious problems with the Tobin-Friesz method. We should point out that Friesz and some other researchers has written a subsequent paper (Cho et al, 1999) in which the mistakes are avoided. However, as the original Tobin-Friesz method is still widely use, a discussion of its problems is nevertheless relevant.

3.1.1 Characteristics of Tobin and Friesz' method

In their method, Tobin and Friesz assume a certain condition on the equilibrium solution. They refer to this as the strict complementarity condition, although it is quite different from our normal route flow-based concept of strict complementarity. Likely, this condition was meant to make sure that the method was not used on network problems where it would not work. However, as we shall see, the condition is not well chosen.

The strict complementarity condition assumed by Tobin and Friesz is:

$$v_a^*(0) = 0, \Longrightarrow t_a(v_a^*(0), 0) \ > \ \lambda_j(0) - \lambda_i(0). \tag{33}$$

Here, a is the link between nodes i and j, (0) denotes unperturbed quantities, and v^* denotes equilibrium flows. The λ's are referred to as 'node numbers' and are defined in the paper as numbers for each node so that

$$\begin{aligned} v_a^*(0) > 0, \Longrightarrow t_a(v_a^*(0), 0) \ &= \ \lambda_j(0) - \lambda_i(0) \\ v_a^*(0) = 0, \Longrightarrow t_a(v_a^*(0), 0) \ &\geq \ \lambda_j(0) - \lambda_i(0) \end{aligned} \tag{34}$$

is satisfied for all links a.

There are two big problems with the condition. First, it only considers cases where there are unused *links*. An equilibrium solution can be non-differentiable when there is an unused route equally cheap as the used ones – even if all links on the route are used by traffic on other routes. A second problem with the condition is that the 'node numbers' λ may actually not exist. However, we shall focus mostly on the consequences of the first problem.

Tobin and Friesz state that, prior to applying their method, one should choose a single route flow solution to associate with the link flow solution. However, there are certain requirements for this route flow solution that Tobin and Friesz seemingly assume can be filled (without explicitly stating it as an assumption). The conditions are that

- 1) The route flow solution is an extreme point in the set of route flow solutions.
- 2) The route flow solution is non-degenerate.

The meaning of the second condition is that the number of the routes with positive flow in the route flow solution is equal to the rank of the matrix $[\Delta^\top | \Lambda^\top]$, where Δ and Λ are the route-link and route-OD-pair incidence matrices (see equation (8)).

After a route flow solution has been chosen, Tobin and Friesz' method is quite straightforward. It basically involves restricting the network to those routes with positive flow in the chosen route flow solution and putting the resulting reduced matrices through a series of formulae. One reason for the conditions on the route flow solution is to make sure that a large Jacobian matrix that is inverted in one of the formulae is not singular. However, as well shall see, the conditions are in some cases impossible to satisfy, even though a solution exists. In these cases, the Tobin-Friesz method does not work.

3.1.2 Counterexample 1 – Non-differentiable equilibrium

Consider the network in Figure 3. Let there be three OD-pairs with demands $d_{12} = 1$, $d_{13} = 1$, and $d_{32} = 1$, respectively. Let the link cost functions be $t_1(v_1) = 2v_1 + \rho$, $t_2(v_2) = v_2$, and $t_3(v_3) = v_3$, with the perturbation parameter $\rho = \rho_0 = 0$. The equilibrium solution for the unperturbed network is then clearly $v_1 = 1$, $v_2 = 1$, $v_3 = 1$, with the (in this case unique) route flows being 1 on routes $1([1])$, $3([2])$ and $4([3])$ and 0 on route $2([2, 3])$. Let us now consider

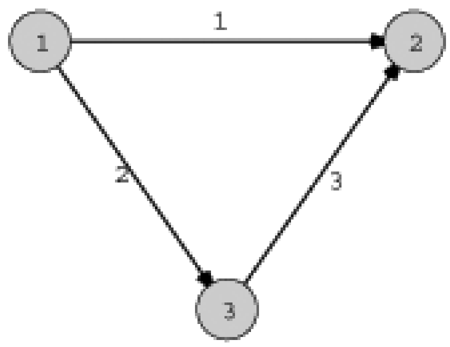


Figure 3: A network

performing sensitivity analysis on this network with respect to the parameter ρ. Note that the equilibrium solution is non-strictly complementary, since route 2 has the same cost as route 1 but is unused. In particular, it is clear that a small negative perturbation in ρ would not affect the equilibrium solution (since route 1 already has all flow) while a small positive perturbation *would* change the solution. Hence there cannot be a gradient at this point, only directional derivatives. This already tells us that Tobin and Friesz' method will not work for this problem (as it can only give gradients), but let us still see what happens when we try to apply it.
First, note that this equilibrium solution actually satisfies the strict complementarity assumption required by Tobin and Friesz' method. This follows trivially since all links have positive flow. So, according to Tobin and Friesz, there's no reason why we should not be able to use their method here. Setting up the incidence matrices, we have:

$$\Lambda^\top = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \Delta^\top = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{35}$$

which gives the composite matrix

$$[\Delta^\top | \Lambda^\top] = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}, \tag{36}$$

which has rank 4. So, by Tobin and Friesz' method, we should express the link flow solution as a route flow solution that has four routes with positive flow. Of course, this is impossible, since the flow on route 2 is necessarily zero. If we still try to use the method and include all four routes in the restricted set, we will end up with an answer that neglects the fact that the flow on route 2 cannot be decreased. In other words, we would likely get the wrong answer. This example

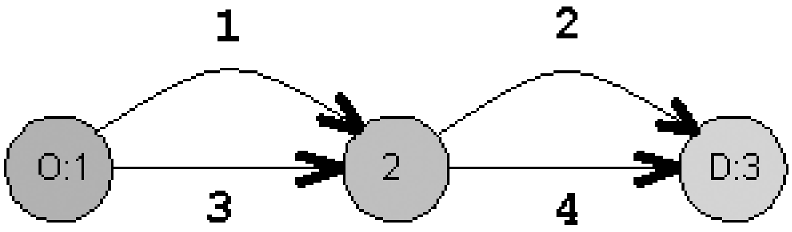


Figure 4: A network

thus shows that it is possible to have an equilibrium solution for which Tobin and Friesz' strict complementarity assumption is satisfied but that still does not have a gradient and for which Tobin and Friesz' method does not work.

3.1.3 Counterexample 2 – Differentiable equilibrium for which the Tobin-Friesz method fails

Consider the network in Figure 4. Let there be a single OD-pair $(1, 3)$ with demand $d_{13} = 2$. Let the link cost functions be $t_1(v_1) = v_1 + \rho$, $t_2(v_2) = v_2$, $t_3(v_3) = v_3$ and $t_4(v_4) = v_4$, with perturbation parameter $\rho = \rho_0 = 0$. By symmetry, the unperturbed equilibrium solution is clearly $v_1 = 1$, $v_2 = 1$, $v_3 = 1$, $v_4 = 1$. Let us now perform sensitivity analysis on this network. By symmetry and inspection, it is easy to see that the equilibrium solution is in fact differentiable with

$$\frac{\partial v_1}{\rho} = -\frac{1}{2}, \frac{\partial v_2}{\rho} = 0, \frac{\partial v_3}{\rho} = \frac{1}{2}, \frac{\partial v_4}{\rho} = 0. \tag{37}$$

Let us see what happens when we try to use Tobin and Friesz' method. First, note that this equilibrium solution satisfies the strict complementarity assumption required by Tobin and Friesz' method, since all links have positive flow. We have,

$$\Lambda^\top = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \Delta^\top = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \tag{38}$$

which gives the composite matrix

$$[\Delta^\top | \Lambda^\top] = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}, \tag{39}$$

which has rank 3. So, by Tobin and Friesz' method, we should express the link flow solution as a route flow solution that has three routes with positive flow. However, this impossible because of the symmetry. To see this, let the route flow on route $[1, 2]$ be a. Then, to satisfy $v_1 = 1$ and $v_2 = 1$, the route flows on $[1, 4]$ and $[3, 2]$ must both be $1 - a$. To satisfy $v_3 = 1$ and $v_4 = 1$, the flow on route $[3, 4]$ must then be a. Hence, any route flow solution must have either all route with positive flow or just two routes with positive flow. Tobin and Friesz' method therefore does not work for this example. In fact, even if we still attempt to use a route flow with four routes of positive flow, the Jacobian matrix used in the method turns out to be singular, so we still do not get an answer. This example therefore shows that there are examples for which there exist gradients but for which Tobin and Friesz' method doesn't work at all. Furthermore, there is nothing in the assumptions to indicate that the method will not work here – the problem is encountered first when one notices that it is impossible to find a satisfactory route flow solution.

3.2 A different approach

Clearly, a reliable method for sensitivity analysis will have to take into account the fact that equilibrium solutions are not generally differentiable. Keeping this in mind, a natural next question is whether equilibrium solutions are generally *directionally* differentiable. To approach this question, we shall make use of the following general technical result from Dontchev and Rockafellar (2002).

Theorem 4 Consider a variational inequality problem of the following form: Let $X \subseteq \mathcal{R}^n$ be a polyhedral set and let $f : \mathcal{R}^d \times \mathcal{R}^n \mapsto \mathcal{R}^n$ be a differentiable mapping in the parameters $\rho \in \mathcal{R}^d$ and variables $x \in \mathcal{R}^n$. Find $x^* \in X$ such that

$$f(\rho, x^*)^\top (x - x^*) \geq 0, \quad \forall x \in X. \tag{40}$$

Now let $S(\rho)$ be the set of solutions to the problem (40) and let $\rho = \rho^*$ be the current value of the parameter vector. Assume that f is monotone on X around $\rho = \rho^*$ and that rank $\nabla_\rho f(\rho^*, x^*) = n$. Then

$$\begin{gathered} S \text{ is strongly regular (i.e. single-valued and locally Lipschitz continuous) at } \rho^* \\ \Longleftrightarrow \\ DS(\rho^* | x^*) \text{ is single-valued,} \end{gathered} \tag{41}$$

where $DS(\rho^*|x^*)(\rho')$ is the solution to the related variational inequality problem of finding $x' \in K$ such that

$$(\nabla_\rho f(\rho^*, x^*)\rho' + \nabla_x f(\rho^*, x^*)x')^\top (x - x') \geq 0, \quad \forall x \in K, \tag{42}$$

where

$$K := T_X(x^*) \cap f(\rho^*, x^*)^\perp. \tag{43}$$

Furthermore, the single value x' of $DS(\rho^*|x^*)(\rho')$ is then the directional derivative of x^* in the direction ρ'.

Before applying this result, let us consider it. Is does seems intuitively reasonable that $DS(\rho^*|x^*)(\rho')$ would be a directional derivative in the direction ρ' as

$$\nabla_\rho f(\rho^*, x^*)\rho' + \nabla_x f(\rho^*, x^*)x'$$

looks exactly like a chain rule expression for the differential of a function $f(\rho, x)$ in real analysis. Furthermore, the fact that x' is restricted to the tangent cone of X at x^*, $T_X(x')$, also makes sense, since this means that $x = x^* + x'$ must be feasible for small perturbations x'. Finally, the fact that x' must lie in the orthogonal complement of $f(\rho^*, x^*)$, $f(\rho^*, x^*)^\perp$, can be interpreted as a requirement for x to retain optimality in the original variational inequality after the perturbation x' is added.

Now, in order to understand what this result means for us, let us apply it to the framework of the traffic assignment problem. We consider a traffic network model in accordance with our assumptions but let the travel cost functions depend both on the link flows and also on a set of parameters ρ. Without loss of generality, we can assume that there is a one-one correspondence between link cost functions and parameters, i.e. $t_l = t_l(v_l, \rho_l)\ \forall l \in \mathcal{L}$ (other situations can still be modeled in this way by imposing constraints on the allowable perturbation directions). We further assume that the travel cost functions are monotone with respect to the parameter values ρ. Now, as we saw in the proof of Theorem 1, we can write TAP as the variational inequality problem

$$t(\rho, v^*)^\top (v - v^*) \geq 0, \quad \forall (h, v) \in C, \tag{44}$$

where C is the feasible set for the TAP as defined in equation (7). This problem is clearly of the form required by the theorem, with $(0, t)$, (h, v), and C corresponding to f, x, and X. Also, we know that the feasible set C is polyhedral, and the travel cost mapping t is differentiable as well as monotone around each $\rho = \rho^*$ by our assumptions. Furthermore, we can easily choose our parameters ρ such that the requirement on the rank of the Jacobian matrix is satisfied, for example by having a one-one correspondence between parameters and links (so that changing any one parameter influences exactly one link travel cost). Hence we can apply the theorem.

We not try to identify the feasible set of (h', v') in the auxiliary variational inequality. It should be noted that the v'- variables are in fact uniquely determined by the h'-variables, so we are really looking for the feasible set of h'. Let us first consider $T_C(h^*, v^*)$, the tangent cone of C at the equilibrium solution. A direction (h', v') is in $T_C(h^*, v^*)$ if and only if $(h^*, v^*) + \delta(h', v') \in C$ for some sufficiently small $\delta > 0$. So:

$$\begin{pmatrix} h' \\ v' \end{pmatrix} \in T_C(h^*, v^*) \iff \begin{cases} \sum_{r \in \mathcal{R}_{pq}} h'_{rpq} = 0 \quad \forall (p,q) \in \mathcal{C} \\ \{h^*_{rpq} = 0\} \implies h'_{rpq} \geq 0 \quad \forall (p,q) \in \mathcal{C}. \end{cases} \tag{45}$$

Let us now consider $t(\rho^*, v^*)^\perp$, the orthogonal complement of the cost functions:

$$\begin{pmatrix} h' \\ v' \end{pmatrix} \in t(\rho^*, v^*)^\perp \iff \sum_{l \in \mathcal{L}} t^*_l v'_l = 0 \iff \sum_{l \in \mathcal{L}} t^*_l \sum_{(p,q) \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}} \delta_{pqrl} h'_{pqr} = 0 \tag{46}$$

$$\iff \sum_{(p,q) \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}} h'_{pqr} \sum_{l \in \mathcal{L}} \delta_{pqrl} t^*_l = 0 \iff \sum_{(p,q) \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}} h'_{pqr} c^*_{pqr} = 0 \tag{47}$$

$$\iff \sum_{(p,q)\in\mathcal{C}} \left[\sum_{r\in\mathcal{R}_{pq}} h'_{pqr}[c^*_{pqr} - \pi^*_{pq}] + \pi^*_{pq} \sum_{r\in\mathcal{R}_{pq}} h'_{pqr} \right] = 0. \tag{48}$$

Now, let us intersect this orthogonal complement with the tangent cone to obtain the feasible set. Using that $\sum_{r\in\mathcal{R}_{pq}} h'_{rpq} = 0$ (by equation (45)), we obtain from equation (48):

$$\sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} h'_{pqr}[c^*_{pqr} - \pi^*_{pq}] = 0. \tag{49}$$

Since $c^*_{pqr} - \pi^*_{pq} = 0$ for all routes such that $h^*_{pqr} > 0$, this becomes:

$$\sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}, h^*_{pqr}=0} h'_{pqr}[c^*_{pqr} - \pi^*_{pq}] = 0. \tag{50}$$

However, by equation (45), $h^*_{rpq} = 0 \implies h'_{rpq} \geq 0$. Also, $c^*_{pqr} - \pi^*_{pq} \geq 0$ by the definition of π, so in fact we have:

$$h'_{pqr}[c^*_{pqr} - \pi^*_{pq}] = 0 \quad \forall r \in \mathcal{R}_{pq} \text{ s.t. } h^*_{pqr} = 0 \quad \forall (p,q) \in \mathcal{C}. \tag{51}$$

So

$$\left\{ h^*_{pqr} = 0 \text{ and } c^*_{pqr} > \pi^*_{pq} \right\} \implies h'_{rpq} = 0 \quad \forall (p,q) \in \mathcal{C}. \tag{52}$$

To summarize, the feasible set for h' is:

$$H' = \left\{ h' \in \Re^{|\mathcal{R}|} \;\middle|\; \begin{array}{l} \sum_{r\in\mathcal{R}_{pq}} h'_{rpq} = 0 \\ h'_{rpq} \geq 0 \text{ if } h^*_{rpq} = 0 \text{ and } c^*_{pqr} = \pi^*_{pq} \\ h'_{rpq} = 0 \text{ if } h^*_{rpq} = 0 \text{ and } c^*_{pqr} > \pi^*_{pq} \\ [r \in \mathcal{R}_{pq},\ (p,q) \in \mathcal{C}] \end{array} \right\} \tag{53}$$

Let us now investigate the auxiliary variational inequality itself, to see what it will be in our traffic assignment framework. Substituting into equation (42), we obtain the problem of finding v' such that

$$(\nabla_\rho t(\rho^*, v^*)\rho' + \nabla_v t(\rho^*, v^*)v')^\top (v - v') \geq 0, \tag{54}$$

for all feasible v (i.e. all v such that there exists $h \in H'$ such that $\Delta h = v$). Using that link travel costs are separable and that each link cost depends on one parameter, we see that this problem is equivalent to the optimization problem

$$\underset{v'}{\text{minimize}} \sum_{l\in\mathcal{L}} \int_0^{v'_l} \left[\frac{\partial t_l(\rho^*_l, v^*_l)}{\partial \rho_l} \rho'_l + \frac{\partial t_l(\rho^*, v^*_l)}{\partial v_l} s \right] ds \tag{55}$$

or, defining $\phi(v')$ as the objective function,

$$\underset{v'}{\text{minimize}}\, \phi(v') = \sum_{l\in\mathcal{L}} \left[\frac{\partial t_l(\rho^*_l, v^*_l)}{\partial \rho_l} \rho'_l v'_l + \frac{1}{2} \frac{\partial t_l(\rho^*, v^*_l)}{\partial v_l} (v'_l)^2 \right], \tag{56}$$

subject to

$$h' \in H', \quad \text{where} \quad v'_l = \sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqrl} h'_{pqr}. \tag{57}$$

Equations (56), (57) above is the sensitivity problem for our traffic network model. We see that this problem is similar our regular traffic equilibrium problems. This similarity will allow us to use the same numerical methods for solving both types of problems.

By Theorem 4, whenever the sensitivity problem has unique solutions, these will be directional derivatives of the corresponding equilibrium quantities with respect to the parameters ρ. However, it is clear that the variables h' may not be unique, since in fact the route flows h themselves are not unique for a traffic equilibrium. Nevertheless, we may note that the objective function in equation (56) contains only v'-variables. Also, we can write

$$V' = \left\{ v' \in \Re^{|L|} \,\middle|\, \exists h' \in H' \text{ with } v'_l = \sum_{(p,q)\in\mathcal{C}} \sum_{r\in\mathcal{R}_{pq}} \delta_{pqrl} h'_{pqr} \right\}, \tag{58}$$

to define a feasible set for v'. As it is of great interest whether the sensitivity problem provides directional derivatives, it would be useful to have a theoretical understanding of when the v'-variables exist uniquely. The following two theorems provide the answer.

Theorem 5 Consider the sensitivity problem (56), (57) for a network satisfying our assumptions. There exists an optimal solution of link flow perturbations v'.

Proof Let

$$\alpha_{max} = \max_{l\in\mathcal{L}} \left| \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l} \rho'_l \right|, \qquad \beta^+_{min} = \min_{l\in\mathcal{L} \text{ s.t. } v_l^*>0} \frac{1}{2} \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial v_l}. \tag{59}$$

Note that $\beta^+_{min} > 0$, since we have strictly increasing link cost functions. Now consider the subset M of the feasible set (58) obtained by adding the constraint

$$\sum_{l\in\mathcal{L}} |v'_l| \quad <= m, \;\; \text{where} \;\; m = \frac{4\alpha_{max}|\mathcal{L}|^2}{\beta^+_{min}} \tag{60}$$

M is clearly non-empty ($v' = 0$ is in it), closed, and bounded. Also, $\phi(v')$ is continuous on M by our assumptions about the link cost functions $t(v)$. Hence, by Weierstrass' Theorem, ϕ attains a minimum on M.

Now consider a solution v' in the feasible set (58) but not in M. Let

$$n = \sum_{l\in\mathcal{L}} |v'_l|. \tag{61}$$

Since the solution is not in M, $n > m$. We have,

$$\phi(v') = \sum_{l\in\mathcal{L}} \left[\left(\frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l} \rho'_l \right) (v'_l) + \frac{1}{2} \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial v_l} (v'_l)^2 \right] \tag{62}$$

$$\geq -\max_{l\in\mathcal{L}}\left|\frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l}\rho_l'\right|\sum_{l\in\mathcal{L}}|v_l'| + \min_{l\in\mathcal{L}\text{ s.t. } v_l^*>0}\left(\frac{1}{2}\frac{\partial t_l(\rho_l^*, v_l^*)}{\partial v_l}\right)\max_{l\in\mathcal{L}\text{ s.t. } v_l^*>0}(v_l')^2 \tag{63}$$

$$= -\alpha_{max}n + \beta_{min}^+\max_{l\in\mathcal{L}\text{ s.t. } v_l^*>0}(v_l')^2 \tag{64}$$

Now, $\sum_{l\in\mathcal{L}} v_l' = 0$ and $v_l^* = 0 \Rightarrow v_l' \geq 0$ (these constraints are implicit in the feasible set (58)), so

$$\sum_{l\in\mathcal{L}\text{ s.t. } v_l^*>0}|v_l'| \geq \frac{n}{2} \Longrightarrow \max_{l\in\mathcal{L}\text{ s.t. } v_l^*>0}|v_l'| \geq \frac{n}{2|\mathcal{L}|}. \tag{65}$$

Hence,

$$\phi(v') \geq -\alpha_{max}n + \beta_{min}^+\left(\frac{n}{2|\mathcal{L}|}\right)^2 = n\left(\frac{\beta_{min}^+ n}{4|\mathcal{L}|^2} - \alpha_{max}\right) \tag{66}$$

$$> n\left(\frac{\beta_{min}^+ m}{4|\mathcal{L}|^2} - \alpha_{max}\right) = n\left(\alpha_{max} - \alpha_{max}\right) = 0. \tag{67}$$

But $v' = 0$ is a point in M for which $\phi(v') = 0$. Hence, for any point outside M (but inside the feasible set), there is a point inside M for which $\phi(v')$ is smaller. Consequently, the minimum attained on M is actually the minimum over the entire feasible set, and thus also an optimal solution to the sensitivity problem. □

Theorem 6 The link flow perturbation solutions v_l', $l \in \mathcal{L}$ are unique for all links l for which the link travel cost functions and equilibrium link flows satisfy

$$\frac{\partial t_l(\rho^*, v_l^*)}{\partial v_l} > 0. \tag{68}$$

The link travel cost perturbations are unique for all links.

Proof Assume that $v'^{(a)}$ and $v'^{(b)}$ are both optimal solutions to (56), (57). By the same type of reasoning as in the proof of Theorem 3,

$$v'^{(c)} = \frac{v'^{(a)} + v'^{(b)}}{2} \tag{69}$$

is then a feasible solution. For brevity of expression define

$$\alpha_l = \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l}\rho_l', \qquad \beta_l = \frac{1}{2}\frac{\partial t_l(\rho_l^*, v_l^*)}{\partial v_l}, \quad \forall l \in \mathcal{L} \tag{70}$$

We can then write the value of the objective function at $v'^{(c)}$ as

$$\phi(v'^{(c)}) = \phi\left(\frac{v'^{(a)} + v'^{(b)}}{2}\right) = \sum_{l\in\mathcal{L}}\left[\alpha_l\left(\frac{v'^{(a)} + v'^{(b)}}{2}\right) + \beta_l\left(\frac{v'^{(a)} + v'^{(b)}}{2}\right)^2\right] \tag{71}$$

$$= \frac{1}{2}\sum_{l\in\mathcal{L}}\alpha_l v_l'^{(a)} + \frac{1}{2}\sum_{l\in\mathcal{L}}\alpha_l v_l'^{(b)} + \frac{1}{4}\sum_{l\in\mathcal{L}}\beta_l(v_l'^{(a)})^2 + \frac{1}{4}\sum_{l\in\mathcal{L}}\beta_l(v_l'^{(b)})^2 + \frac{1}{2}\sum_{l\in\mathcal{L}}\beta_l(v_l'^{(a)}v_l'^{(b)}) \tag{72}$$

$$= \frac{1}{2}\phi(v_l^{\prime(a)}) + \frac{1}{2}\phi(v_l^{\prime(b)}) + \frac{1}{2}\sum_{l\in\mathcal{L}} \beta_l \left[v_l^{\prime(a)} v_l^{\prime(b)} - \frac{1}{2}(v_l^{\prime(a)})^2 - \frac{1}{2}(v_l^{\prime(b)})^2 \right]. \tag{73}$$

Now, since $v^{\prime(a)}$ and $v^{\prime(b)}$ are optimal, we must have

$$\phi(v^{\prime(c)}) \geq \phi(v^{\prime(a)}) = \phi(v^{\prime(b)}). \tag{74}$$

From (73) we thus obtain

$$\sum_{l\in\mathcal{L}} \beta_l \left[v_l^{\prime(a)} v_l^{\prime(b)} - \frac{1}{2}(v_l^{\prime(a)})^2 - \frac{1}{2}(v_l^{\prime(b)})^2 \right] \geq 0. \tag{75}$$

However, $\beta_l \geq 0 \; \forall l \in \mathcal{L}$, and

$$v_l^{\prime(a)} v_l^{\prime(b)} - \frac{1}{2}(v_l^{\prime(a)})^2 - \frac{1}{2}(v_l^{\prime(b)})^2$$

is a quadratic expression with a maximum of 0 at $v_l^{\prime(a)} = v_l^{\prime(b)}$. Hence

$$\beta_l > 0 \Longrightarrow v_l^{\prime(a)} = v_l^{\prime(b)}, \quad \forall l \in \mathcal{L}, \tag{76}$$

so link flow perturbations are unique for all links for which $\beta_l > 0$ and hence for all links for which

$$\frac{\partial t_l(\rho^*, v_l^*)}{\partial v_l} > 0.$$

Furthermore, the link travel cost perturbations

$$t_l'(\rho_l^*, v_l^*) = \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l}\rho_l' + \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial v_l}v_l' \tag{77}$$

are unique for all links since

$$\frac{\partial t_l(\rho_l^*, v_l^*)}{\partial v_l} = 0$$

for any link for which v_l' is possibly non-unique. □

We might want to stop and draw some conclusions from these theorems and their proofs. First, we now know that the sensitivity problem (56), (57) will have a solution that we can find using numerical methods for TAP. However, it should be noted that the proof of Theorem 5 relies on the assumption that link cost functions are strictly increasing. A popular way of modeling certain features of real-life networks is to add links with zero or constant costs. In particular, so-called "centroids", nodes used only to introduce a source of traffic from outside the network (e.g. to model a large residential area) are often implemented by a zero-cost link connection to some part of the network. If we consider such networks, the sensitivity problem is not guaranteed to have a solution.

We can also draw the conclusion that the solution to the sensitivity problem will provide us with directional derivatives for the link flows, at least for those links that have non-zero cost-flow derivatives at equilibrium. This of course includes all links that are used at equilibrium, by our assumption of strictly increasing route cost functions. However, there may be some other links for which there are no unique directional derivatives. Nevertheless, the directional derivatives that are usually most interesting in applications are those of costs, and these will always exist and be provided by our sensitivity analysis.

We end this section with some examples illustrating its somewhat theoretical content in practical terms. We start by looking at an example where we calculate a unique directional derivative. We then look at a case illustrating the possibility of non-unique solutions to the sensitivity problem. Finally, we see how non-strictly increasing cost functions can lead to a situation where the sensitivity problem has no solution.

3.2.1 Example 1 – Braess network

Let us now return to the network of Figure 2 in Section 2.4. Consider the same example but with $t_3(v_3, \rho) = v_3 + 23 + \rho$, where ρ is a variable parameter. For $\rho = 0$, (29) and (30) give the equilibrium solution $v_1 = v_2 = v_3 = v_4 = 3,\ v_5 = 0$ and $h_1 = h_2 = 3,\ h_3 = 0$. Let us now solve the sensitivity problem of calculating the directional derivatives of the link flows in the direction $\rho' = -1$. We have

$$\frac{\partial t_3(v_3,\rho)}{\partial \rho} = 1 \quad \text{and} \quad \frac{\partial t_2(v_2)}{\partial v_2} = \frac{\partial t_3(v_3,\rho)}{\partial v_3} = \frac{\partial t_4(v_4)}{\partial v_4} = 1, \frac{\partial t_1(v_1)}{\partial v_1} = \frac{\partial t_5(v_5)}{\partial v_5} = 10. \tag{78}$$

From (55), (56), (57), we can hence write the problem as

$$\underset{v'}{\text{minimize}} \ -v_3' + \frac{1}{2}[(v_2')^2 + (v_3')^2 + (v_4')^2] + 5[(v_1')^2 + (v_5')^2] \tag{79}$$

subject to (58), which is equivalent to

$$\underset{h'}{\text{minimize}} \ -h_3' + \frac{1}{2}[(h_1')^2 + (h_3')^2 + (h_2')^2] + 5[(h_1' + h_3')^2 + (h_2' + h_3')^2] \tag{80}$$

subject to

$$h_1' + h_2' + h_3' = 0, \quad h_3' \geq 0. \tag{81}$$

By symmetry, $h_1' = h_2'$. We denote this quantity by h_0'. The constraints can then be written $h_3' = -2h_0'$, $h_0' \leq 0$, and the problem is hence equivalent to

$$\underset{h_0'}{\text{minimize}} \ 2h_0' + \frac{1}{2}[6(h_0')^2] + 5[2(h_0')^2] \quad \text{subject to} \quad h_0' \leq 0, \tag{82}$$

which has the solution

$$h_0' = -\frac{1}{13}. \tag{83}$$

The corresponding link flow perturbation solutions are

$$v_1' = v_5' = -\frac{1}{13}, \quad v_2' = v_4' = \frac{1}{13}, \quad v_3' = -\frac{2}{13}. \tag{84}$$

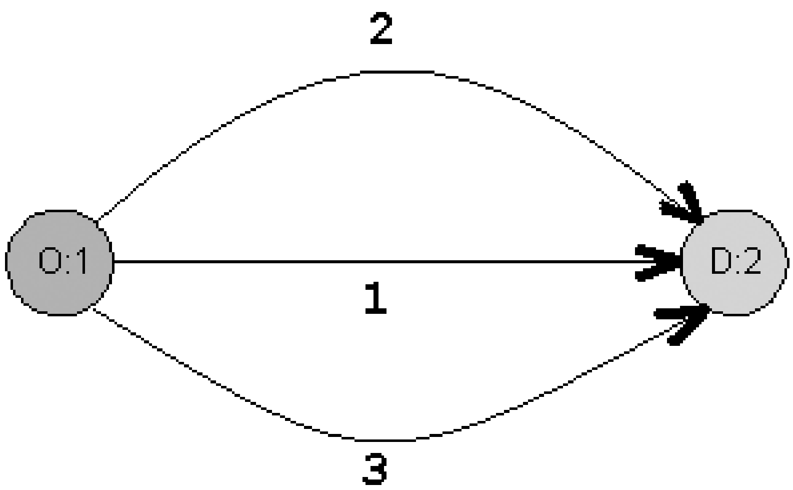


Figure 5: A network

Hence, these are also directional derivatives of the link flows. Note that if instead of $\rho' = -1$ we would have had the direction $\rho' = 1$, the problem would have been

$$\underset{h_0'}{\text{minimize}} \; -2h_0' + \frac{1}{2}[6(h_0')^2] + 5[2(h_0')^2] \quad \text{subject to} \quad h_0' \leq 0, \tag{85}$$

with the solution $h_0' = 0$ and the link flow directional derivatives

$$v_1' = v_2' = v_3' = v_4' = v_5' = 0. \tag{86}$$

Clearly, equilibrium link flows are non-differentiable at the point $\rho = 0$.

3.2.2 Example 2 – Sensitivity problem with non-unique solutions

Consider the network in Figure 5. Let the link cost functions be

$$t_1(\rho, v_1) = v_1^2 + \rho + 9, \qquad t_2(v_2) = v_2^2 + 10, \qquad t_3(v_3) = v_3^2 + 10, \tag{87}$$

where the parameter in the unperturbed case is $\rho^* = 0$ and the demand for the single OD-pair $(1, 2)$ is $d_{12} = 1$. The equilibrium solution is clearly $v_1^* = 1,\ v_2^* = v_3^* = 0$. Let us now solve the sensitivity problem for $\rho' = 1$. We have

$$\underset{v_1', v_2', v_3'}{\text{minimize}} \; \phi(v') = \frac{\partial t_1(\rho^*, v_1^*)}{\partial \rho}\rho' v_1' + \frac{1}{2}\frac{\partial t_1(\rho^*, v_1^*)}{\partial v_1}(v_1')^2 + \frac{1}{2}\frac{\partial t_2(v_2^*)}{\partial v_2}(v_2')^2 + \frac{1}{2}\frac{\partial t_3(v_3^*)}{\partial v_3}(v_3')^2 \tag{88}$$

$$= 1 \cdot 1 \cdot v_1' + \frac{1}{2} \cdot 2 \cdot (v_1')^2 + \frac{1}{2} \cdot 0 \cdot (v_2')^2 + \frac{1}{2} \cdot 0 \cdot (v_3')^2 \tag{89}$$

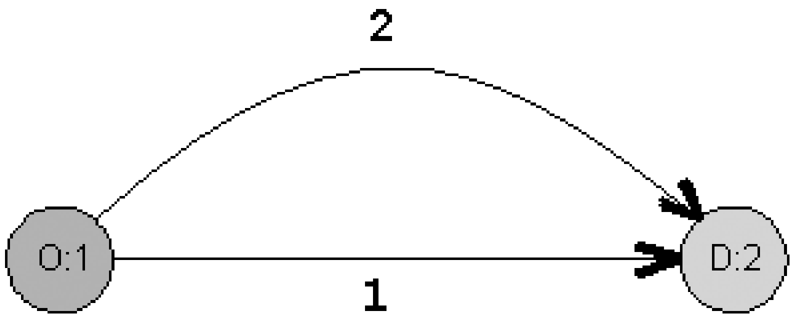


Figure 6: A network

$$= v_1' + (v_1')^2. \tag{90}$$

subject to

$$v_1' + v_2' + v_3' = 0, \quad v_2', v_3' \geq 0. \tag{91}$$

The objective function (90) here does not contain any v_2' or v_3' terms and has a minimum at $v_1' = -\frac{1}{2}$. Hence, any feasible point satisfying $v_1' = -\frac{1}{2}$ is a minimum. The set of optimal solutions can be written

$$v_1' = -\frac{1}{2}, \quad v_2' = a, \quad v_3' = \frac{1}{2} - a, \quad \text{where } 0 \leq a \leq \frac{1}{2}. \tag{92}$$

Clearly, only link 1 has a unique directional derivative in the direction $\rho' = 1$.

3.2.3 Example 3 – Sensitivity problem with no solution

Consider the network in Figure 6. Let the link cost functions be

$$t_1(v_1, \rho) = \rho + 10, \qquad t_2(v_2) = v_2^2 + 10, \tag{93}$$

where the parameter in the unperturbed case is $\rho^* = 0$ and the demand for the single OD-pair $(1, 2)$ is $d_{12} = 1$. The equilibrium solution is clearly $v_1^* = 1,\ v_2^* = 0$. In this problem, the travel cost function for link 1 is not strictly increasing. Let us see what happens when we try to solve the sensitivity problem for $\rho' = 1$. We have

$$\underset{v_1', v_2'}{\text{minimize}}\, \phi(v') = \frac{\partial t_1(\rho^*, v_1^*)}{\partial \rho} \rho' v_1' + \frac{1}{2} \frac{\partial t_1(\rho^*, v_1^*)}{\partial v_1} (v_1')^2 + \frac{1}{2} \frac{\partial t_2(v_2^*)}{\partial v_2} (v_2')^2 \tag{94}$$

$$= 1 \cdot 1 \cdot v_1' + \frac{1}{2} \cdot 0 \cdot (v_1')^2 + \frac{1}{2} \cdot 0 \cdot (v_2')^2 \tag{95}$$

$$= v_1' \tag{96}$$

subject to

$$v_1' + v_2' = 0, \quad v_2' \geq 0. \tag{97}$$

Clearly, there is no solution. $\phi(v') \to -\infty$ as $v_1' \to \infty, v_2' = -v_1'$. No directional derivatives for the link flows exist. Note again, that by Theorem 5, this situation cannot occur if all link cost functions are strictly increasing (as by our assumptions in Table 1 in Section 2.2).

4 Network Design

4.1 Overview

Network design is the optimization of some objective function of the traffic on a network with respect to one or several *design parameters* – attributes of the network that can be altered. Typically, the design parameters are link capacity augmentations, and the optimization is the minimization of some *total cost function*. The problem is then sometimes also referred to as the *capacity enhancement problem*. Normally, the total cost function is the sum of the total travel costs on the network and some function of the capacity augmentations, referred to as the *investment cost function*. The investment cost function is, as the name implies, meant to model the costs of rebuilding or enhancing parts of the network.

4.2 Bilevel programming

The objective functions used in network design are functions of both the design parameters (link capacities) and the traffic on the network, but only the link capacities can be controlled directly. As we know from studying traffic assignment problems, just calculating the network flows given the link capacities requires solving a non-linear optimization problem. Hence, we have a situation where we want to optimize a quantity with respect to the output of another optimization process on our variables. This is an example of the class of problems known as *bilevel programming* (or bilevel optimization) problems.

Conceptually, a bilevel optimization situation can be thought of in game-theoretic terms as a game between a *leader* and a *follower*, each trying to minimize some respective *loss functions*. The leader chooses the values of his variables, which determines the position of the follower. The follower then chooses the values of his own variables so as to minimize his cost function. These variables in turn influence the loss function of the leader (who will have anticipated these values, using his knowledge of the follower's loss function).

In network design, the leader is the network designer, and the follower is the set of users of the network. The loss function of the follower is simply the objective function of the TAP, whereas the loss function of leader is the above mentioned total cost function. In notation, we can write the problem as:

$$\underset{y}{\text{minimize}}\, Z(v(y), y) = \sum_{l\in\mathcal{L}} t_l(v_l(y_l), y_l)v_l(y_l) + \psi(y), \quad \text{subject to} \quad y_l \in [y_l^l, y_l^u], \; \forall l \in \mathcal{L}, \tag{98}$$

where

$$v(y) = \arg\min_{v} \sum_{l\in\mathcal{L}} \int_0^{v_l} t_l(y_l, s)ds, \quad \text{subject to } (7). \tag{99}$$

Here, Z is the total cost function (see above; also called *network design objective function*), and ψ is the investment cost function. y_l^u and y_l^l are upper and lower bounds for the design parameters (link capacity augmentations) y_l. Although we have associated one design parameter with each link l, it is of course possible that only some link capacities can be modified. This can be modeled by letting $y_l^l = y_l^u = 0$ for that particular link. Normally, the investment cost function is assumed to be separable, i.e.

$$\psi(y) = \sum_{l \in \mathcal{L}} \psi_l(y_l). \tag{100}$$

4.3 Solving the network design problem

The network design problem is generally very challenging to solve, and several different algorithms have been used by researchers with varying degrees of success. Some information about these algorithms as well as comparisons of their results to the results of our own method are presented in Section 9.1.

One possible approach to solving the network design problem is to use an algorithm for the TAP together with some search method. Better and better network design solutions can then be generated iteratively. This is the approach that we shall use, and our method of performing sensitivity analysis shall provide the basis for the search direction generation. By solving a sensitivity problem for each link that is considered for a possible capacity augmentation, we can obtain a vector of directional derivatives of Z with respect to the design parameters. Now, this vector will not always be a true gradient, since the network flows (and hence the network design objective function Z) are not everywhere differentiable. Nevertheless, we will use this vector as if it were a gradient. By adding a line search, we then have a steepest descent method for network design. In notation, our algorithm for network design can then be written:

$$y_l^{k+1} = y_l^k - m \left[\frac{\partial Z}{\partial y_l} \right]_{proj.}, \quad \forall l \in \mathcal{L}, \tag{101}$$

where

$$\left[\frac{\partial Z}{\partial y_l} \right]_{proj.} = \begin{cases} \frac{\partial Z}{\partial y_l}, & -\frac{\partial Z}{\partial y_l} \in T_{[y_l^l, y_l^u]}(y_l^k), \\ 0, & \text{otherwise}, \end{cases} \tag{102}$$

and m is a step length determined by the line search (an alternative approach is to use a projection in the line search itself – then it is not necessary to send a maximum step length to the line search). One important consideration for the line search is to give it sufficiently loose stopping criteria. The reason for this is that each function call to calculate Z is quite computationally intensive. Remember, each computation of Z requires solving a TAP. Furthermore, it will be necessary to solve each TAP quite accurately. For more details about the line search used and other parts of the network design implementation, refer to Section 6.3.

5 Algorithms for Traffic Assignment Problems

In this section, we give an overview of two basic types of algorithms that can be used to solve traffic assignment problems: the *Frank-Wolfe algorithm* (and its extensions) and algorithms based on *Disaggregate Simplicial Decomposition* (DSD). The work presented in this paper mainly involves implementation and application of DSD algorithms, although a Frank-Wolfe algorithm was implemented as a comparison. However, the description of the Frank-Wolfe algorithm is mainly included because of historical reasons and because its use is still so wide-spread.

5.1 The Frank-Wolfe algorithm

The Frank-Wolfe algorithm for the traffic assignment problem builds on a general method (due to Frank and Wolfe) for solving convex quadratic programming problems. In the specific case of the TAP, the method gives an algorithm that is easy to implement and makes satisfactory initial progress when it is executed. This Frank-Wolfe algorithm has become very popular both in the industry and among some researchers, and is still widely used.

The basis of the Frank-Wolfe algorithm is the generation of a link-space search direction by the approximation of the problem as a linear programming problem. In network terms, this means that a search direction is generated by assuming fixed link costs. With fixed costs, each OD-pair problem can be considered independently. Each separate problem can be solved by using a method for calculating shortest routes in a graph, such as Dijkstra's algorithm. A line search is then performed to minimize the objective function along the line segment in aggregate link flow space that joins the previous iteration's solution to this fixed-costs solution. Note that the algorithm works entirely in the link-flow space – no route flow solutions are generated.

Termination of the Frank-Wolfe algorithm is decided by approximation of the maximum relative error in the objective function by the calculation of lower bounds. Since the objective function is convex, a lower bound can be obtained in each iteration by extracting the value of the appropriate point on a tangent (hyper-)plane to the objective function.

An initial guess for the Frank-Algorithm can be obtained by calculating shortest routes assuming zero-congestion costs. All flow for each OD-pair is then assigned to the shortest route for that pair.

In notation, the Frank-Wolfe algorithm can be summarized as follows:

- Let v^0 be a feasible link flow solution, obtained e.g. by assigning all flow for each OD-pair to the cheapest route (assuming zero congestion).
- Calculate link costs: $t_l^0 = t_l^0(v_l^0)$.
- Let $k = 0$ and $LBD = 0$. Let ε be the desired maximum relative error in the objective function value.
- Iterate
  - Let y^k be the link flows obtained by assigning all flow for each OD-pair to the cheapest route, assuming fixed link costs t^k.

- Let $p^k = y^k - v^k$. p^k is the generated search direction.
- If $T(y^k) > LBD$, let $LBD = T(y^k)$.
- If $\frac{T(v^k)-LBD}{LBD} < \varepsilon$, terminate with v_k as the solution.
- Calculate the step length l such that $T(v^k + lp^k)$ is minimized for $l \in [0, 1]$.
- Let $v^{k+1} = v^k + lp^k$.
- Calculate link costs: $t_l^{k+1} = t_l(v_l^{k+1})$.
- If $\frac{T(v^k+1)-LBD}{LBD} < \varepsilon$, terminate with $v_k + 1$ as the solution.
- Increment k by one.

5.1.1 Line search for the Frank-Wolfe algorithm

There are many possible line search implementations that can be used in the Frank-Wolfe algorithm. The fact that the objective function is convex and that the directional derivative is easy to calculate can be utilized to implement an efficient search, e.g. a binary search. However, as the lengths of the steps taken in the successive iterations of the algorithm tend to diminish very quickly, it is also important to implement the search with a sensible system for making an initial guess of the step length.

5.1.2 Problems with the Frank-Wolfe algorithm

The Frank-Wolfe algorithm has many disadvantages compared to more advanced algorithms such as DSD algorithms. The biggest problem is that of its slow convergence. As has been noted in previous studies (Larsson and Patriksson 1992) and can be seen from the results in this paper (see Table 11 in Section 7.1.5), the Frank-Wolfe algorithms performs very poorly, especially after the first few iterations. The underlying problem is that the step length very quickly tends towards zero. The slow convergence means that even small problems cannot be solved very accurately using this algorithm. Apart from making determinations of equilibrium flows less accurate, this also means that any applications that utilize the Frank-Wolfe algorithm will suffer in accuracy. For instance, network design methods using this algorithm will, regardless of the methods' other characteristics, be unable to find very accurate design optimums.

Another problem with the Frank-Wolfe algorithm is related to the fact that it works only in the link-flow space. Since no route flows are generated, it is not possible to implement route-cost based termination criteria. A criterion based on route costs is usually seen as natural, since this provides a direct test of to what degree the Wardrop condition is being satisfied. The objective function of TAP is entirely artificial and has no concrete interpretation. Additionally, terminating at a certain maximum relative error of the objective function relies on the determination of a lower bound of this function. Nevertheless, with the Frank-Wolfe algorithm it would be too computationally expensive to calculate a route flow solution at each iteration, so termination criteria based on the objective function value are the only feasible choices.

5.1.3 Extensions of the Frank-Wolfe algorithm

There are several possible extensions of the Frank-Wolfe algorithm that somewhat improve its convergence. One way of improving the algorithm is to replace the objective function-minimizing line search with some heuristic. E.g., a sequence of fixed step lengths can be used.

Another extension involves the use of more than one previous solution in the line search. Two line searches are then performed, first the regular one, and then another one between the new point and the solution from two iterations earlier. I.e., $v^k = v^k_{temp} + m_k(v^{k-2} - v^k_{temp})$, where $v^k_{temp} = v^{k-1} + l_k p^{k-1}$ and m_k and l_k are step lengths. This approach is known as the *parallel tangents* or PARTAN method. An implementation of PARTAN for the traffic assignment problem can be found in the appendix. While the PARTAN method has a slightly quicker convergence than the standard Frank-Wolfe algorithm, the difference is quite small, so the PARTAN-implementation was not used very extensively in our tests.

5.2 The Disaggregate Simplicial Decomposition algorithm

Most of the work behind this paper utilizes DSD-algorithms to solve TAP-based problems. Implementation details for two DSD-type algorithms are presented in Section 6, so the description here will be a general overview.

5.2.1 Column generation

While the Frank-Wolfe algorithm deals only with link flows, the TAP itself is most naturally considered and specified in terms of route flows. The problem with solving TAP over the set of feasible route flows is that the total number of routes is typically much too large to handle. A way to get around this problem is to restrict the problem to a subset of the possible routes. This can be done by including a route into the set of routes considered only once it appears that it should have a positive flow. This process of iteratively generating those routes that are of interest is an example of a type of method known as *column generation* (Patriksson 1994, p. 114).

5.2.2 Restricted master problems

Column generation lets us limit the set of routes over which we search for an optimum, but we still have the problem of finding this optimum. This problem is known as a *Restricted Master Problem* (RMP). The number of variables in a RMP is relatively limited, which is what makes it manageable. However, the difficulty remains that the RMP is not separable with respect to the OD-pairs. Remember, RMP is the same problem as TAP, just restricted to a smaller set of variables.

There are various ways of solving RMPs. An iterative method utilizing a route cost-based search direction generation can be used together with a line search in the link flow space in order to construct a good solver. One thing that makes the search direction generation somewhat easier is the fact that the gradient of the objective function in the route flow space is simply equal to the vector of route costs:

$$\frac{\partial Z}{\partial h_{pqr}} = \sum_{l \in \mathcal{L}} \frac{\partial \int_0^{v_l} t_l(s)\, ds}{\partial h_{pqr}} = \sum_{l \in \mathcal{L}} \frac{\partial v_l}{\partial h_{pqr}} t_l(v_l) = \sum_{l \in \mathcal{L}} \delta_{pqrl} t_l(v_l) = c_{pqr}. \tag{103}$$

As with the Frank-Wolfe algorithm, a good line search in the link flow space is easy to implement, since the objective function is convex.

5.2.3 The algorithm

A *Disaggregate Simplicial Decomposition* (DSD) algorithm is an algorithm that combines the column generation idea with the basis of the Frank-Wolfe algorithm – the decomposition of the complete inseparable problem into independent problems for each OD-pair by approximating link costs as fixed. In the Frank-Wolfe algorithm, the results from the cheapest route calculations are used only to generate a link flow search direction. Here, on the other hand, the routes found are added to the set of routes (columns) over which the restricted master problems are solved.

The algorithm is referred to as *disaggregate* because the variables (routes) for each OD-pair are represented separately. It is called *simplicial* because the RMPs are solved over the simplex-product defined by demand-feasibility. An initial set of columns (routes) to start the algorithm with can be determined by calculating zero-congestion cheapest routes as well as cheapest routes assuming all flow are on the zero-congestion cheapest routes. The basic algorithm can be summarized as follows:

- Calculate the cheapest route for each OD-pair, assuming zero congestion (i.e. $t_l(v_l) = t_l(0)$. Add all these routes to the set of RMP routes.
- Calculate link flows $v_l(h)$, assuming all flows to be on these cheapest routes.
- Calculate link costs: $t_l = t_l(v_l)$.
- Calculate the cheapest route for each OD-pair, assuming fixed link costs t_l.
- Add any of these new cheapest routes not already included in the set of RMP routes to this set.
- Iterate
  - Solve the RMP (i.e. TAP over the set of RMP routes) to the required tolerance.
  - Calculate link flows and link costs using the RMP route flow solution.
  - Calculate the cheapest route for each OD-pair, assuming fixed link costs.
  - Add any of these new cheapest routes not already included in the set of RMP routes to this set. If no new routes were found, terminate.

6 Implementation

All implementation was done in MATLAB version 6.0. While MATLAB code executes relatively slowly, it was chosen for ease of coding and richness of features for analyzing and visualizing the data generated. The MATLAB program code written can be found in the appendix.

6.1 DSD algorithm

Prior to implementing a DSD algorithm-based equilibrium solver, implementations of the Frank-Wolfe algorithm and the PARTAN method were constructed. However, these were written for learning purposes and used only to compare the performance and results with those of the DSD algorithm, so most of the details are omitted here. Nevertheless, we should mentioned that the line search used in both implementations was a bisection (binary) search using the previous iteration's step length as an initial guess.

Two very similar versions of the DSD algorithm implementation were written, using slightly different formats for the travel cost functions. In one version, travel cost functions were assumed to be polynomials, i.e.

$$t_l(v_l) = K_1 + K_2 \cdot v_l + K_3 \cdot v_l^2 + K_4 \cdot v_l^3 + ..., \tag{104}$$

where K_i are constants different for each link travel cost function t_l. The other version of the implementation used travel cost functions that were so-called BPR-functions, i.e. could be written

$$t(v_l) = K_1 + K_2(v_l/K_3)^{K_4}, \tag{105}$$

where, as before, K_i are constants different for each t_l. Here, $K_3 > 0$ for each t_l. The BPR function format was implemented in order to allow direct comparison with other algorithms for sensitivity analysis, many of which use BPR functions with the constant K_3 containing a parameter.

It should be noted here that these cost function formats *do not* in themselves enforce our assumptions about the link travel cost functions (see Table 1 in Section 2.2).

For the BPR-format, an additional variation of the implementation was written in which a guess for the solution was used as an in-parameter. This version was tested as a possible means of enhancing performance of the network design implementation (see Section 9.2.5 for more details).

6.1.1 Data input

Traffic networks were fed to the programs as ASCII text files. For each traffic network, there were three files:

- One file specifying all links in the network and the associated cost data,
- One file specifying all the OD-pairs and the associated travel demands, and
- One file specifying all the locations of the nodes (used for graphing).

For each of the files, one row was used for each item (link, OD-pair, or node) with white-space separating numeric values.

For link data input file, one row was used for each link, with the links numbered according to the order of the input file. For both the travel cost formats (polynomial and BPR functions), the format of the rows in the file was

$$N_1 \quad N_2 \quad K_1 \quad K_2 \quad K_3 \quad K_4 \quad [...]$$

with N_1 and N_2 being the start and end nodes of the link, respectively, and the K's specifying the link cost function of the link (see (104) and (105)).

The rows of the OD-pair input file were formatted

$$N_1 \quad N_2 \quad D,$$

with N_1 and N_2 being the origin and destination nodes, respectively, and D being the OD-pair demand. The rows of the nodes input file were formatted

$$X \quad Y,$$

with X and Y being the x- and y-coordinates, respectively, of the node.

6.1.2 Data types

The set of route flows was represented as a MATLAB cell array. A cell array is an array where each element, called a cell, can be of any data type. In this case, each cell (one for each OD-pair) contained a vector with the flows on each route between the OD-pair. The cell array representation was used since there can be different numbers of routes between different OD-pairs, so the vectors can be of different lengths. The same data type was used for the route costs.

The set of routes itself was represented as a *nested* cell array. Each cell, one for each OD-pair, contained another cell array with a cell for each route. Each of these lower-level cells held a vector with the indecis of the links on the different routes.

Vector representations were used for the link flows and link costs of the links in the network. Also, a vector was used to store the number of routes for each OD-pair.

6.1.3 Settings and constants

A quite general main algorithm was implemented in order to make the DSD-solver flexible. A sequence of master problem error tolerances, specified using a constant vector, was used to successively increase accuracy to the required level. Three other constants were used to customize the program flow of the algorithm:

- 'iterations before failure' : a limit on the maximum number of master problem iterations performed before terminating the master problem solver (a 'failure') and checking for new routes.

- 'maximum failures' : a limit on the maximum number of consecutive failures before terminating the entire DSD solver.
- 'maximum iterations' : a limit on the *total* number of master problem iterations performed before terminating the entire algorithm.

A discussion of the appropriate choices of these constants depending on the networks studied can be found in Section 7.2.2.

6.1.4 Main algorithm

The main algorithm for DSD solver implementation can be summarized, using pseudo code, as follows:

- Calculate cheapest routes for all OD-pairs (assuming zero congestion) and initialize flows to these routes.
- Calculate route costs.
- Check for new cheapest routes. Add those not already included to the set of routes considered.
- Initialize master problem error tolerance to first value in sequence.
- Initialize failures to zero.
- Iterate until failures reaches the maximum value or the total number of iterations reaches the maximum value.
  - Solve restricted master problem (see 6.1.6 below).
  - If the required tolerance was reached:
    * Reset failures to zero.
    * Check for new routes.
    * If no routes are found, decrease tolerance to next value in sequence. If the final tolerance has been reached, terminate.
  - Else:
    * Check for new routes.
    * If no new routes were found, increment failures by one. Otherwise, reset failures to zero.

6.1.5 Route adding algorithm

In order to simplify cheapest-route calculations, a modified node-node incidence matrix A was used. In the non-zero elements of this matrix, the indices of the *cheapest* links between each pair of nodes were held. This matrix was updated and sent to the route-adding function for each route-adding attempt.

To calculate cheapest routes, Dijkstra's algorithm was executed once for each origin in the set of OD-pairs. Dijkstra's algorithm works by assigning a set of temporary labels to each node in the network. The labels represent the length of the shortest route thus far found from the origin to that particular node. In each iteration of the algorithm, the node with the lowest temporary label receives a fixed label of the same value. Any nodes with temporary labels are then checked to see if a shorter route (and thus lower label) can achieved by passing through the node with the newly fixed label. In our case, node-index labels were used in addition to route-length labels so that the indices of the previous nodes on the shortest route to each node could be stored. In this way, any shortest route could later be reconstructed. In other words, each execution of Dijkstra's algorithm generated a vector representing a cheapest tree from one node to all other nodes. The algorithm can be summarized as follows:

- Set all route-length labels to infinity.
- Set all node-index labels to zero.
- Mark all labels as temporary.
- Set route-length label of origin to zero.
- Iterate
  - Mark node with lowest temporary route-length label as the current node.
  - Mark the labels of this node as fixed.
  - Use the matrix A to retrieve the link indices of all cheapest links from the current node to other nodes.
  - Iterate for each such other node.
    - If a lower route-length label results from using the route through the current node, update this label and set the node-index label to the index of the current node.

Cheapest routes between all OD-pairs were reconstructed by traversing the generated shortest route-trees backwards. Each route was then checked to see if it was already included in the 'routes'-variable. Those that were not were added, and the corresponding route flow, route cost, and number of routes variables were updated.

6.1.6 Algorithms for restricted master problems

Three different algorithms were implemented to solve the restricted master problems: A gradient point projection, a gradient projection, and a Newton method. Both the gradient point projection and the Newton method that were used utilize a search direction generation step, a line search performed in the link flow space, and a route cost-based termination criterion. The methods differ only in the search direction generation step – the other steps are identical for both algorithms.

On the other hand, the gradient projection method that was implemented is based on a search along the entire projection arc (whereas the other methods involve searches towards a projected point). This approach turned out to lead to significantly worse performance than the other two, and its implementation was therefore rarely used. Hence, a detailed description of this method is omitted here.

Search direction generation for the gradient point projection method In the gradient point projection method, a search direction is generated by projecting a given point on the objective function gradient onto the feasible space. Since the gradient is in fact the route cost vector (see Section 5.2.3), this amounts to finding the route flow vectors h_{pq} that solve the $|\mathcal{C}|$ separate quadratic programming problems

$$\underset{h_{pq}}{\text{minimize}} \sum_{r\in\mathcal{R}_{pq}} \left[(h^k_{pqr} - \frac{1}{a}c_{pqr}) - h_{pqr}\right]^2 \tag{106}$$

$$\text{subject to} \quad \sum_{r\in\mathcal{R}_{pq}} h_{pqr} = d_{pq}, \quad h_{pqr} \geq 0, \quad \forall r \in \mathcal{R}_{pq}, \tag{107}$$

where a is a positive constant ($a = 1$ was used – tests later showed that changing this value did not provide any performance advantage). Each problem can be solved by using a Lagrange multiplier λ for the demand feasibility constraint. Expressing the Lagrangian G, the problem becomes

$$\underset{h_{pq}}{\text{minimize}} \sum_{r\in\mathcal{R}_{pq}} \left[h^k_{pqr} - h_{pqr} - \frac{1}{a}c_{pqr}\right]^2 + \lambda\left(\sum_{r\in\mathcal{R}_{pq}} h_{pqr} - d_{pq}\right) \tag{108}$$

$$\text{subject to} \quad h_{rpq} \geq 0, \quad \forall r \in \mathcal{R}_{pq}. \tag{109}$$

The solution can then be found from the relations

$$\begin{cases} \dfrac{\partial G(h_{pqr},\lambda)}{\partial h_{pqr}} = 0 \quad \text{or} \quad \left[h_{pqr} = 0 \text{ and } \dfrac{\partial G(h_{pqr},\lambda)}{\partial h_{pqr}} \geq 0\right], & \forall r \in \mathcal{R}_{pq} \\ \dfrac{\partial G(h_{pqr},\lambda)}{\partial \lambda} = 0, & \forall r \in \mathcal{R}_{pq} \\ h_{pqr} \geq 0, & \forall r \in \mathcal{R}_{pq} \end{cases} \tag{110}$$

$$\Longleftrightarrow$$

$$\begin{cases} h_{pqr}(\lambda) = \left\{ \begin{array}{ll} h^k_{pqr} - \dfrac{\lambda}{2} - \dfrac{1}{a}c_{pqr}, & \lambda \leq 2(h^k_{pqr} - \dfrac{1}{a}c_{pqr}) \\ 0, & \text{otherwise} \end{array} \right\}, & \forall r \in \mathcal{R}_{pq} \\ \displaystyle\sum_{r\in\mathcal{R}_{pq}} h_{pqr} = d_{pq} \end{cases} \tag{111}$$

If we disregard the demand feasibility constraint, the other relations in the equation above can be seen as a vector-valued function $h_{pq}(\lambda)$. Note that this function is decreasing in each variable h_{pqr}. It is also piecewise linear, with the breakpoints at $\lambda = 2(h^k_{pqr} - \frac{1}{a}c_{pqr})$. Hence, we can determine the value of λ that satisfies the demand feasibility constraint and thus gives the solution h_{pq} by finding:

1. The smallest breakpoint for which the sum of the route flows is still smaller than the demand, and
2. The largest breakpoint for which the sum of the route flows is still greater than the demand,

and performing a linear interpolation between the two points. In our implementation, this was done by sorting the breakpoints and performing a bisection search. The basic algorithm used was the following:

- Calculate point on gradient.
- Calculate breakpoints.
- Sort breakpoints.
- If setting $\lambda =$ smallest breakpoint gives a total flow less than the demand:
  - Use $\lambda =$ smallest breakpoint - missing flow / number of routes.
- Else:
  - Set upper bound to the number of routes.
  - Set lower bound to 2.
  - Iterate while bounds are different
    - Set index = (upper bound + lower bound) / 2, rounded down.
    - Calculate the total flow resulting from setting $\lambda =$ the breakpoint of this index.
    - If this total flow is greater than the demand, set the lower bound to the index + 1. Otherwise, set the upper bound to the index.
  - Set the index equal to the upper bound.
  - Calculate λ by interpolating between the breakpoint of the index and the next greater breakpoint.

Calculate the search direction using λ and equation (111).

Search direction generation for the Newton method The Newton method that was used is methodologically very similar to the projected gradient point method described above, but is based on another optimization problem. In the Newton method, the values on the diagonal of the Hessian matrix of the objective function are used in generating the search direction. The optimization problem used in our implementation was that of finding the route flow vectors h_{pq} that solve the $|\mathcal{C}|$ separate quadratic programming problems

$$\underset{h_{pq}}{\text{minimize}} \sum_{r \in \mathcal{R}_{pq}} \left[c_{pqr}(h_{pqr} - h_{pqr}^k) + \frac{1}{2} c'_{pqr}(h_{pqr} - h_{pqr}^k)^2 \right] \tag{112}$$

$$\text{subject to} \quad \sum_{r \in \mathcal{R}_{pq}} h_{pqr} = d_{pq}, \quad h_{pqr} \geq 0, \quad \forall r \in \mathcal{R}_{pq}. \tag{113}$$

Note that this problem is actually identical to (106),(107) if the constant a is replaced by the different values c'_{pqr} in each equation and the objective function value is multiplied by a constant. By the same reasoning as in the case of the projected gradient point method, the solution here is therefore

$$\left\{ \begin{array}{l} h_{pqr}(\lambda) = \left\{ \begin{array}{ll} h_{pqr}^k - \dfrac{\lambda + c_{pqr}}{c'_{pqr}}, & \lambda \leq c'_{pqr} h_{pqr}^k - c_{pqr} \\ 0, & \text{otherwise} \end{array} \right\}, \qquad \forall r \in \mathcal{R}_{pq} \\ \displaystyle\sum_{r \in \mathcal{R}_{pq}} h_{pqr} = d_{pq} \end{array} \right. \tag{114}$$

The breakpoints in this case are therefore $\lambda = c'_{pqr} h_{pqr}^k - c_{pqr}$.

The Newton method was implemented using an algorithm very similar to the one used in the projected gradient point method. However, a few implementation details should be mentioned. First, it is necessary to implement extra code for the problematic special case that one or more of the route cost derivatives is zero. There are several ways to handle this case. One approach sometimes used when implementing methods of this kind is to alter the problem slightly by replacing the actual values of all c'_{pqr} smaller than some constant by this constant. Then exactly the same code can be used as was used in the projected gradient point case. If one instead wishes to still solve the optimization problem exactly as stated, it is relatively simple to code a special solver for the case that one of the c'_{pqr} is zero. Both these approaches were tested, and the second one was finally chosen, although the performance difference was quite small.

Another point worth mentioning is that the route cost derivatives needed by the Newton method were obtained by first calculating link cost derivatives. These were then passed on to each iteration (one for each OD-pair) of the search direction generation algorithm, in which the route cost derivatives were then calculated.

Line search implementation The line search was performed in the total link flows space in order to reduce the number of variables and thus make it faster. The generated search direction was converted to the corresponding link search direction using the 'routes' cell array (which is in fact a compact representation of the link-route incidence matrix). A check was implemented to terminate the algorithm whenever the link search direction turned out to be a zero vector. This will happen on the first iteration if one attempts to solve a problem in which congestion is not great enough to affect travelers' route choices.

The line search itself was implemented as a bisection search utilizing the convexity of the objective function. The directional derivative of the objective function in the search direction was calculated using that the gradient is equal to the link cost vector. The greatest feasible step length was calculated and used as the initial upper bound. The minimum of the upper bound and the length of the step taken in the previous iteration was used as an initial guess. Prior to the bisection procedure, an initial check was made whether the longest feasible step was in fact the optimal step. This check was implemented in order to avoid many unnecessary iterations in such cases. The error tolerance (maximal deviation from the optimal step length) of the line search was implemented as a constant in the main program and was passed to the line search function for each execution. It was found that quite a strict tolerance worked well, performance-wise (see Section 7.2.4).

Termination criterion It is natural to use a route cost-based termination criterion in an implementation of the DSD algorithm as both route cost data and route flow data are easily available. Several criteria were tried, e.g. the maximum and average relative route costs differences. The termination criterion finally chosen was the following:

$$S = \frac{\sum_{pq \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}} [c_{pqr} - \min_i(c_{pqi})] \cdot h_{pqr}}{t(v)^\top v} \quad \leq S_\varepsilon, \tag{115}$$

where the v_l have been calculated from the route flow solution and S_ε is the error tolerance. S is called the tolerance coefficent (or the termination coefficent). The advantage of this termination criterion is that it takes into account the relative amount of traffic on different routes in the network when determining how far from equilibrium a solution is. In other words, many travelers using a route slightly more expensive than the cheapest is considered a bigger problem than a few travelers using such a route. This is of course natural. On the other hand, it should be noted that there is nothing in this termination criterion that makes sure that the set of routes used at termination is equal to the set of routes used at equilibrium. On the contrary, a nearly unused route slightly more expensive than the cheapest route will be considered a minor deviation from equilibrium. Since the set of routes used at equilibrium is a factor in sensitivity analysis, it is critical to determine that set whenever solving an equilibrium problem in order to perform sensitivity analysis. In that case, one must make sure that the algorithm is such that the set of used routes has converged by the time the termination criterion has been satisfied. See the discussion in Section 7.2.5 for more details.

6.2 Sensitivity analysis

The similarity of the sensitivity problem to the equilibrium problem meant that much of the code from the DSD algorithm implementation could be reused. Of course, in the sensitivity context, flow and cost derivatives take the place of flows and costs themselves. For simplicity, these derivatives can be considered 'virtual' costs and flows. The restricted master problem solver code then only had to be altered slightly to allow the subset of the routes that were used at equilibrium to take on negative 'flow' values.

As in the DSD algorithm case, two slightly different versions of the sensitivity analysis implementation were constructed, one for each of the two link cost function formats considered. For the BPR link cost function format, the parameter for each link was assumed to be a capacity augmentation (in the BPR format, the denominator of the link flow is known as the *capacity*), i.e.

$$t_l(v_l) = K_{1l} + K_{2l} \left(\frac{v_l}{K_{3l} + \rho_l} \right)^{K_{4l}}. \tag{116}$$

for each link l.

As seen by comparing the objective function of the sensitivity problem (55) to that of the TAP (7), the virtual link cost functions for the sensitivity problem are linear functions

$$t_l^{virtual}(v_l^{virtual}) = \frac{\partial t_l(\rho^*, v_l^*)}{\partial v_l} \cdot v_l^{virtual} + \frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l} \rho_l', \tag{117}$$

which meant that either of our link cost formats implementations for the DSD algorithm could be used for the sensitivity analysis. We chose to use the same format in the solver algorithm as in the data input, i.e. we stored the virtual cost functions in the same format as the real cost functions.

6.2.1 Data input

In the sensitivity analysis implementation, an additional text file was used to input parameter data. This file had one row for each link-parameter pair. The rows of this file had the formats

$$R \quad T, \qquad \text{and} \qquad R,$$

in the polynomial link cost function and BPR link cost function cases, respectively. In each row, R specified the parameter perturbation (i.e. ρ_l' for the link l), and T specified the partial derivative of the link cost function with respect to the associated parameter (i.e. $\frac{\partial t_l(\rho_l^*, v_l^*)}{\partial \rho_l}$ for the link l). Note that no T was necessary in the BPR function case because the assumption (116) lets this derivative be calculated from just the link cost data and equilibrium link flow.

6.2.2 Data types

The variables that were used in the sensitivity analysis implementation are completely analogous to the ones used in the DSD algorithm. The only difference is that an additional vector was used to store, for each OD-pair, the number of routes that were used in the equilibrium solution. An additional requirement for the 'routes' cell array was that those routes that *were* used at equilibrium were always stored first in each sub-cell array (i.e. for each OD-pair), so that would be easy to separate the two subsets of routes.

6.2.3 Main algorithm

The results from the equilibrium solver (set of routes, flows, costs) was passed to the main algorithm of the sensitivity analysis. The 'virtual' link cost function was constructed using the data from the parameter data input file. The set of routes was then rebuilt, including only those routes in the cell array that were used in the equilibrium solution. In order to compensate for the possibility that the equilibrium solution might not perfectly identify these routes (see Section 7.2.5 for a discussion of this problem), a 'fuzz'-factor was used. In other words, to determine whether a route was used, the route flow was compared not to zero but to a very small positive number, obtained by multiplying the OD-pair demand by a tiny factor.

The number of routes used for each OD-pair was stored in a vector. A route generating algorithm was then executed to add all those unused routes to the new cell array that had equal cost to the cheapest routes in the equilibrium solution. The virtual route costs (in fact the route cost derivatives) were calculated for all the routes. A single execution of the restricted master problem code was then executed to solve the problem. Note that this single execution is sufficient the sensitivity case, since all those routes that are allowed to be used are known initially.

The basic algorithm for the sensitivity analysis implementation was the following:

- Calculate virtual link cost function using parameter data.
- Iterate for each OD-pair.
  - Find the indices of all routes used at equilibrium.
  - Put these routes in the new route variable
  - Store the number of such free routes in a vector.
  - Generate all other routes as cheap as the used routes.
  - Add these routes to the new route variable and store the total number of routes for the OD-pair in a vector.
- Calculate virtual link and route costs.
- Solve the restricted master problem over the set of routes.

6.2.4 Route generating algorithm

The route generating algorithm used was a recursive depth-first tree search. In order to make the implementation more efficient, shortest route-length labels obtained by executing Dijkstra's algorithm (once for each origin) were used to terminate the instance of the search whenever the thus-far taken route plus the least cost remaining exceeded the 'target' route cost for the OD-pair. In order to allow for slight route cost differences in the approximate equilibrium solution, the 'target' route cost was set to the minimum route cost for the OD-pair multiplied by a small 'fuzz'-factor.

After the generation process itself, each of the generated routes was compared with the equilibrium-used routes previously stored to make sure that it was not already included. Those routes that were new were added to the 'routes' variable.

6.2.5 Algorithm for the restricted master problem

The restricted master problem code was nearly exactly the same as in the DSD algorithm case. The only difference was that the search direction generation was altered to allow the 'free' routes to take on negative virtual route flow values (corresponding to negative route flow derivatives). Also, a different termination criterion had to be used (see discussion under 'termination criterion' below).

Search direction generation The gradient point projection method was used. The method has to be modified slightly to allow the flows on all the 'free' routes (those routes used at equilibrium) not to be constrained by non-negativity. As a result, the solution relations for the search direction generation are in this case given by (again, considering the derivative quantities actually involved as virtual costs and flows):

$$\begin{cases} h_{pqr}(\lambda) = \left\{ \begin{array}{ll} h_{pqr}^k - \dfrac{\lambda}{2} - \dfrac{1}{a} c_{pqr}, & \lambda \leq 2(h_{pqr}^k - \dfrac{1}{a} c_{pqr}) \\ 0, & \text{otherwise} \end{array} \right\}, & \forall r \in \mathcal{R}_{pq}^{\text{non-free}} \\ h_{pqr}(\lambda) = h_{pqr}^k - \dfrac{\lambda}{2} - c_{pqr} & \forall r \in \mathcal{R}_{pq}^{\text{free}}, \\ \displaystyle\sum_{r \in \mathcal{R}_{pq}} h_{pqr} = 0 & \end{cases} \tag{118}$$

There are hence no breakpoints for the flows on the free routes. So the only implementation modifications needed from the DSD algorithm code is to set the demand to zero and not include any breakpoints for the free variables. This was done by using an indexing vector to access the non-free routes. Since the free routes were all in the beginning of the 'routes' cell array, the indexing vector was simply the indices from the first non-free route to the end of the array.
In order to speed up the code, an extra check was implemented prior to the search generation to treat the special case when all routes are free. In such cases, the Lagrange multiplier for the OD-pair is simply the average of the route costs.

Termination criterion All normal traffic equilibrium termination criteria are unsuitable for the sensitivity problem. The reason for this is that the virtual costs can be zero at equilibrium, so it is very difficult to construct a relative measure that is certain to converge when the problem is solved accurately. Because of this, the following tolerance coefficient was used:

$$\mathcal{S} = \frac{\sum_{pq \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}^{\text{live}}} c_{pqr} - \min_i(c_{pqi})}{\sum_{pq \in \mathcal{C}} \sum_{r \in \mathcal{R}_{pq}^{\text{live}}} c_{pqr}^0 - \min_i(c_{pqi}^0)}, \tag{119}$$

where the 0 superscript indicates initial values and $\mathcal{R}_{pq}^{\text{live}}$ is the set of free routes plus the set of non-free routes with positive (virtual) flow. One strength of this measure is that it is independent of the magnitude of the parameter perturbation used. In practice, it seemed to work very well (see Section 8.2 for a discussion).

6.3 Network Design

The network design problem solver was implemented using the DSD algorithm TAP solver and sensitivity analysis implementations. The BPR function link cost function format was used in this implementation, with the design parameters present in the link capacities, i.e.

$$t_l(v_l) = K_{1l} + K_{2l}(v_l/(K_{3l} + \rho_l))^{K_{4l}}, \tag{120}$$

where K are constants ($K_3 > 0$).

The set of links to be subject to the optimization could be reduced by setting the maximum and minimum capacity increase bounds to equal values, i.e. $y_l^l = y_l^u$ (refer back to Section 4.2 for a reminder of the network design problem).

In this section, the following aspects of the implementation will be discussed in turn: data input, the main algorithm used, the line search method employed, the termination criterion applied, and some performance improvements implemented.

6.3.1 Data input

In the network design implementation, a special text file was used in addition to the network-defining files listed in Section 6.1.1. The file had one row for each link with the format

$$L \quad U \quad C_1 \quad C_2 \ldots.,$$

with L and U specifying the minimum and maximum capacity adjustments (i.e. y_l^l and y_l^u for the link l) and the values C specifying the investment cost function ψ_l for the link on polynomial form (C_1 being the first-degree term).

6.3.2 Main algorithm

The following main algorithm was used.

- Initialize the parameters ρ (usually to zero).
- Iterate
  - Call the DSD-solver to solve TAP for the network with the current parameters.
  - Calculate the objective function value.
  - Calculate the 'gradient' (i.e. directional derivatives of objective function w.r.t. to all non-fixed parameters).
  - Project the gradient unto feasible space assuming an arbitrarily short step.
  - Normalize the gradient.
  - Calculate the maximum allowed step length.
  - Determine a satisfactory step length using an Armijo line search.

6.3.3 Line search method

An Armijo line search was implemented to determine satisfactory step lengths at relatively low computational costs. The basis of this method is to determine whether a step length is 'good enough' by testing whether the objective function value at the new point is less than or equal to the one at the old point minus the scalar product of the gradient and the step vector, multiplied by a factor. In general, this condition is

$$f\left(x^k - m \cdot p^k\right) \leq f(x^k) - \tau \cdot m \cdot \nabla f(x^k)^\top p^k, \tag{121}$$

where p^k is the search direction. In our case, the condition becomes

$$Z\left(y^k - m\left[\frac{\partial Z(y^k)}{\partial y}\right]_{proj.}\right) \leq Z(y^k) - \tau \cdot m\left[\frac{\partial Z(y^k)}{\partial y}\right]^\top \left[\frac{\partial Z(y^k)}{\partial y}\right]_{proj.}, \tag{122}$$

where τ is called the *Armijo factor* and m, as in Section 4, is the step length. While no step length satisfying this condition has been found, the step length is divided by a fixed value (the *Armijo multiplier*, κ) until the condition is satisfied. However, if the very first step length tried satisfies (122), the step length is instead multiplied by κ until the condition is no longer satisfied. Finally, the longest step length satisfying the condition is the actual step length used.

6.3.4 Termination criteria

The algorithm was terminated whenever no step could be taken or the step was so short that the new parameter values were equal to the old ones to the numerical precision available (i.e. Matlab's 'double' type). This happens sooner than might be expected for many problems. However, for large problems, it may be better to use a less strict termination criterion, e.g. a fixed limit on the number of main iterations.

6.3.5 Performance improvements

A discussion of the effects of these implementation modifications can be found in Section 9.2.5.

Parameter 'freezing' It was noticed that, in many executions, some parameters quickly reached a bounding value and remained at that value. To take advantage of this, a method was implemented by which the directional derivatives of these parameters needed not be calculated in every single iteration. This worked in the following way: Whenever a parameter remained at a bounding value for a certain number of iterations, it was 'frozen' for another fixed number of iterations before being included again. While frozen, the parameter was treated as a fixed parameter. Hence, using this mechanism allowed a reduction of CPU-time consumption while still allowing the frozen parameters occassional opportunities to change their values.

Variable DSD-solver accuracy In order to reduce the computational burden of multiple DSD-algorithm executions during the Armijo line searches, a mechanism was implemented by which the strictest error tolerance for the DSD-solver was used only at the final execution of each main iteration. During the Armijo line searches, lower accuracy solutions were calculated since only the objective function values of each point in parameter space was of interest.

DSD-solver 'hot start' In order to further speed up DSD-calculations, a mechanism was tried by which an older DSD-solver solution (e.g. for the previous parameter values) was used as a starting guess for each DSD-solver execution.

7 Results - Algorithm Performance of TAP Solvers

7.1 Results of tests on well known networks

The solvers were run using MATLAB 6 on a Windows XP platform with an AMD Athlon XP2000+ 1.63 GHz processor. This section presents the results of several executions for each of a number of well known networks. The networks are presented in order of complexity, with the smallest networks first. For each network, there is a table showing the settings for each execution. Refer back to Section 6.1.3 for a reminder of what these settings mean.

In a second table for each network, the results of the executions are presented. For most networks, results from Larsson and Patriksson (1992) are included for comparison. In their paper, Larsson and Patriksson used a DSD algorithm with slightly different RMP algorithms than we have implemented.

Although most of the items in the tables of results should be self-explanatory, there are some that we should discuss. The 'relative feasibility error' of a solution is the sum of the absolute values of the differences between the sums of OD-pair flows and the corresponding OD-pair demand, divided by the total demand:

$$F = \frac{\sum_{(p,q)\in\mathcal{C}} \left|\sum_{r\in\mathcal{R}} h_{pqr} - d_{pq}\right|}{\sum_{(p,q)\in\mathcal{C}} d_{pq}}. \tag{123}$$

This is hence a measure of how much a solution deviates from the feasible set. Because of the limited numerical precision of computer arithmetic, tiny feasibility errors are inevitably introduced throughout each execution. In order to make the objective function values and other characteristics of the solutions that we present as accurate as possible, a correction algorithm (in essence, a projection unto the feasible set) was applied to each final solution. Hence, two quantities are presented for each solution: the relative feasibility error of the final solution and the relative feasibility error of the uncorrected solution. The objective function value of each uncorrected solution is presented as well. For a further discussion of the effects of feasibility errors and the procedure used to correct them, see Section 7.2.1.

In addition to the table of results, two diagrams illustrating the convergence characteristics of the different executions are shown for each network tested. One diagram shows objective functions values as a function of CPU-time. These diagrams clearly illustrate the initial progress of each execution. However, it should be understood that the long-run convergence trends are not easily visible in these diagrams. For this reason, most of them have been cut off at CPU-time values significantly before termination of the executions.

The second diagram for each network shows the maximum relative error of the objective function values as a function of CPU-time. Note that the best lower bounds (from all the executions) have been used to calculate these errors. In these diagrams, the long-run convergence trends are much easier to spot.

7.1.1 Nguyen/Dupuis network

This is a small hypothetical network consisting of 13 nodes, 19 arcs, and 4 OD-pairs. It was first discussed in Nguyen and Dupuis (1984). Table 2 shows the solver settings used for the executions on this network, and the results are summarized in Table 3. The time-convergence characteristic of the algorithm executions is illustrated in Figure 7.

For this network, the Newton-based algorithm has difficulties and does not reach the final required tolerance. The reason for this is lack of numerical precision. As can be seen from the table, the solution in the Newton case eventually becomes quite infeasible. Not surprisingly, this stops the algorithm from achieving a very accurate final solution. It is interesting that the Newton-based method suffers from numerical errors here whereas the gradient-based method seems much more robust. Possibly this is not algorithm-related but due to the specific implementations. Also, we should point out that the accuracy reached by the Newton method is still very high – it is just that this network is so small and simple that, without the precision problem, it would be computationally realistic to calculate the solution to a much higher accuracy.

Note also that the uncorrected objective function values are reported as *lower* than the lower bound. This demonstrates the fragility of objective function value measures. When a problem is solved to a great level of precision, slight feasibility errors can be large enough to have a noticeable effect on the apparent objective function values. This is because a slight deficit in the total traffic for an OD-pair will reduce the objective function value significantly. On the contrary, small feasibility errors are unlikely to affect route-cost based termination coefficients, since these are not so sensitive to the total traffic. This is one reason why a route cost-based criterion like the one we are using is superior to any criterion based on the objective function value.

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	20	50
Newton / 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	20	50

Table 2: Settings for the Nguyen/Dupuis network.

	Gradient P.P.	Newton	Larsson/Patriksson
CPU-time (s)	1.493	8.382	N/A
Line searches	326	2250	50
$S(v)$	6.3361e-11	1.1583e-7	?
$T(v)$	85028.07173509084	85028.07173509499	85028.07(5) [rounded]
Lower bound on $T(v)$	85028.07172871966	85028.06008779377	85028.03
Max. rel. error (using best lbd)	7.493e-11	7.498e-11	3.85e-8
Avg. rel. route cost diff.	6.2245e-11	2.9977e-8	?
Routes generated	11	11	10
Routes used by solution	9	9	?
Avg cost at solution	50.2766	50.2766	?
Rel. feas. error of solution	0	0	?
Rel. feas. error before corr.	3.9597e-12	6.8551e-7	N/A
$T(v)$before corr.	85028.07173469731	85028.01073615927	N/A

Table 3: Results for the Nguyen/Dupuis network.

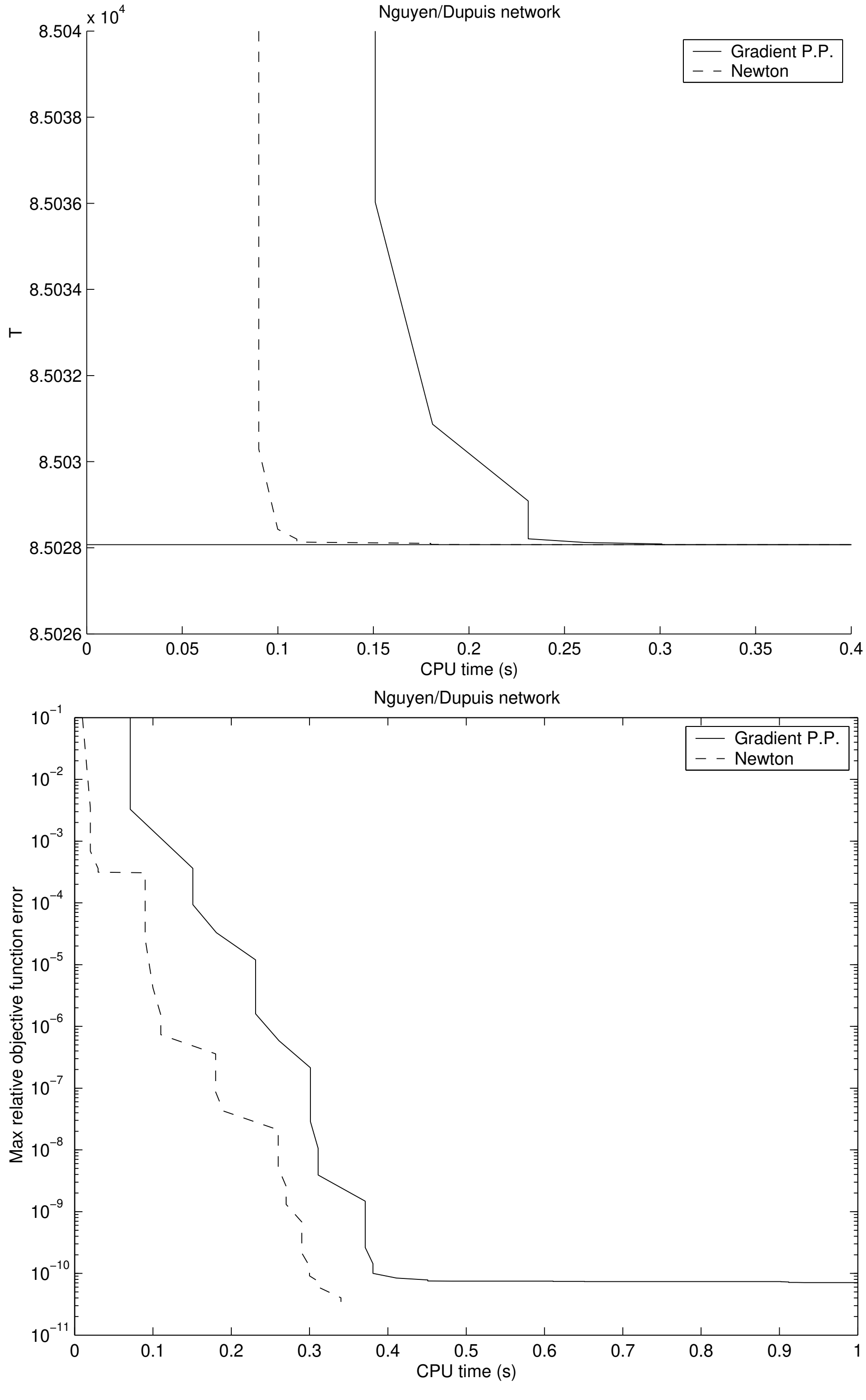


Figure 7: Algorithm performance on the Nguyen/Dupuis network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors. The graphs of the Newton-based algorithm's performance should be viewed with skepticism as they show the objective function value of a solution that (at least eventually) has a significant amount of infeasibility.

7.1.2 Steenbrink network

This is a small hypothetical network consisting of 9 nodes, 36 arcs, and 12 OD-pairs. It was first discussed in Steenbrink (1974). Table 4 shows the solver settings used for the executions on this network, and the results are summarized in Table 5. The time-convergence characteristic of the algorithm executions is illustrated in Figure 8.

We see here that the Newton-based algorithm again performs significantly worse than the Gradient-based algorithm. As in the Nguyen/Dupuis network, the Newton implementation has a problem with feasibility due to numerical imprecision.

By looking at Table 5, we can note another weakness of using the objective function value as a measure of the accuracy of a solution. The termination coefficient S and even the objective function lower bound differ quite a lot between the solutions obtained by the different executions, whereas the objective function values differ very little. In other words, the objective function values fail to indicate the real differences in the accuracy of the different solutions that are apparent by examining the other measures. Clearly, we have made the right choice in using a route cost-based termination criterion.

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8	100	50
Gradient P.P./ 2	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50
Newton / 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50

Table 4: Settings for the Steenbrink network.

	Gradient P.P./1	Gradient P.P./2	Newton	Larsson/Patriksson
CPU-time (s)	0.891	32.076	29.943	N/A
Line searches	52	5833	5325	37
$S(v)$	6.3178e-9	9.5928e-11	3.3092e-7	?
$T(v)$	16957.67466437824	16957.67466437823	16957.67466438163	16957.674664378
Lower bound on $T(v)$	16957.67449395362	16957.67466179057	16957.66573784799	16957.673978581
Max. rel. error (using best lbd)	1.526e-10	1.526e-10	1.528e-10	2.25e-9
Avg. rel. route cost diff.	5.3201e-9	1.2499e-10	9.0387e-8	?
Routes generated	35	37	35	?
Routes used by solution	29	31	29	?
Avg cost at solution	2.7248	2.7248	2.7248	?
Rel. feas. error of solution	0	0	2.2967e-17	?
Rel. feas. error before corr.	2.9596e-14	2.6196e-10	8.1895e-8	N/A
$T(v)$before corr.	16957.67466437758	16957.6746605544	16957.67270297324	N/A

Table 5: Results for the Steenbrink network.

7.1.3 Harker and Friesz network

This is a small hypothetical network consisting of 6 nodes, 16 arcs, and 2 OD-pairs. It was described in Harker and Friesz (1984). A diagram of the network is shown in Figure 9.

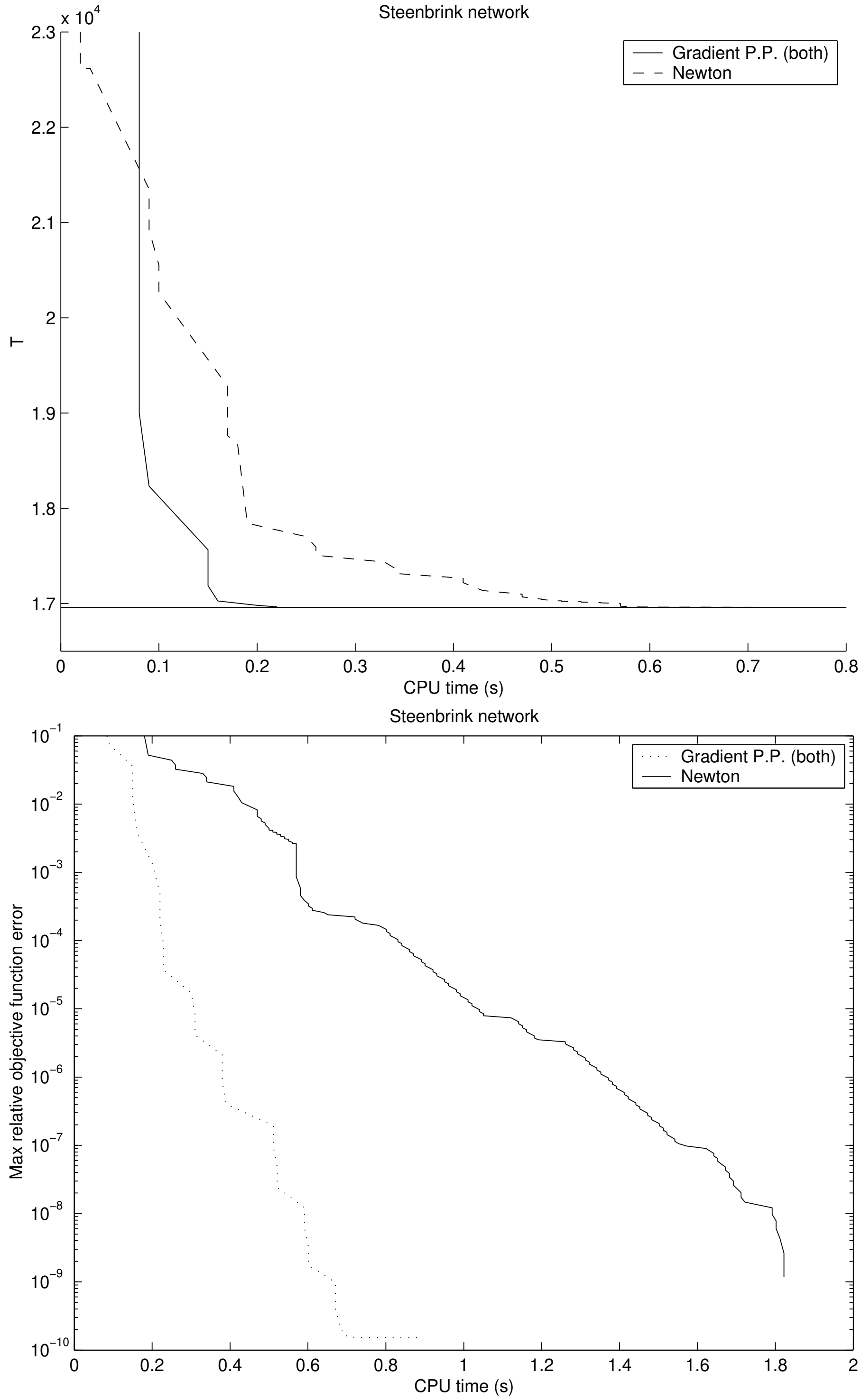


Figure 8: Algorithm performance on the Steenbrink network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors. The graphs of the Newton-based algorithm's performance should be viewed with some skepticism as they show the objective function value of a solution that (at least eventually) has a significant amount of infeasibility.

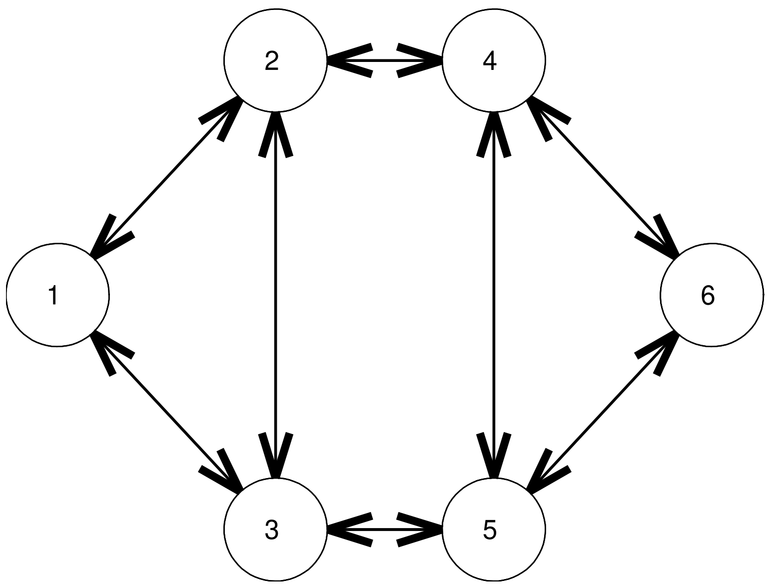


Figure 9: The Harker and Friesz network.

Slightly different demand scenarios for this network have been referenced in previous publications. In this test, the OD-pair demands used were $d_{1,6} = 5$ and $d_{6,1} = 5$. Table 6 shows the solver settings used for the executions on this network, and the results are summarized in Table 7. The time-convergence characteristic of the algorithm executions is illustrated in Figure 10.

The results for this network show that here, the Newton method does not have the same problem with feasibility errors as for the previous two networks. Furthermore, we see that the performance of the Newton method is approximately the same as that of the gradient-based method (the "Gradient P.P." execution terminates slightly earlier, but with a slightly less precise solution).

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50
Newton / 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50

Table 6: Settings for the Harker and Friesz network.

7.1.4 Hull network

This is a 501-node, 798-arc, 142-OD-pair network modeling the real-life city of Hull, Canada. Table 8 shows the solver settings used for the executions on this network, and the results are summarized in Table 9. The time-convergence characteristic of the algorithm executions is illustrated in Figure 11.

This network is *much* larger than the previous ones, but both algorithm were still able to find extremely accurate solutions. The execution of the gradient-based algorithm took much longer to terminate, however. We see from the graphs that, although the Newton-based algorithm did

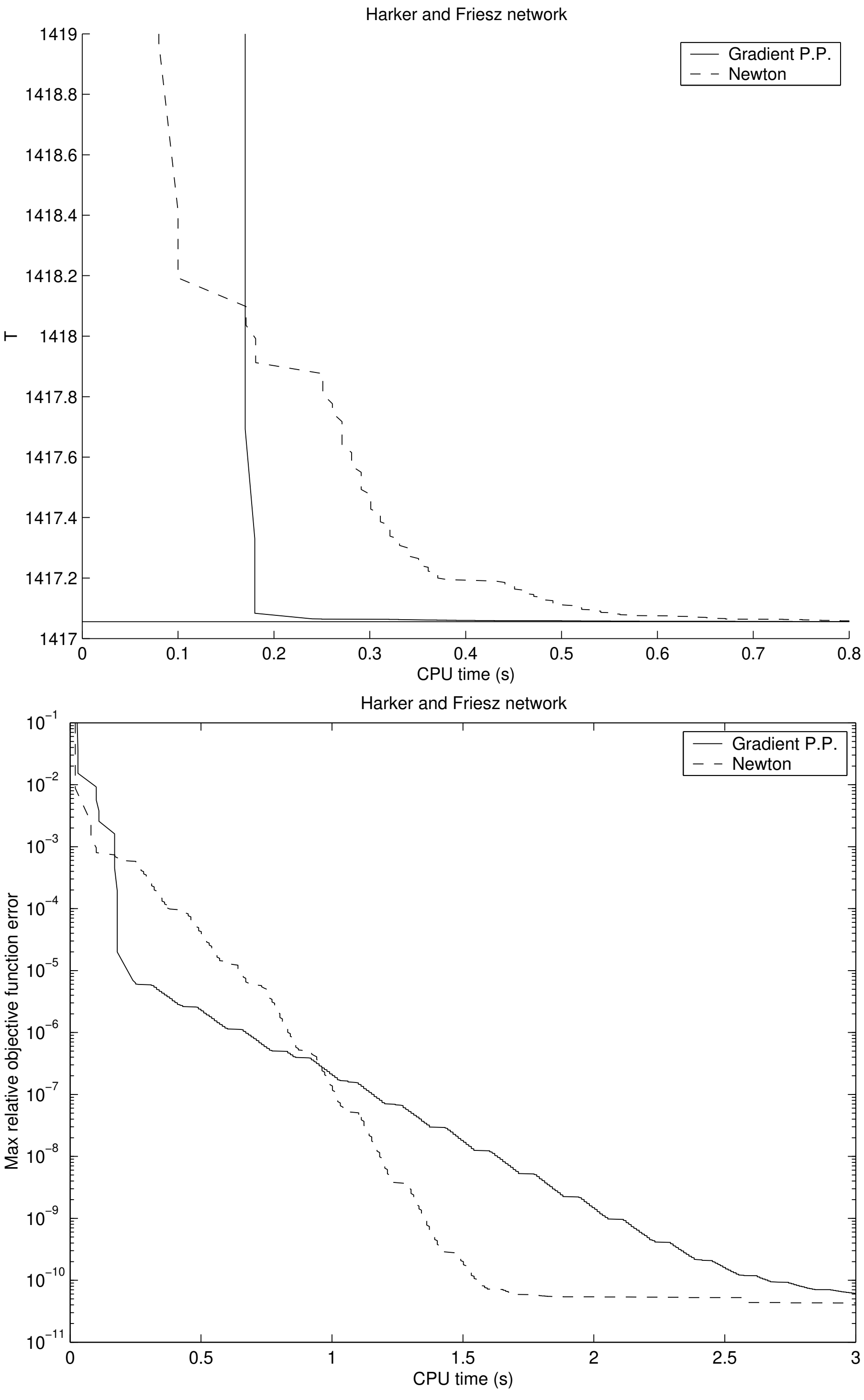


Figure 10: Algorithm performance on the Harker and Friesz network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors. A reminder: since the top diagram uses a linear scale, the long-run convergence trends is not clearly visible in it.

	Gradient P.P.	Newton
CPU-time (s)	4.066	6.73
Line searches	1118	1883
$S(v)$	8.20e-11	1.32e-11
$T(v)$	1417.05544164494	1417.05544164494
Lower bound on $T(v)$	1417.05544117299	1417.05544156873
Max. rel. error (using best lbd)	5.38e-11	5.38e-11
Avg. rel. route cost diff.	1.16e-9	1.39e-11
Routes generated	6	6
Routes used by solution	6	6
Avg cost at solution	1.918864	1.918864
Rel. feas. error of solution	0	1.18e-16
Rel. feas. error before corr.	1.35e-14	5.00e-12
$T(v)$before corr.	1417.05544164483	1417.05544160801

Table 7: Results for the Harker and Friesz network.

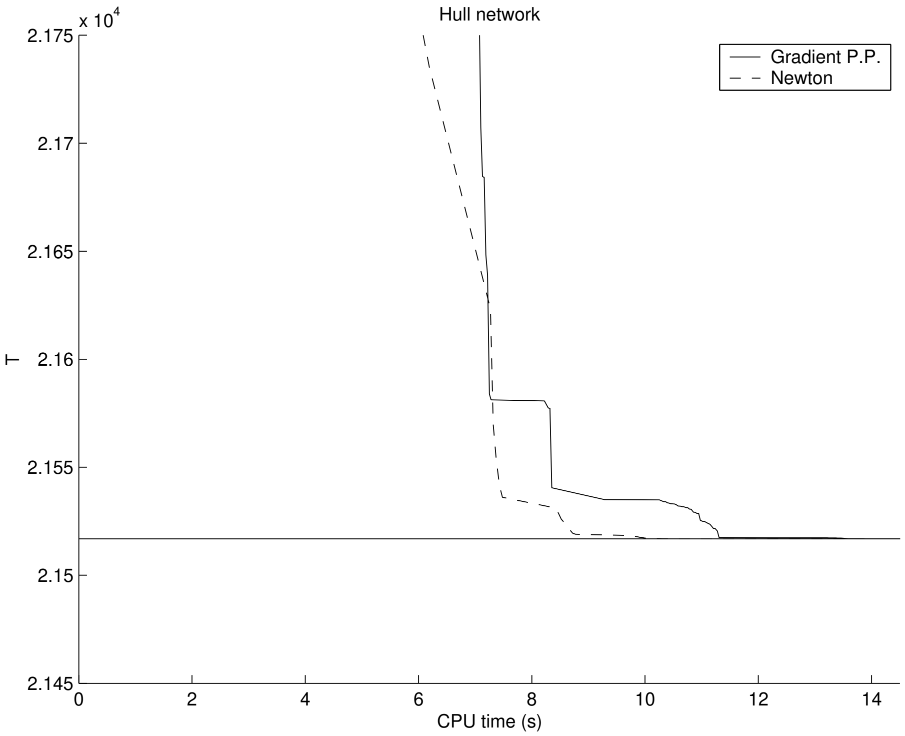
perform best, the difference was far less than what is apparent from the table. It seems that the gradient-based algorithm reached a point very near the final required tolerance after about 25s, but then had difficulties improving it further. The reason for this may have been implementation-related rather than related to the algorithm. It should be noted that tolerances of 10^{-10} are so low that there are likely issues with the numerical precision of variables in the implementation. It was observed that tolerances significantly lower than this could not be obtained regardless of the algorithm and network used.

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50
Newton / 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50

Table 8: Settings for the Hull network.

	Gradient P.P.	Newton	Larsson/Patriksson
CPU-time (s)	294.55	15.30	N/A
Line searches	5440	79	56
$S(v)$	9.12e-10	8.48e-11	?
$T(v)$	21516.84578406213	21516.84578406212	21516.85
Lower bound on $T(v)$	21516.8457548	21516.8457813459	21516.64
Max. rel. error (using best lbd)	1.26e-10	1.26e-10	1.96e-7
Avg. rel. route cost diff.	8.27e-10	1.25e-10	?
Routes generated	329	316	?
Routes used by solution	232	229	?
Avg cost at solution	2.71667	2.71667	?
Rel. feas. error of solution	3.77e-19	4.82e-18	?
Rel. feas. error before corr.	5.34e-12	1.11e-14	N/A
$T(v)$before corr.	21516.8457840	21516.8457840619	N/A

Table 9: Results for the Hull network.



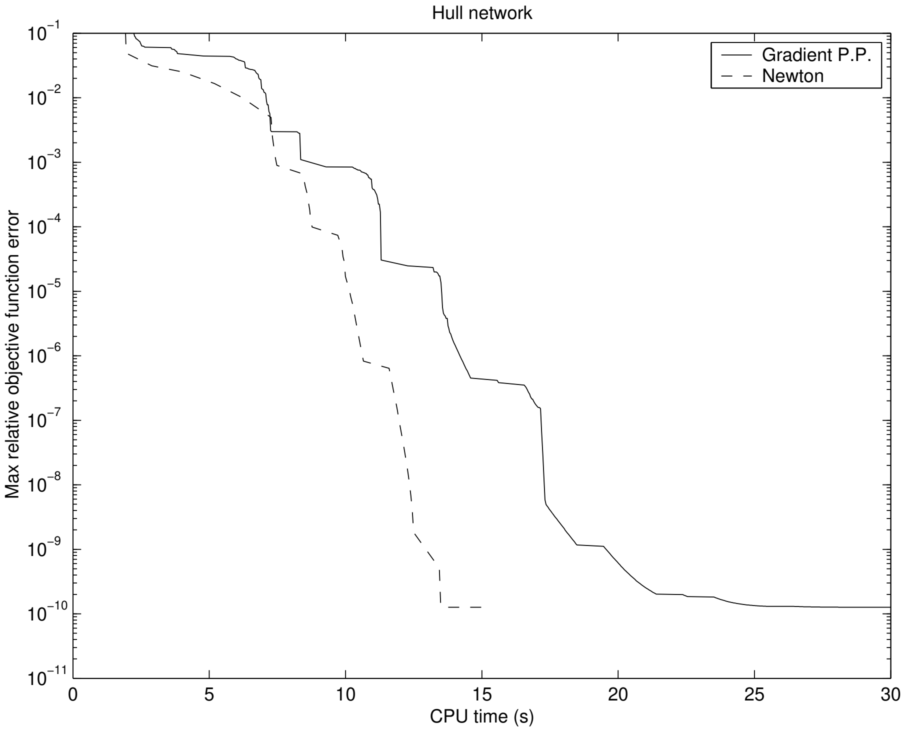


Figure 11: Algorithm performance on the Hull network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors.

7.1.5 Sioux Falls network

This is a relatively small network (76 arcs, 24 nodes, 528 OD-pairs) modeling the city of Sioux Falls, South Dakota, USA. The network is illustrated in the diagram in Figure 7.1.5 below. Apart from the links and nodes, the diagram also shows equilibrium link flows. Flows are represented by bars, so that the thickness of each bar is proportional to the link flow on the link next to it. Seen in the direction of travel on each link, the bar is always to the right of the link.

Table 10 shows the solver settings used for the executions on this network, and the results are summarized in Table 8. The time-convergence characteristic of the algorithm executions is illustrated in Figure 13.

We see from the graphs that, although the Newton-based algorithm ultimately reached the final required tolerance slightly quicker than the gradient-based method, it did in fact perform significantly worse throughout most of the executions (before catching up). The likely reason that the Newton algorithm was slower initially is simple: the more advanced search direction generation means that each iteration takes a bit a longer. The fact that the Sioux Falls network has unproportionally many OD-pairs means that this effect is especially significant for this network. In fact, search direction generation accounted for a full 46 percent of the total CPU-time consumption in the 'Newton/1' execution.

The fact that the Newton method eventually catches up and terminates first should not be seen as significant – this only happens at tolerances so strict (about 10^{-9}) that the numerical imprecison of the system is likely to affect the convergence.

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	1e-1; 1e-2; 1e-3; 1e-5	100	25
Gradient P.P./ 2	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50
Newton / 1	1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	100	50

Table 10: Settings for the Sioux Falls network.

	Gradient P.P./1	Gradient P.P./2	Newton	Frank-Wolfe	Lar./Pat.
CPU-time (s)	12.5	122.4	108.0	168.6	N/A
Line searches	103	1097	778	878	99
$S(v)$	7.82e-6	5.72e-11	9.91e-11	?	?
$T(v)$	42.313355	42.3133528717117	42.3133528717117	42.318164	42.31356
Lower bound on $T(v)$	41.312769	42.3133528674	42.313352864	42.3104	42.311
Max. rel. error (best lbd)	3.95e-8	1.02e-10	1.02e-10	1.14e-4	4.90e-6
Avg. rel. route cost diff.	4.86e-6	5.23e-11	5.83e-11	?	?
Routes generated	1099	1103	1132	N/A	?
Routes used by solution	717	740	711	N/A	?
Avg cost at solution	0.2074292	0.2074383	0.2074383	50.2803	?
Rel. feas. error of solution	1.92e-18	2.85e-18	1.92e-18	?	?
Rel. feas. error before corr.	3.14e-15	3.57e-14	4.50e-13	N/A	N/A
$T(v)$before corr.	42.313355	42.3133528717106	42.3133528716981	N/A	N/A

Table 11: Results for the Sioux Falls network.

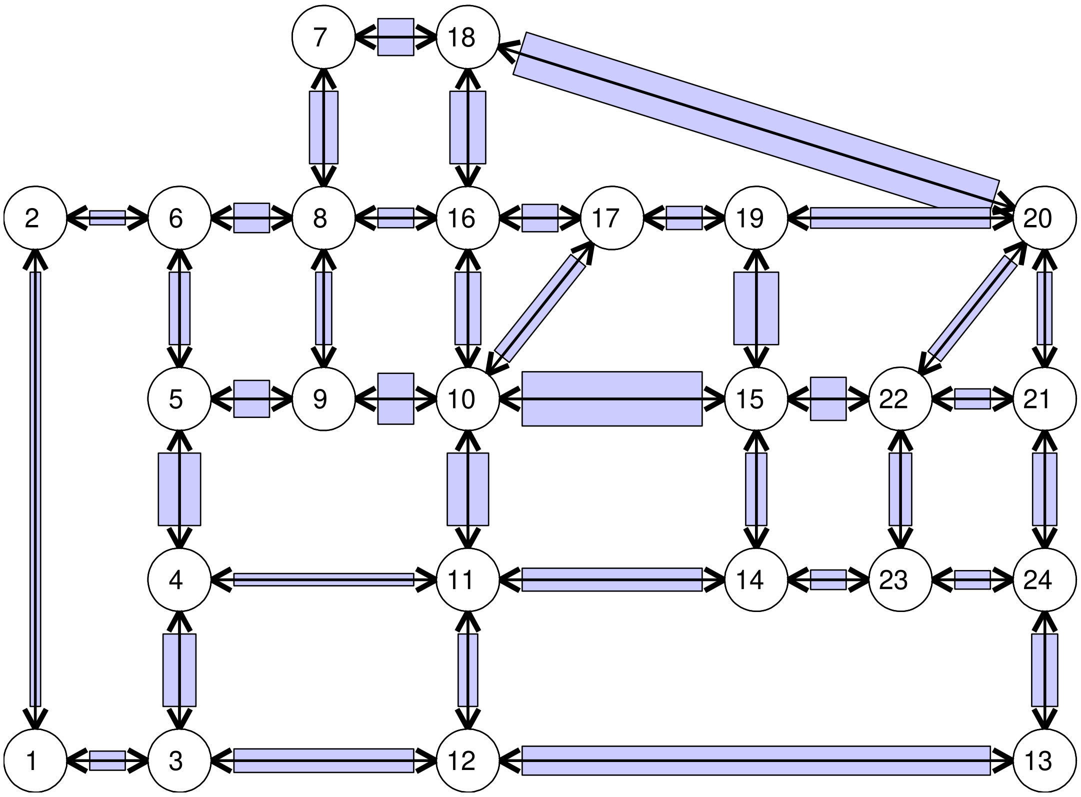


Figure 12: Sioux Falls network, with equilibrium flows.

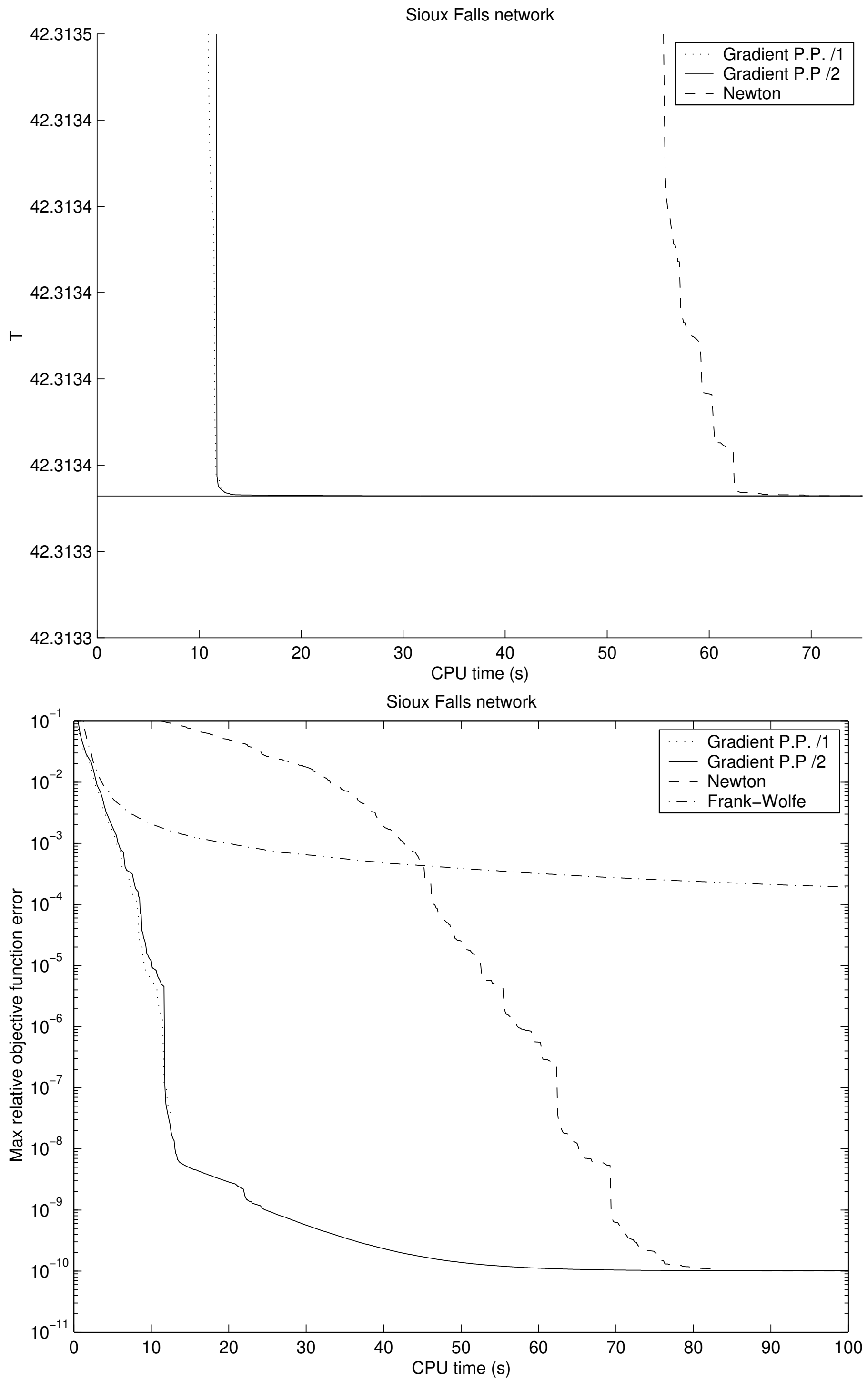


Figure 13: Algorithm performance on the Sioux Falls network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors. A reminder: since the top diagram uses a linear scale, the long-run convergence trends is not clearly visible in it.

7.1.6 Barcelona network

This is a 2522-link, 1020-node, 7922-OD-pair network modeling the city of Barcelona. It should be noted that this model includes links with constant travel cost functions and thus does not fully conform to our network model standards. In theory, some problems could therefore arise with non-unique solutions. Also note that this model uses travel cost functions with non-integer powers of the link flows. The BPR-type version of the solver was therefore necessarily used.

Table 12 shows the solver settings used for the executions on this network, and the results are summarized in Table 13. The time-convergence characteristic of the algorithm executions is illustrated in Figure 14.

We see here that the Newton algorithm performed significantly better than Gradient-based method. One noteworthy phenomenon is that for ”Newton/1”, all of the routes generated were still used (i.e. had flow greater than zero) in the final solution. This indicates that the solution may not be accurate enough to completely discriminate between those routes that would be used in a perfect equilibrium solution and those that would not, which could be a problem when wishing to apply sensitivity analysis to the solution. See Section 7.2.5 for a discussion of this issue.

As can be seen, the second execution of the Newton algorithm (”Newton/2”) used a much stricter final tolerance value than the others in order to reach a better solution. This execution also seems to show faster convergence – the reason for this is that the ’iterations before failure’ setting was set higher, so the time-consuming checks for new routes were performed more rarely.

Table 13 illustrates one other point: the relative weakness of the ’average relative route cost difference’ tolerance coefficient. Note that the final value of this coefficient was actually lower for the solution from the gradient-based algorithm than from the (much better) solution from the Newton execution. The reason why this happens is that this coefficient does not take into account the size of the flows on different routes – just which routes are used at all (and the route costs). Hence, a situation where most of the flow for an OD-pair is on a slightly too expensive route is valued the same as a situation where a tiny flow is on the slightly too expensive route.

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	0.1; 0.02; 0.005; 0.001	2	10
Newton / 1	0.1; 0.02; 0.005; 0.001	2	10
Newton / 2	0.001; 1e-4; 1e-5	10	50

Table 12: Settings for the Barcelona network.

7.1.7 Winnipeg network

This is a 2836-link, 1052-node, 4344-OD-pair network modeling the city of Winnipeg, Canada. As the Barcelona model, this network model includes links with constant travel cost functions, so the same reservations apply here. Furthermore, as with the Barcelona network, this model uses travel cost functions with non-integer powers of the link flows. The BPR-type version of the solver was therefore necessarily used.

Table 14 shows the solver settings used for the executions on this network, and the results are summarized in Table 15. The time-convergence characteristic of the algorithm executions is

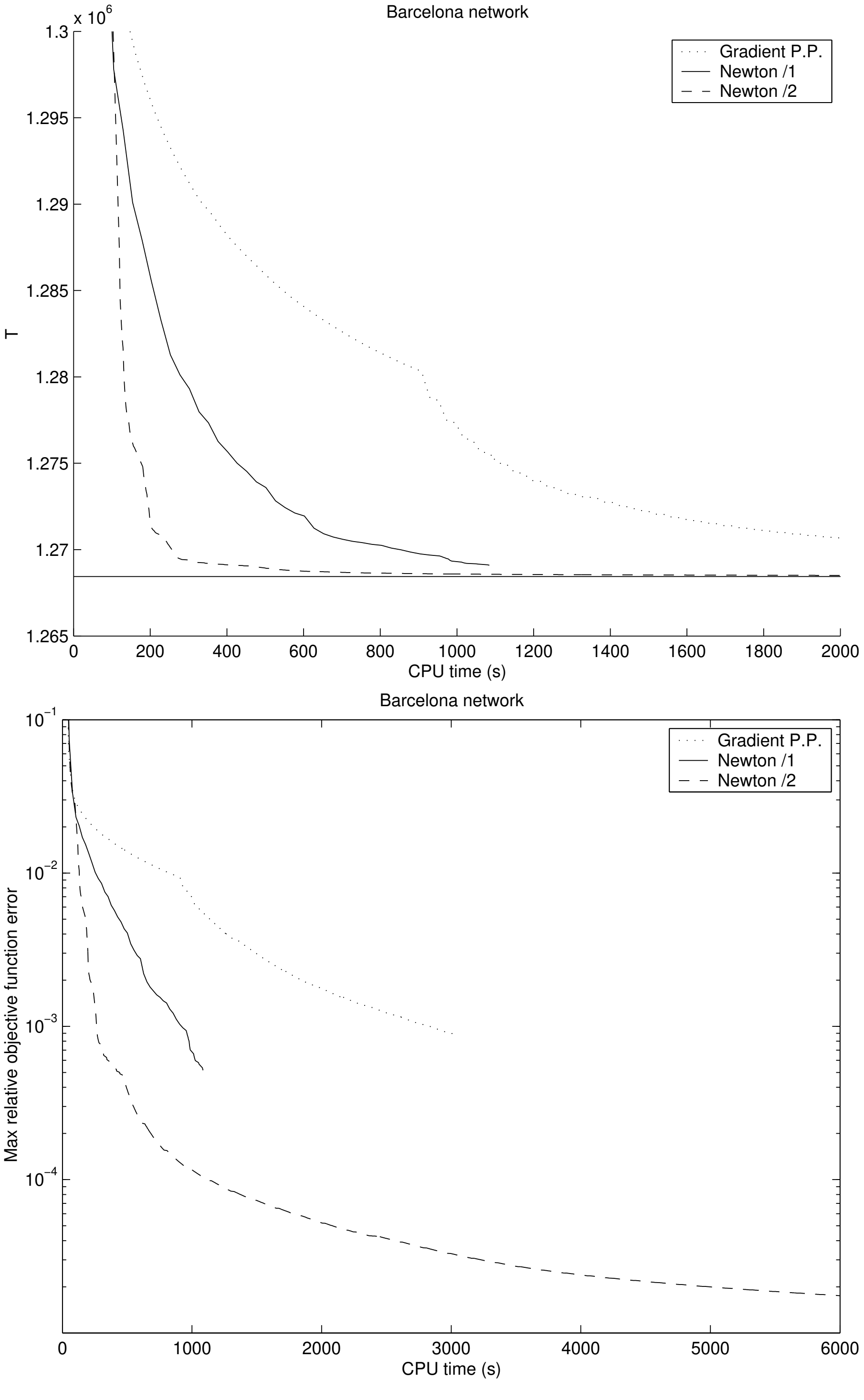


Figure 14: Algorithm performance on the Barcelona network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors. Note that the reason some graphs seem cut off is that the executions of the algorithms reached the requested error tolerances at those points.

	Gradient P.P.	Newton/1	Newton/2	Larsson/Patriksson
CPU-time (s)	3039	1085	11444	N/A
Line searches	516	54	3040	89
$S(v)$	1.23e-3	6.8e-4	1.2e-5	?
$T(v)$	1269553.75	1269097.76	1268460.50	1272128.59
Lower bound on $T(v)$	1267784.36	1268117.19	1268443.09	1268414.61
Max. rel. error (using best lbd)	8.756e-4	5.161e-4	1.373e-5	2.906e-3
Avg. rel. route cost diff.	0.000378	0.017914	0.006326	?
Routes generated	29581	32279	25015	20528
Routes used by solution	13013	32279	21463	18376
Avg cost at solution	7.7697	7.7550	7.7499	?
Rel. feas. error of solution	6.08e-18	9.02e-18	4.18e-18	?
Rel. feas. error before corr.	1.06e-15	5.01e-13	1.30e-12	N/A
$T(v)$before corr.	1269553.75	1269097.76	1268460.50	N/A

Table 13: Results for the Barcelona network.

illustrated in Figure 15.

As with the Barcelona network, the Newton-based algorithm performed significantly better than the gradient-based method, especially in the second execution, where fewer shortest-route checks were made.

Algorithm/execution	Sequence of tolerances	Max failures	Iterations before failure
Gradient P.P./ 1	0.1; 0.02; 0.005; 0.001	2	10
Newton / 1	0.1; 0.02; 0.005; 0.001	2	10
Newton / 2	0.001; 1e-4; 1e-5	10	50

Table 14: Settings for the Winnipeg network.

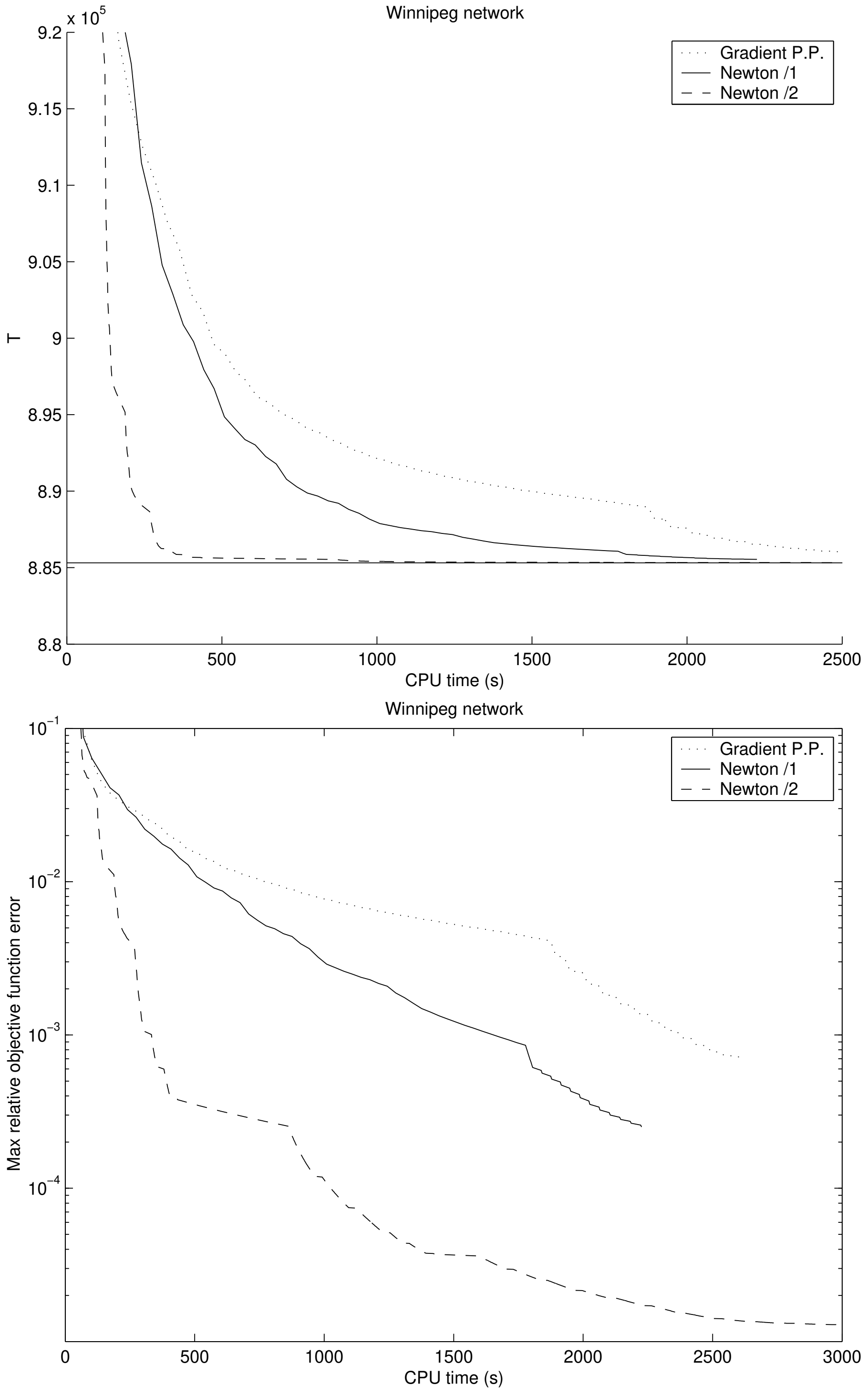


Figure 15: Algorithm performance on the Winnipeg network. In the top diagram, the horizontal line shows the best lower bound. This bound was used to calculate the maximum errors. Note that the reason some graphs seem cut off is that the executions of the algorithms reached the requested error tolerances at those points.

	Gradient P.P.	Newton/1	Newton/2	Patriksson 92
CPU-time (s)	2624	2225	3056	N/A
Line searches	169	85	878	177
$S(v)$	9.56e-4	4.40e-4	9.95e-6	?
$T(v)$	885944.76	885539.24	885328.13	885656.67
Lower bound on $T(v)$	884855.02	885037.91	885316.79	884134.68
Max. rel. error (using best lbd)	7.09e-4	2.51e-4	1.28e-5	3.84e-4
Avg. rel. route cost diff.	0.000586	0.03711	0.03887	?
Routes generated	22524	24694	18073	16744
Routes used by solution	9340	24694	18073	9583
Avg cost at solution	17.227	17.206	17.582	?
Rel. feas. error of solution	2.43e-18	7.53e-18	2.17e-18	?
Rel. feas. error before corr.	1.83e-15	6.38e-14	2.00e-13	N/A
$T(v)$before corr.	885944.76	885539.24	885328.13	N/A

Table 15: Results for the Winnipeg network.

7.2 Algorithm performance and convergence issues

7.2.1 Feasibility Errors

In theory, any correct algorithm should not introduce any feasibility errors in the solution. However, because of the limited numerical precision used in computer implementations, feasibility errors will necessarily occur. It is therefore good to be aware of these errors and to what degree they influence the convergence of the algorithm implementations. Typically, these errors will remain very small and hence not pose a significant problem. However, the feasibility errors do tend to build up and increase as more and more iterations are performed. This along with the fact that small networks can be solved very quickly and precisely means that eventually the errors will be large enough (relative to the precision of the solution) that further progress will be slowed down or stopped altogether. In practice, it was found in our implementation that it was rarely possible to achieve a solution more accurate than having a termination coefficient of 10^{-10}, regardless of the network. Of course, for the very large networks, such an accuracy could not be achieved within a reasonable computation time anyway, so there feasibility and numerical errors did not pose a problem. On the other hand, for even more accurate solutions to the smaller networks, an implementation using variables of greater numerical precision (or some other improvement) would be needed.

One piece of evidence for the influence of numerical rounding effects that was observed was that when the code was executed on a Unix system, a different number of iterations was required than when the same code was executed on the PC system normally used. Presumably, this means that MATLAB behaves slightly different numerically on the two different platforms.

The problem of feasibility errors would be even more severe in an implementation where the objective function value is used in the termination criterion. This is because the objective function value is extremely sensitive to infeasibilities. As seen from several of the tables in the previous section, a slightly infeasible solution may even have an apparent objective function value less than the actual lower bound. Hence, using a termination criterion based on the objective function value could in such a case cause a premature termination of the algorithm. Luckily, route-cost based termination criteria suffer much less from this problem.

In order to analyze and present true results, a correction process was used to eliminate (or at least reduce) the feasibility errors of the calculated solution after each termination of an execution. The basic method used was to either add the 'missing' OD-pair flow to the cheapest route or remove the 'surplus' OD-pair flow from the most expensive route for each OD-pair.

An experiment was performed to see if enforcing feasibility throughout each execution would let us reach greater accuracies. However, no significant improvement in convergence was noted. On the other hand, this mechanism could still be useful if there for some reason was a need to generate accurate objective function values throughout the execution.

7.2.2 Settings and constants

As explained in the implementation Section (see 6.1.3), a sequence of progressively stricter tolerance settings was used for each execution of the algorithms. It was observed that the entire sequence used (as opposed to just the final tolerance) had a significant effect on the CPU time necessary to reach the final tolerance. While this partly seemed to be due to random numerical effects, some general guidelines in selecting a tolerance sequence can be observed. One important point is to use fewer and/or stricter tolerance steps whenever a large network (and thus computationally expensive shortest route calculations) is used. Otherwise, there is a risk that a lot of computation time is initially spent generating new routes over and over again while remaining at a very lenient tolerance level. In a more general setting, it was observed that using fewer tolerance steps typically could be somewhat faster, whereas more tolerance steps was possibly slightly more robust. Table 16 illustrates some examples using the Newton-based algorithm on the Hull network.

Another important setting is the value of the 'iterations before failure'. Setting this constant to a high value makes sure that the algorithm rarely checks for new routes except whenever the tolerance has been reached. This is useful when solving very large problems, particularily if the number of links is also large compared to the number of OD-pairs. In such a case, cheapest route algorithms are likely to be expensive relative to the search direction generation.

Sequence of tolerances	CPU-time used (s)
1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10	14.79
1e-1; 1e-3; 1e-6; 1e-10	13.44
1e-2; 1e-6; 1e-10	12.47
1e-2; 1e-10	12.26
1e-4; 1e-10	15.87
1e-4; 1e-6; 1e-8; 1e-10	18.59
1e-10	13.07

Table 16: Execution times when using different tolerance sequences (Hull network).

7.2.3 CPU-time Consumption

The CPU-time consumption was analyzed using MATLAB 6's 'profiler' tool in order to optimize the code. It was found that, for small and medium-sized networks, solving the master problems (and specifically, generating the search directions) was the most computationally intensive component of the implementation. This was particularly true for the Sioux Falls network, which is not surprising, considering the relatively large number of OD-pairs in this model. However, for the larger networks (Barcelona, Winnipeg and Hull), the route generation algorithm was by far the most CPU-intensive component. Of course, the actual numbers depend on the settings used for each execution. By setting the 'iterations before failure' constant to a higher value, it is possible to reduce the number of shortest route calculations and instead perform more iterations (i.e., line searches) during each main iteration. Similarly, by setting the same constant to a lower value, it is possible to force more frequent route generation processes to possibly reduce the necessary number of iterations in restricted master problem phases. It should be noted, though, that this issue is quite implementation-specific – implemented in a different programming language, the bottlenecks of the algorithms may be different.

7.2.4 Line Search Tolerance

It was found to be advantageous to perform the line searches in the restricted master problem algorithms very accurately. The reason for this is that, since the search is done in the link flow space, this operation is usually much faster than the operations performed in the often much larger route flow space, such as the search direction generation. The maximum error allowed in any component of the calculated link flow step vector (i.e. link flow search direction vector times step length) was therefore set to 10^{-12} times the mean demand for the OD-pairs in the network.

7.2.5 Considerations for solving TAP for sensitivity analysis

Our sensitivity analysis method makes use of the knowledge of which routes in a network have positive equilibrium flows. Hence, solving the original TAP well enough that the set of equilibrium-used routes can be determined correctly is important. Of course, the equilibrium route-flow solution is generally non-unique, so the set of equilibrium-used routes will also be non-unique. Nevertheless, if a route that will have zero flow at the true equilibrium a route flow solutions is near to has even a tiny positive flow in the calculated solution, the result of a sensitivity analysis performed on the solution may contain significant errors. Hence, it is useful to have at least an idea of how well a solution fills the criterion of having the set of routes with positive flow equal to such a set for a corresponding exact equilibrium. Table 17 shows the routes generated and used in the Hull network solution as a function of the final termination coefficient used. In all cases, the tolerances sequence used (up to the final value in each case) was [1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10]. The number of iterations before failure was 50.

	Newton	Newton	Gradient P.P.	Gradient P.P.
S_ε	Routes generated	Routes used	Routes generated	Routes used
1e-10	316	229	329	232
1e-8	316	229	329	232
1e-7	316	229	329	232
1e-6	316	231	329	232
1e-5	316	231	329	230
1e-4	316	231	329	218
1e-3	316	242	328	204
1e-2	316	316	322	235
1e-1	316	316	296	234

Table 17: Number of routes generated and used when using the two different algorithms with various error tolerances.

Both algorithms seem to acquire the final set of used routes at about the same tolerance level. Note that the fact that the two algorithms converge to solutions using different number of routes is not surprising – this is just a result of the fact that the equilibrium solution is non-unique. However, some other interesting observations can be made. Note that, for the Newton-based method, all generated routes are used up until and including the tolerance coefficient 1e-2. This phenomenon was previously observed for the Barcelona and Winnipeg networks at the level 1e-3. By contrast, for the gradient-based method, the set of routes used seems to change much more throughout the execution, even at the less strict tolerance levels. Also, route usage seems more conservative (i.e. fewer routes are used) at the less strict tolerance levels. This could indicate that this algorithm may be a better choice when solving a problem where acquiring sufficient accuracy is not feasible, but where one still requires a good approximation of the set of equilibrium-used routes. However, more testing would be needed to verify this theory.

Note that, in the practical use of the sensitivity algorithm, we chose to safeguard against the problem discussed in this section. Refer to Section 6.2.3 for a reminder.

8 Results - Sensitivity analysis

There are few previously published implementations of sensitivity analysis for equilibrium network problems, so there does not exist much previous data to compare with. Additionally, there is no natural measure of the accuracy of a sensitivity solution. Although objective function values can be calculated for our solutions, these are specific to our approach and other sources are unlikely to give them. Nevertheless, these objective function values *can* be calculated for any solution regardless of how it was obtained. Of course, for a comparison to be meaningful, both sensitivity solutions must be based on the same equilibrium solution and consider the same direction of change.

We will consider a single numerical example in detail, using the Sioux Falls network. This network was chosen because it is computationally non-trivial while still being small enough that complete data can be presented reasonably concisely.

8.1 Numerical example

We will consider the impact of increasing the capacity of the link from node 13 to node 24 for the Sioux Falls network (refer back to Section 7.1.5, or see Figure 16 below for a diagram of this network). In other words, this link's capacity shall have a directional derivative of 1 with respect to the single parameter used, all other link capacities having derivatives zero. For the basis of the sensitivity analysis, the solution labeled "Newton/1" (refer back to Section 7.1.5) will be used. This equilibrium solution is presented in detail along with the sensitivity analysis solution in Table 18.

The sensitivity solution is illustrated in Figure 16 below. The dark bars represent negative link flow derivatives and the light bars positive link flow derivatives.

This solution was obtained using a tolerance level of 1e-7 for the sensitivity problem (recall that the tolerance coefficient is different in the sensitivity case). Some more data about this solution and other solutions to the same problem obtained by using different tolerance levels is given in the next section.

8.2 Impact of termination coefficient

In Table 19 below, we present some key properties of sensitivity solutions obtained for the same problem using different tolerance levels. Some further characterizations of the different solutions are given in Table 20. We use here $\mathcal{S}_\varepsilon$ to denote the tolerance level of the sensitivity algorithm (recall that we use S_ε for the regular equilibrium tolerance level).

We see that the solver seems to behave much like in the regular equilibrium case. It appears that our termination coefficent for sensitivity analysis (see (119)) is a reliable measure of how accurate a solution is.

Note that the same feasibility correction procedure used in the equilibrium case has not been applied here. One reason for this is that, in the sensitivity case, a relative error cannot be defined in the same way since each total OD-pair demand is zero. However, for each OD-pair, the

Node i	Node j	Link flow	Link flow derivative	Node i	Node j	Link flow	Link flow derivative
1	2	4.494657641875	-0.078743781806	13	24	11.121357959899	0.377370466378
1	3	8.119079942939	-0.012225446609	14	11	9.814069071086	0.015909426855
2	1	4.519079942939	-0.012225446609	14	15	9.036334131500	0.042453293356
2	6	5.967336397248	-0.049759086978	14	23	8.400436815445	-0.021059598765
3	1	8.094657641875	-0.078743781806	15	10	23.192283386014	-0.010431230214
3	4	14.006371021194	-0.223857606225	15	14	9.079820317238	-0.002790964990
3	12	10.022319618587	0.005352353246	15	19	19.083289739208	0.028275429712
4	3	14.030560920276	-0.011406894973	15	22	18.409934986589	0.027276918959
4	5	18.006371021194	-0.223857606225	16	8	8.406714396147	-0.001801285612
4	11	5.200000000000	0.000000000000	16	10	11.073009306330	-0.004447000631
5	4	18.030560920276	-0.011406894973	16	17	11.695002844614	-0.001461429387
5	6	8.798267710795	-0.007323639648	16	18	15.278325422001	-0.016162721806
5	9	15.780782065771	-0.187549271750	17	10	8.100000000000	0.000000000000
6	2	5.991758698311	0.016759248219	17	16	11.683838208802	0.005653658616
6	5	8.806498665890	0.007137584184	17	19	9.953021440056	-0.001812148677
6	8	12.492925352671	-0.086067421453	18	7	15.854621672193	0.147048137253
7	8	12.101529114114	-0.007830020817	18	16	15.333406842197	0.030263458950
7	18	15.794010829850	0.049644705336	18	20	18.976796307887	-0.008332700989
8	6	12.525578608828	-0.005087862425	19	15	19.116724259134	0.011695516834
8	7	12.040918271771	-0.105233452734	19	17	9.941856804244	0.005302939326
8	9	6.882664901433	-0.004543444004	19	20	8.688366967529	0.006873981275
8	16	8.388713049658	-0.031242532104	20	18	18.992488570426	0.135496911684
9	5	15.796741009759	0.010440215670	20	19	8.710636851645	-0.002590843599
9	8	6.836705968759	-0.050408563384	20	21	6.302022924275	0.002382790225
9	10	21.744076097013	-0.137140708365	20	22	7.000000000000	0.000000000000
10	9	21.814076108326	0.014983659674	21	20	6.239985070928	0.136747578024
10	11	17.726625035204	0.041176410101	21	22	8.619539651704	-0.006159832849
10	15	23.125797308624	-0.144150494022	21	24	10.309410805328	-0.015995503973
10	16	11.047093868435	-0.028547022898	22	15	18.386472729790	0.132331837298
10	17	8.100000000000	-0.000000000000	22	20	7.000000000000	0.000000000000
11	4	5.300000000000	0.000000000000	22	21	8.607387881054	-0.018378294199
11	10	17.604223531232	0.035481492065	22	23	9.661824223643	-0.027742294383
11	12	8.365285655104	-0.013952463664	23	14	8.394900159180	0.108291051056
11	14	9.776119541613	-0.068196964619	23	22	9.626210196193	0.065094162607
12	3	9.973707418442	-0.273616693203	23	24	7.902983920981	0.024595614392
12	11	8.404934621659	-0.103753773175	24	13	11.112394726309	0.008600110419
12	13	12.287605273691	-0.008600110419	24	21	10.259524722632	0.130587745175
13	12	12.378642040101	-0.377370466378	24	23	7.861833237266	0.246782721203

Table 18: Links, equilibrium link flow solution, and link flow sensitivity solution for the Sioux Falls network. The direction of each link is from node i to node j.

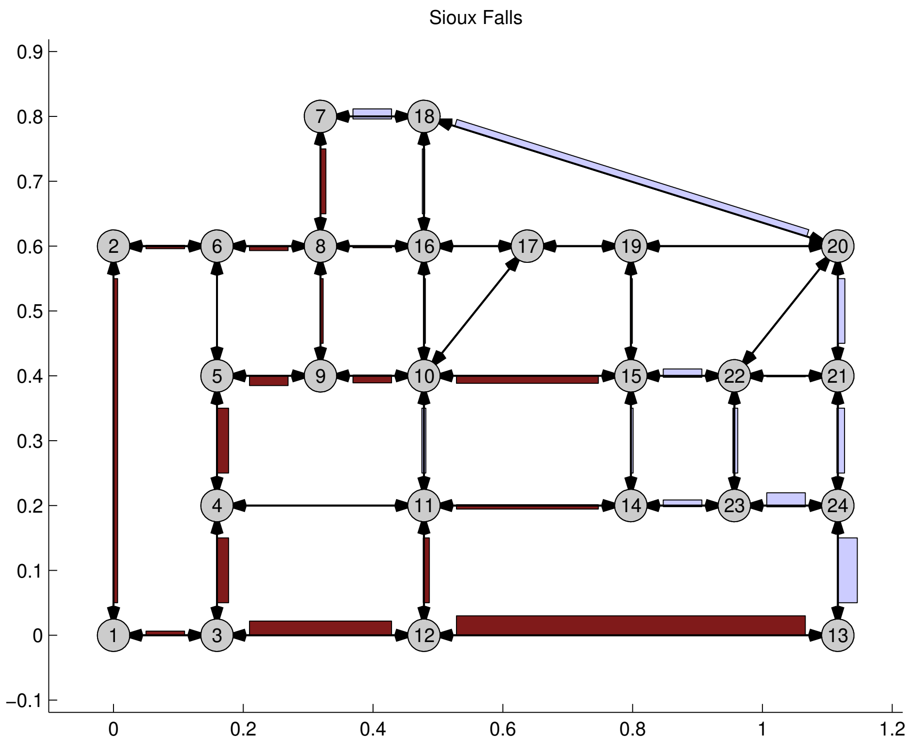


Figure 16: A sensitivity solution for the Sioux Falls network. Dark bars represent negative derivatives.

$\mathcal{S}_\varepsilon$	CPU time (s)	Iterations	Objective function value
1e-8	61.0	510	-5.63629225310869e-3
1e-7	54.9	455	-5.63629225310863e-3
1e-6	41.1	337	-5.63629225310234e-3
1e-5	29.5	246	-5.63629225283967e-3
1e-4	20.6	169	-5.63629218692616e-3
1e-3	7.7	61	-5.63628426721465e-3
1e-2	3.6	27	-5.63620819810396e-3
1e-1	2.3	15	-5.62531661762288e-3

Table 19: Characterization of the different solutions obtained when using different tolerance levels (Hull network).

$\mathcal{S}_\varepsilon$	Avg cost deriv.	Link (13,24) flow deriv.	Link (20,18) flow deriv.	Link (4,5) flow deriv.
1e-8	-5.54165148439e-4	0.377370464954	0.135496838929	-0.223857605766
1e-7	-5.54165251885e-4	0.377370466377	0.135496911683	-0.223857606224
1e-6	-5.54166205016e-4	0.377370479496	0.135497579675	-0.223857608806
1e-5	-5.54171990876e-4	0.377370549442	0.135501683763	-0.223857694719
1e-4	-5.54273507735e-4	0.377371955052	0.135572830380	-0.223857886630
1e-3	-5.55351321535e-4	0.377378205442	0.136310181789	-0.223840118125
1e-2	-5.56921591146e-4	0.377343468943	0.137132917608	-0.223652442355
1e-1	-5.66624415113e-4	0.370202329037	0.133956908342	-0.214559491134

Table 20: Further characterization of the different solutions obtained when using different tolerance levels (Hull network).

feasibility error relative to the total shift in flow between routes was very tiny (smaller than 1e-16).

A tolerance level of 1e-9 or better for this example could not be achieved within a reasonable number of iterations (2500). This was likely due to the limited numerical precision of the implementation.

8.3 Impact of accuracy of original equilibrium solution

In Tables 21 and 22 below, we present data similar to in the previous section but for sensitivity solutions based on equilibrium solutions of different accuracies. In all cases, the equilibrium solver tolerance sequence used (up to the final value in each case) was [1e-1; 1e-2; 1e-3; 1e-4; 1e-6; 1e-8; 1e-10]. The number of iterations before failure was 50. For the sensitivity solver, the tolerance level in each case was 1e-7.

S_ε	CPU time (s)	Iterations	Objective function value
1e-10	54.9	455	-5.63629225310863e-3
1e-8	45.1	377	-5.63628930882169e-3
1e-7	77.1	561	-5.63628461434828e-3
1e-6	88.3	1113	-5.63624391920170e-3
1e-5	85.5	730	-5.63631058061617e-3
1e-4	101.1	915	-5.62531180691629e-3
1e-3*	167.1	1500	-6.17930871610098e-3

Table 21: Characterization of the different sensitivity solutions obtained when using different error tolerances for the equilibrium solver (Hull network).

S_ε	Avg cost deriv.	Link (13,24) flow deriv.	Link (20,18) flow deriv.	Link (4,5) flow deriv.
1e-10	-5.54165251885e-4	0.377370466377	0.135496911683	-0.223857606224
1e-8	-5.54165036482e-4	0.377370390234	0.135496629455	-0.223857521373
1e-7	-5.54164059310e-4	0.377370218372	0.135495302040	-0.223857327529
1e-6	-5.54161744496e-4	0.377369222117	0.135488374335	-0.223856582141
1e-5	-5.54124432862e-4	0.377367940053	0.135467851962	-0.223871234892
1e-4	-5.53825221495e-4	0.377103688548	0.134253897090	-0.222720033934
1e-3*	-4.76288511464e-4	0.411988996797	0.170905471938	-0.137664076855

Table 22: Further characterization of the different sensitivity solutions obtained when using different error tolerances for the equilibrium solver (Hull network).

Note that, in the S_ε =1e-3 case, the sensitivity solver was terminated at the preset limit of 1500 iterations without reaching the tolerance level 1e-7.

From the tables we can spot some interesting trends. Apparently, sensitivity calculations based on equilibrium solutions of lower accuracy generally take much longer to reach a certain tolerance level.

Additionally, the sensitivity solutions then obtained are significantly worse than what you get using the same sensitivity tolerance level with a better equilibrium solution. It is clear from the data that the tolerance level of the original equilibrium solution should be at least as strict as the tolerance level for the sensitivity solution.

9 Results - Network Design

9.1 Comparisons of results for well known networks

9.1.1 Braess network

This is the same network as was discussed in Section 2.4. Refer back to Figure 2 in this section for a diagram. It has been used as a network design test network in several previous publications. These have used different variations of the network in terms of link cost functions, investment cost functions, and demands. The variation we shall look at is the one used in Bell and Iida (1997), since this problem demonstrates the problem of non-uniqueness of local optima very well, yet has a simple enough structure to be looked at in detail analytically. For reference, see the diagram of the network in Section 2.4, Figure 2. Since this network is very simple, it will not be a particularly important test of our algorithm in terms of performance. However, the symmetry and simplicity of the problem will allow us to look at some aspects of it analytically, which will be helpful in understanding the problems involved in performing network design on this and other more complicated networks.

Now, let the link cost functions for the network be

$$\begin{aligned} t_1(y_1,v_1) &= 1+2\left(\frac{v_1}{3.2+y_1}\right)^2, \quad & t_2(y_2,v_2) &= 10+1\left(\frac{v_2}{3.2+y_2}\right)^2, \\ t_3(y_3,v_3) &= 1+2\left(\frac{v_3}{3.2+y_3}\right)^2, \quad & t_4(y_4,v_4) &= 1+2\left(\frac{v_4}{3.2+y_4}\right)^2, \\ & & t_5(y_5,v_5) &= 1+2\left(\frac{v_5}{3.2+y_5}\right)^2, \end{aligned} \tag{124}$$

and the investment cost function and bounds be

$$\psi(y) = \sum_{l=1}^{5} 3y_l, \qquad y_l^l = 0,\ y_l^u = 25, \quad l = 1,2,3,4,5. \tag{125}$$

First note that there is a symmetry in the network (i.e. in network topology, travel cost functions, *and* investment cost functions) between links 2 and 4, and similarly, between links 1 and 5. This means that any local optimum in the subset defined by $y_1 = y_5$ and $y_2 = y_4$ will also be a local optimum in the entire space. The symmetry itself does not tell us that the *global* optimum will necessarily be in this hyperplane, and we shall not prove this. However, this theory seems reasonable considering that both the travel cost functions and the investment cost functions are convex. Now, to simplify the problem further, we shall assume that y_2 (and hence also y_4) is zero. This may seem a very arbitrary assumption, but it is based on the fact that a significant part of the OD-pair flow will always be on the route through links 1,3, and 5, and hence, increasing y_2 (and y_4) from zero will have a positive cost derivative regardless of y_1 (=y_5). Now, let us graph the design objective function (Z) on the plane specified by ($y_1 = y_5$, $y_2 = y_4 = 0$). The result is shown in Figure 17.

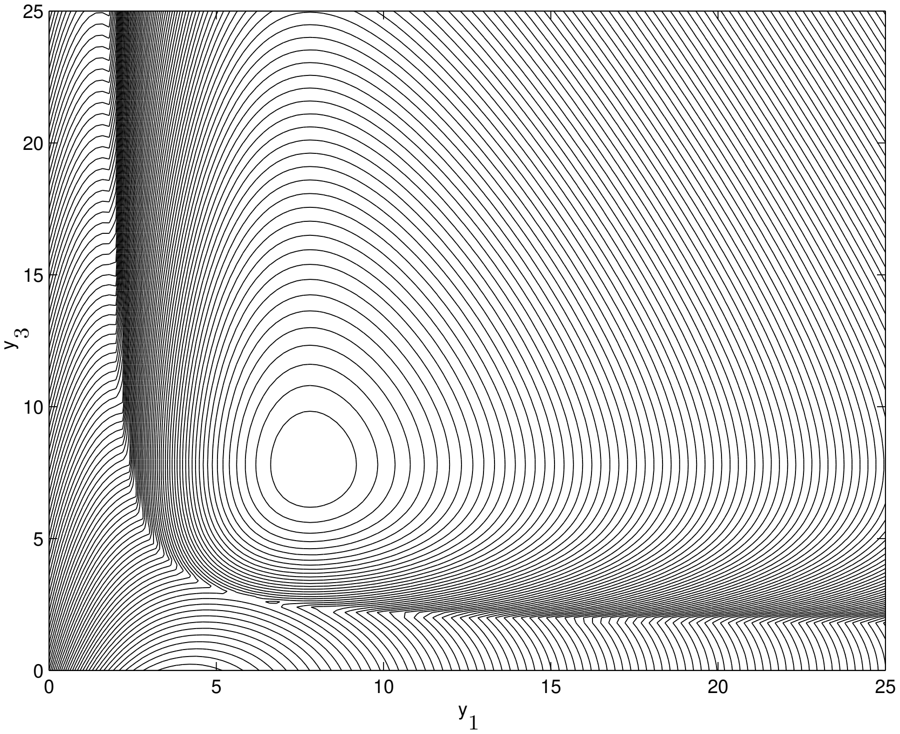


Figure 17: Graph of the design objective function on the plane defined by $y_1 = y_5$, $y_2 = y_4 = 0$.

We see in this figure that there appear to be two regions in the plane, separated by a curve of points where the problem is non-differentiable. Each region has a single optimum. In fact, these two regions represent the sets of capacity enhancements that lead to all routes being used (the region containing $(0, 0)$) and only the route through links 1,3, and 5 being used (the second region, containing e.g. $(10, 10)$). We can analytically calculate the equation of the curve of non-differentiable points separating the regions. Recall that non-differentiability arises from the equilibrium solution being non-complimentary. Thus, the curve between the two regions can be calculated by assuming that the route through links 1,3, and 5 is the only one used, but that the other routes (i.e. through the links 1 and 4, and 2 and 5, respectively) have the same cost. This relation can be written:

$$1 + 2\left(\frac{10}{3.2 + y_1}\right)^2 + 10 + 2\left(\frac{0}{3.2 + y_2}\right)^2 = 2 + 4\left(\frac{10}{3.2 + y_1}\right)^2 + 1 + 2\left(\frac{10}{3.2 + y_3}\right)^2 \quad (126)$$

$$\Longrightarrow 8 = 2\left(\frac{10}{3.2 + y_1}\right)^2 + 2\left(\frac{10}{3.2 + y_3}\right)^2 \quad (127)$$

$$\Longrightarrow 1 = 2\left(\frac{5}{3.2 + y_1}\right)^2 + 2\left(\frac{5}{3.2 + y_3}\right)^2. \quad (128)$$

Solving for one variable we can also obtain,

$$y_1 = \frac{5}{\sqrt{1-\left(\frac{5}{3.2+y_3}\right)^2}} - 3.2, \quad y_3 > 1.8. \tag{129}$$

As can be determined by counting the equidistance lines in diagram 17 or otherwise, the optimum in the second region is the better one. Let us now calculate the location of this local optimum analytically. Assuming that all flow is on the route through links 1,3, and 5, we can write down the design objective function as:

$$Z = 6y_1 + 3y_3 + C = 6y_1 + 3y_3 + 10\left[2\left(1+2\left(\frac{10}{3.2+y_1}\right)^2\right) + 1 + 2\left(\frac{10}{3.2+y_3}\right)^2\right] \tag{130}$$

$$= 6y_1 + 3y_3 + 3 + 40\left(\frac{10}{3.2+y_1}\right)^2 + 20\left(\frac{10}{3.2+y_3}\right)^2 \tag{131}$$

$$\Longrightarrow \frac{\partial Z}{\partial y_1} = 6 - 8000\left(\frac{1}{3.2+y_1}\right)^3, \frac{\partial Z}{\partial y_3} = 3 - 4000\left(\frac{1}{3.2+y_3}\right)^3 \tag{132}$$

$$\Longrightarrow y_1 = y_3 =^3\sqrt{\frac{4000}{3}} - 3.2 \approx 7.806424162982. \tag{133}$$

As a quick check, note that this value seems reasonable considering the apparent location of the optimum in the diagram. So how does this relate to the results obtained using our algorithm? The fact is that, whenever starting at $(0, 0)$, the algorithm finds the optimum in the first region (the worse one). The reason for this is that the descent direction for low values of y_1 always has a negative y_1-component (this can be seen in Figure 17). Hence, y_1 (and y_3) will never obtain a positive value. The only way to find the better optimum is to use a different starting point. Even so, it can be difficult to predict which local optimum the algorithm will find.

The better local optimum was found when a starting point of $y = (10, 0, 10, 0, 10)$ was used. The location of the optimum was calculated to $(7.80642420, 0, 7.80642420, 0, 7.80642420)$, which is approximately the same as the theoretical solution. The design objective function value was found to be 149.7867262002582 (exactly the same as the theoretical value to the precision given by Matlab). It should be noted that also Bell and Iida (1997) found both of these local optima. Bell and Iida used two different algorithms that he referred to as the *iterative assignment algorithm* and the *iterative sensitivity algorithm*. The former was based on assuming fixed link flows during the upper level optimization (so as to make it a single level optimization), then re-solving the equilibrium problem and repeating. Bell and Iida's iterative sensitivity algorithm was similar to their iterative assignment algorithm, except that also some simple sensitivity information was used at the upper level. Their value for the location of the optimum was $y = (7.8064, 0, 7.8064, 0, 7.8064)$.

9.1.2 Harker and Friesz network

This is the same network as was discussed in Section 7.1.3 and shown in Figure 9. It has been used as a network design test network in many previous publications. However, several slightly different network demand scenarios have been used by various researchers. We shall here consider two different scenarios in order to compare our results with as many sources as possible. Capacity augmentations are allowed for all 16 arcs in the network. Network details and the investment cost functions used can be found e.g. in Suwansirikul et al (1987) or Friesz et al (1989).

The settings used for all the executions of our sensitivity analysis-based descent algorithm (SBD) on the Harker and Friesz network are summarized in Table 23. The convergence characteristics of SBD on the network are illustrated in Tables 24 and 25. The initial solution used was $y^0 = 0$ in both cases. Z is the network design objective function value (refer to Section 4.2 for a reminder). More accurate Z-values that were obtained by applying the DSD algorithm with a very strict tolerance (1e-10) to the solution at each stage is shown as well. Cumulative counts of the number of equilibrium and sensitivity problems solved are also given.

We note here that the computational load of network design on the more congested network scenario is higher than for the first scenario. Note that both of these executions ran until the algorithm could not make further progress (because of the limits to numerical precision).

Number of main iterations	not limited
Armijo factor, τ	0.01
Armijo multiplier, κ	2
Turns before freezing	3
Turns frozen	5
DSD route generation algorithm	Newton-based
DSD error tolerance, S_ε	1e-6
Sensitivity analysis tolerance, $\mathcal{S}_\varepsilon$.	1e-4
Sensitivity analysis iteration limit	5000

Table 23: Algorithm parameters used for the Harker and Friesz network. Refer to section 6.3.3 for a reminder of the parameters κ and τ used in the Armijo search.

Suwansirikul et al (1987) tested both these scenarios using two algorithms known as Hooke-Jeeves (H-J) and Equilibrium Decomposed Optimization (EDO). H-J is a heuristic which alternates between choosing one parameter and modifying it (the upper level problem), and then calculating a new equilbrium solution to base the next choice on (the lower level problem). EDO is similar except that, in the upper level problem, all links can be modified (this is achieved by a decomposition of the upper level problem into one subproblem for each link).

Friesz et al (1989) also tested both scenarios, using a method known as Simulated Annealing (SA). This is a method inspired by the natural cooling processes that can be described by statistical mechanics. Its advantage is that it (in theory) will always converge to the global minimum. Basically, the algorithm is an iterative method in which there is always a certain probability that a worse solution will be accepted over the previous better one.

Suh and Kim (1992) tested the first scenario using an algorithm denoted the Bilevel Descent Algorithm (BDA). This algorithm is similar to ours except that Suh and Kim used a simpler

Iterations	Z	Z, recalculated	CPU time (s)	# Eq. problems solved	# Sens. problems solved
0	336.57110783	336.57115629	0.27	1	0
1	280.72206448	280.72206448	2.25	3	16
2	214.02965024	214.02968156	2.80	6	32
3	208.35771015	208.35773196	4.47	11	48
4	202.15340987	202.15335272	5.90	16	52
5	199.86896642	199.86885553	6.63	20	56
6	199.75702114	199.75694079	7.16	22	59
7	199.63496438	199.63482357	7.69	24	62
8	199.62544206	199.62538689	8.16	27	64
9	199.62532661	199.62527217	9.15	31	78
10	199.62531933	199.62526459	9.62	34	80
11	199.62531846	199.62526393	10.12	37	83
12	199.62531835	199.62526392	11.17	46	85
13	199.62531834	199.62526393	11.88	50	88
14	199.62531834	199.62526393	15.11	82	90

Table 24: Convergence characteristics of the sensitivity analysis-based network design algorithm on the Harker and Friesz network. Demand scenario I: $d_{1,6} = 5, d_{6,1} = 10$. Upper bound: $y^u = 20$.

method of sensitivity analysis, similar to the Tobin and Friesz method (see Section 3.1), and that they utilized the Frank-Wolfe algorithm to solve equilibrium problems.

Huang and Bell (1998) tested the second scenario using all of H-J, EDO, and SA.

Here we will compare all these results with the ones we have obtained using our SBD algorithm. In cases where a researcher has published several results for the same algorithm, using different parameters or starting values, the best of these results have been included in the comparison. Tables 26 and 27 below show the comparison of the results. To compare the Z-values accurately, each Z was recalculated from the corresponding solution vector (y) using the DSD algorithm. A very strict tolerance level(1e-10) was used. It was found that many of the previously published solutions really had objective function values that differed quite significantly from what had been stated. Both the originally reported objective function values and those recalculated are therefore presented in the tables.

It may seem that in the first scenario (Table 26), the H-J solution due to Suwansirikul et al is quite different from the others and could possibly therefore be in the vicinity of a different local optimum. However, this is not the case, which we can see by graphing the values of Z along a straight line through this solution and the solution produced by the sensitivity analysis-based algorithm. Such a graph is displayed in Figure 18 below. Apparently, Z is differentiable and even convex on this one-dimensional subset between the two solutions. Hence, it is clear that Suwansirikul et al's solution is in the same region as the others (we have also checked that the objective function is differentiable at this solution, so it is not on a region boundary either).

The other solutions are all very similar in that links 6 and 16 are only ones with positive capacity enhancement. In order to investigate whether there were several local optima in the plane defined by $y_a = 0, a \neq 6, 16$, Z was graphed on this plane. The result is shown in Figure 19 below. Clearly, there is only one local optimum in this plane. The bottom figure is a close-up of the optimum.

Iterations	Z	Z, recalculated	CPU time (s)	# Eq. problems solved	# Sens. problems solved
0	5756.59154620	5756.59174315	1.16	1	0
1	771.74251321	771.74239424	9.06	10	16
2	737.21744233	737.21719812	9.35	12	32
3	546.71853021	546.71833235	10.28	14	48
4	540.45969467	540.45954299	11.14	18	55
5	535.11964998	535.11947312	11.70	20	62
6	531.06074221	531.06062993	12.33	22	69
7	529.72417303	529.72407088	12.88	24	75
8	529.24324023	529.24310163	13.46	26	81
9	527.38222206	527.38204275	14.45	28	96
10	526.17884191	526.17867092	15.04	30	102
11	525.40732853	525.40724007	15.75	34	108
12	523.95752574	523.95732442	16.36	36	115
13	522.84278875	522.84262572	16.94	38	121
14	522.78601249	522.78582163	17.51	40	127
15	522.73631617	522.73613459	18.50	42	142
16	522.72791881	522.72771717	19.07	44	148
17	522.70192736	522.70173322	19.64	46	154
18	522.59679585	522.59659555	20.28	48	161
19	522.59099689	522.59078968	20.85	50	167
20	522.58572696	522.58552039	21.41	52	173
21	522.58462899	522.58442003	22.38	54	188
22	522.58320074	522.58299150	22.89	56	194
23	522.58295851	522.58274805	23.40	58	200
24	522.58267090	522.58246030	23.94	60	207
25	522.58262935	522.58241849	24.45	62	213
26	522.58261429	522.58240314	24.97	64	219
27	522.58260463	522.58239343	26.05	66	234
28	522.58260151	522.58239015	26.65	68	240
29	522.58259817	522.58238688	27.21	70	246
30	522.58257091	522.58235951	27.82	72	253
31	522.58256929	522.58235779	28.39	74	259
32	522.58256902	522.58235744	29.16	78	265
33	522.58256885	522.58235739	30.17	80	280
34	522.58256880	522.58235732	30.72	82	286
35	522.58256879	522.58235731	31.47	85	292
36	522.58256878	522.58235730	32.66	93	299
37	522.58256876	522.58235730	33.26	95	305
38	522.58256876	522.58235730	36.51	125	311

Table 25: Convergence characteristics of sensitivity analysis-based network design algorithm on the Harker and Friesz network. Demand scenario II: $d_{1,6} = 10, d_{6,1} = 20$. Upper bound: $y^u = 50$.

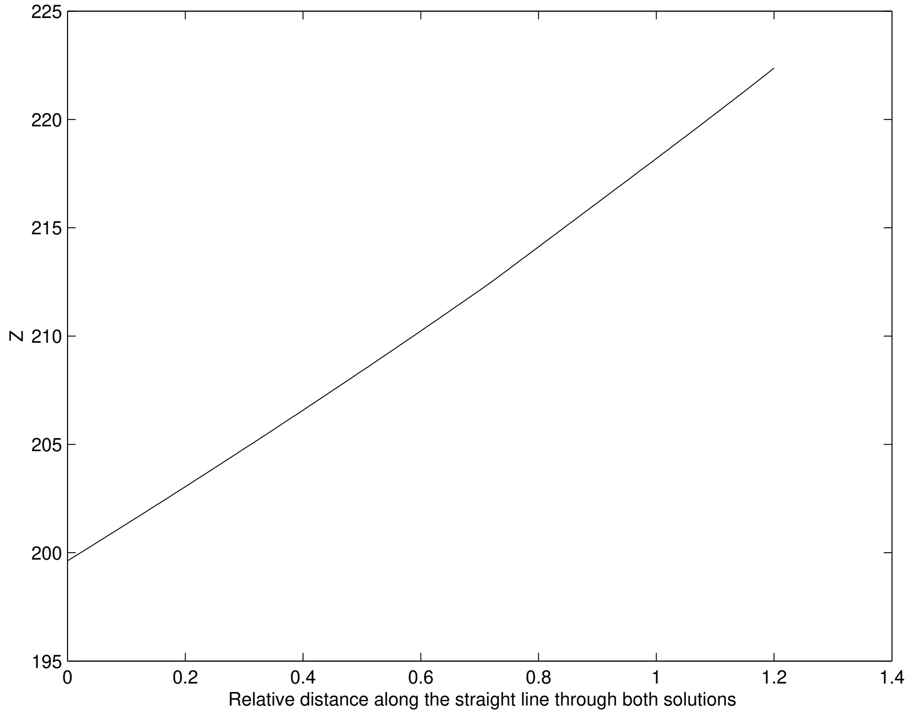


Figure 18: Graph of objective function along straight line through two solutions. X-axis shows relative distance (i.e. sensitivity analysis-based solution is at x=0, Suwansirikul et al's H-J solution is at x=1). Note that x-values below zero here are not feasible for the design since some enhancement parameters y would become negative.

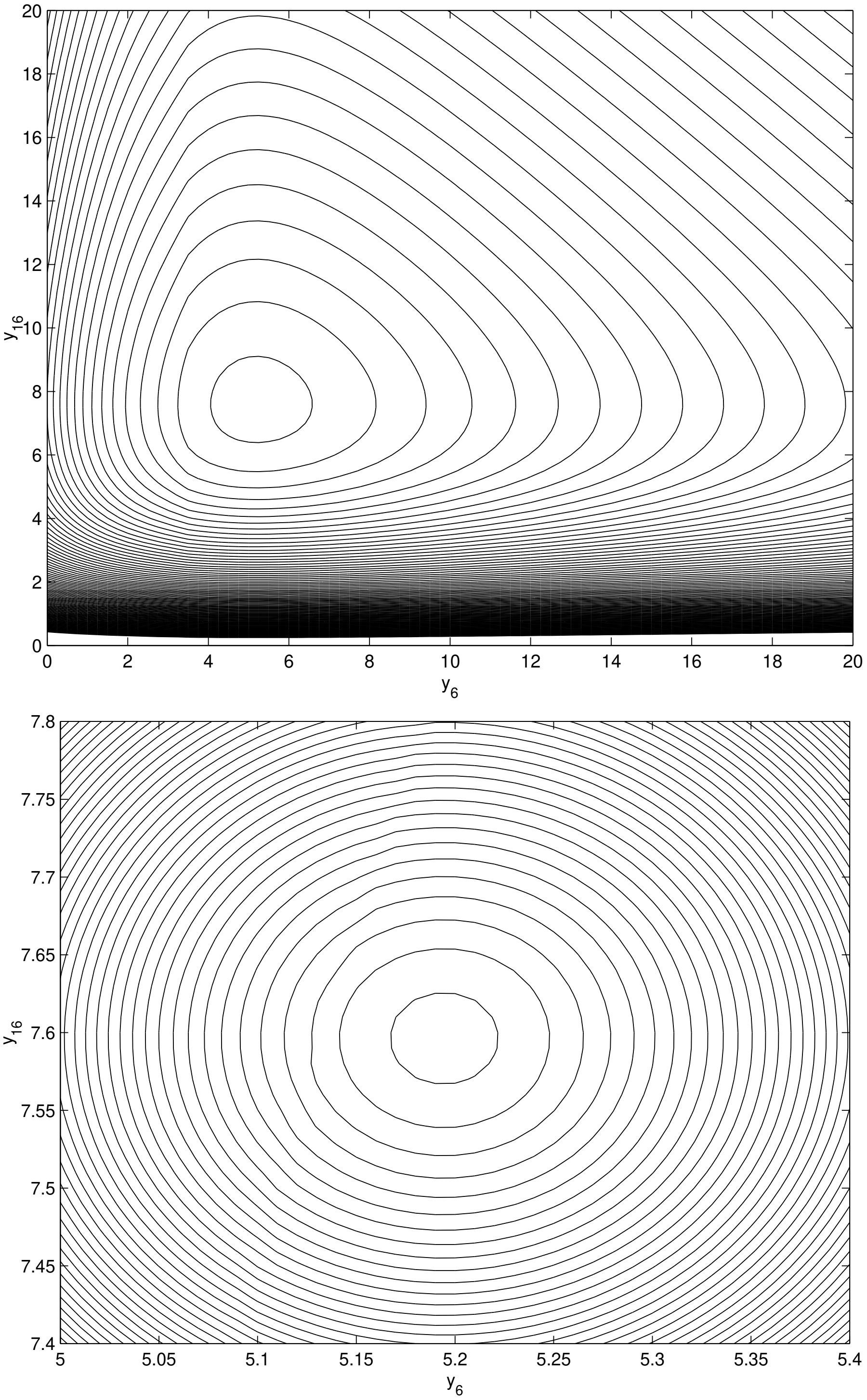


Figure 19: Graph of objective function on plane defined by all but two enhancement parameters being zero. Top diagram: contour equidistance is 1, lowest contour is at $Z = 200$, highest contour is at $Z = 300$ (hence the white space at the bottom of the figure). Bottom diagram: contour equidistance is 0.0005, lowest contour is at $Z = 199.6255$.

	SBD	Suw.:H-J	Suw.:EDO	Friesz:SA	Suh:BDA
Initial value y^0	0	0	-	-	-
Upper bound y^u	20	-	20	20	-
y_1					
y_2					
y_3		1.2	0.13		
y_4					
y_5					
y_6	5.1944681	3.0	6.26	3.1639	5.11
y_7					
y_8					
y_9					
y_{10}					
y_{11}					
y_{12}					
y_{13}					
y_{14}					
y_{15}		3.0	0.13		
y_{16}	7.5964176	2.8	6.26	6.7240	5.71
Z, reported	199.62532	215.08	201.84	198.10378	202.18
Z, recalc.	199.62526	218.20	201.20	201.33577	200.66

Table 26: Results for network design on the Harker and Friesz network. Demand scenario I: $d_{1,6} = 5, d_{6,1} = 10$. It should be noted that, while Z-values may seem relatively close to each other, this is largely due to a large 'constant' in Z because of the zero-congestion link costs.

	SBD	SBD	Suw.:H-J	Suw.:EDO	Friesz:SA	Huang:H-J	Huang:EDO	Huang:SA
Initial value y^0	0	0	0	-	-	0	0	0
Upper bound y^u	20	50	-	20	20	-	20	20
y_1						0.009	0.005	
y_2	4.6144255	4.6144255	5.40	4.88		4.613	4.616	1.230
y_3	9.8091762	9.9102327	8.18	8.59	10.1740	9.872	12.341	10.432
y_4							0.005	
y_5							0.005	0.008
y_6	7.7989278	7.3744793	8.10	7.48	5.7769	5.859	7.659	7.332
y_7			0.00	0.26			0.005	
y_8	0.5922391	0.5922367	0.90	0.85			0.593	0.207
y_9							0.005	
y_{10}							0.005	
y_{11}							0.005	
y_{12}							0.019	
y_{13}							0.005	
y_{14}	1.3152551	1.3152551	3.90	1.54		0.816	1.314	2.238
y_{15}	19.0864543		8.10	0.26		0.591	0.005	0.022
y_{16}	0.8823583	20.7661321	8.40	12.52	17.2786	17.288	19.995	19.472
Z, reported	557.14050943	522.58256876	557.22	540.74	528.497	526.495	525.973	531.931
Z, recalc.	557.14050943	522.58256876	561.49	540.20	533.329	526.513	525.871	532.327

Table 27: Results for network design on the Harker and Friesz network. Demand scenario II: $d_{1,6} = 10, d_{6,1} = 20$.

For the second scenario (Table 27), we clearly have solutions in the vicinity of different local optima. A Z-graph similar to the one in Figure 18 is shown in Figure 20 below. Here, the solutions obtained using the sensitivity analysis-based algorithm with an upper bound of 50 is mapped to 0 on the abscissa, while the solution obtained using the sensitivity analysis-based algorithm with an upper bound of 20 is mapped to 1 on the abscissa. Clearly, here we have non-convexity and even non-differentiability between the two solutions. We could have guessed this already when looking at Table 27 since the SBD-execution with $y^u = 20$ terminated with a solution not at the upper bound.

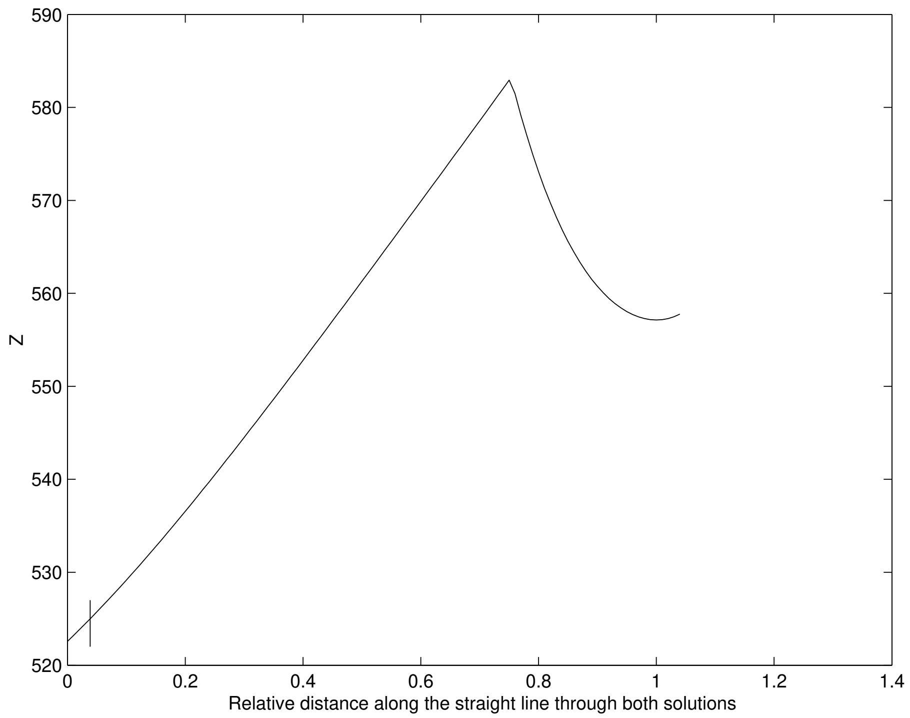


Figure 20: Graph of objective function along straight line through two solutions. The abscissa (x-axis) shows relative distance (i.e. the solutions are at x=0 and x=1). Note that x-values below zero or above one are not feasible for the design since some enhancement parameters y would become negative. The vertical line near x=0.004 shows the end of the feasible set if $y^u = 20$ rather than $y^u = 50$.

In Figure 21, a similar graph is shown between the SBD solution (with upper bound 50) at 0 on the abscissa and the solution from Friesz et al's SA method at 1 on the abscissa. Clearly, Friesz et al's solution is in the vicinity of yet another local optimum.

9.1.3 Sioux Falls network

This is the same network as was discussed in Section 7.1.5, except that all OD-pair demands have been multiplied by 1.1. Network design on this network was first performed by Suwansirikul et al (1987), who used this modified demand matrix to model peak-hour conditions. Suwansirikul et al

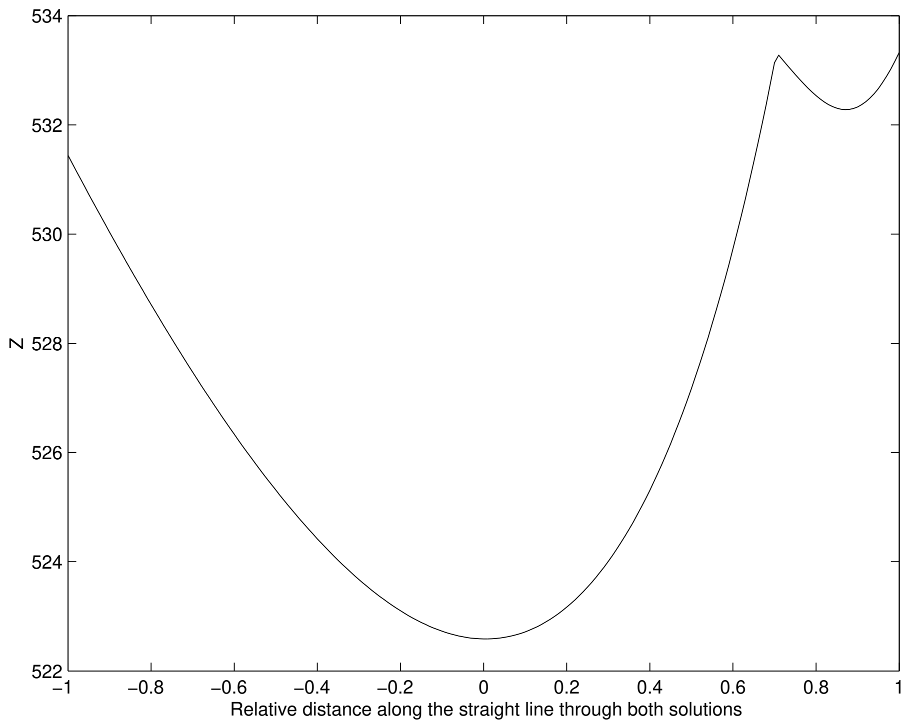


Figure 21: Graph of objective function along straight line through two solutions. The abscissa (x-axis) shows relative distance (i.e. the solutions are at x=0 and x=1). Note that x-values below zero are not feasible for the design since some enhancement parameters y would become negative.

used Hooke-Jeeves (H-J) and Equilibrium Decomposed Optimization (EDO). Capacity augmentations were allowed only on a set of 10 links that were chosen as candidates for enhancement. The investment cost functions used for these links are described in Suwansirikul et al (1987).

The settings used for all the executions of the SBD algorithm on the Sioux Falls network are summarized in Table 28. The convergence characteristics of the sensitivity analysis-based algorithm on the Sioux Falls network are illustrated in Table 29 below. The initial solution used was $y^0 = 0$. As before, more accurate Z-values that were obtained by applying the DSD algorithm with a very strict tolerance (1e-10) to the design solution at each stage is shown as well as the Z-values directly from the algorithm.

Number of main iterations	12
Armijo factor, τ	0.01
Armijo multiplier, κ	2
Turns before freezing	3
Turns frozen	5
DSD route generation algorithm	Newton-based
DSD error tolerance, S_ε	1e-6
Sensitivity analysis tolerance, $\mathcal{S}_\varepsilon$.	1e-4
Sensitivity analysis iteration limit	5000

Table 28: Algorithm parameters used for the Sioux Falls network

Iterations	Z	Z, recalc.	CPU time (s)
0	99.9412	99.9413	36
1	87.9257	87.9248	321
2	80.4735	80.4735	681
3	80.4298	80.4294	1390
4	80.3262	80.3260	2242
5	80.0948	80.0944	2942
6	80.0145	80.0137	3899
7	80.0040	80.0035	5114
8	80.0009	80.0007	5942
9	80.0006	80.0000	6423
10	80.0003	80.0002	7755
11	79.9995	79.9987	8228
12	79.9969	79.9961	9457

Table 29: Convergence characteristics of sensitivity analysis-based network design algorithm on the Sioux Falls network.

Apart from Suwansirikul et al, several other researchers have also tested network design algorithms on this design problem. Friesz et al (1989) used Simulated Annealing (SA). Huang and Bell (1998) tested all of H-J, EDO, and SA. Lim (2002) used an algorithm called the Penalty Interior Point Algorithm (PIPA). This is an iterative method based on the removal of some feasibility constraints for the equilibrium problem in favor of a penalty term in the objective function, according to how infeasible a solution is.

Here we will compare these results with the ones we have obtained using our sensitivity analysis-based method. In cases where a researcher has published several results for the same algorithm, using different parameters or starting values, the best of these results have been included in the comparison. Table 30 below shows a comparison of the results. To compare the network design objective function values (Z) accurately, each value was recalculated from the corresponding design solution vector (y) using the DSD algorithm. A very strict tolerance level (1e-10) was used. It was found that many of the previously published solutions really had objective function values that differed quite significantly from what had been stated. Both the originally reported objective function values and those recalculated are therefore presented in the tables.

	SBD	SBD	SBD	SBD	Suw.:H-J	Suw.:EDO
Initial value y^0	0	2	3	6.25	2	-
Upper bound y^u	25	25	25	25	-	25
y_{16}	5.3027	5.1492	5.3457	5.2773	4.8	4.59
y_{17}	2.0560	2.0214	1.9786	2.0533	1.2	1.52
y_{19}	5.3430	5.1679	5.3741	5.3002	4.8	5.45
y_{20}	1.9901	2.0012	1.9460	2.0369	0.8	2.33
y_{25}	2.5216	2.4945	2.7856	2.7670	2.0	1.27
y_{26}	2.5548	2.5447	2.8245	2.8222	2.6	2.33
y_{29}	2.9883	2.9535	2.9257	3.0124	4.8	0.41
y_{39}	4.8559	4.8330	4.7528	4.7348	4.4	4.59
y_{48}	3.0026	2.9798	2.9732	2.9746	4.8	2.71
y_{74}	4.8496	4.8212	4.7347	4.7511	4.4	2.71
Z, reported	79.9969	79.9968	79.9987	80.0026	80.78	83.08
Z, recalculated	79.9961	79.9971	79.9990	80.0043	80.67	82.34
	Friesz:SA	Huang:H-J	Huang:EDO	Huang:SA	Lim:PIPA	
Initial value y^0	6.25	3	0	6.25	-	
Upper bound y^u	25	-	25	25	-	
y_{16}	5.38	4.507	4.276	5.322	5.4680	
y_{17}	2.26	4.509	2.288	2.596	2.0039	
y_{19}	5.50	4.520	4.080	5.664	5.4471	
y_{20}	2.01	4.052	1.618	1.309	1.9395	
y_{25}	2.64	4.299	1.654	2.498	2.9448	
y_{26}	2.47	2.949	1.130	2.732	2.8191	
y_{29}	4.54	3.000	3.219	4.123	3.4039	
y_{39}	4.45	3.601	3.326	4.508	4.8061	
y_{48}	4.21	3.006	1.981	3.736	3.2364	
y_{74}	4.67	3.200	3.190	3.903	4.7779	
Z, reported	80.87	83.316	83.703	81.983	80.8669	
Z, recalculated	80.42	81.185	81.345	80.304	80.0528	

Table 30: Results for network design on the Sioux Falls network

Clearly, the sensitivity analysis-based network design algorithm achieves much better solutions than those previously reported for the other algorithms. However, the poor performance of other algorithms in the previously published results may in part be due to the low accuracy used in solving the equilibrium problems – the large discrepancy between the reported and accurately recalculated Z-values of these results certainly suggests this. It seems likely that all four solutions produced by the sensitivity analysis-based algorithm is in the vicinity of the same local optimum. As for the other solutions, they are not accurate enough (i.e. near enough to a local optimum) to make a qualified guess about whether they are in the region of this same optimum or not. The fact that all Z-values are quite close together does not in itself indicate that all solutions are near the same optimum, particularly as the design objective function for this problem is very flat.

9.2 Performance and convergence issues

9.2.1 Further discussion of non-differentiability

As we have seen, the traffic flows (and hence the network design objective function) are generally non-differentiable at points corresponding to non-strictly complementary solutions. In particular, we (generally) have hyper-surfaces of non-differentiabilities between regions in the space of feasible enhancements corresponding to solutions with different sets of cheapest routes. It is important to consider, then, how the non-differentiability at these points will affect our network design algorithm. Now, these points are so few (in probability measure terms) that we should never expect to arrive at one. Nevertheless, in practice this might happen (if, e.g., a non-differentiable point is chosen as a starting point), so we should examine what would happen. Clearly, if the point in question is on a ”ridge” (i.e. the directional derivative is negative in both directions, see Figure 22), then the non-differentiability will not affect the algorithm, as the di-

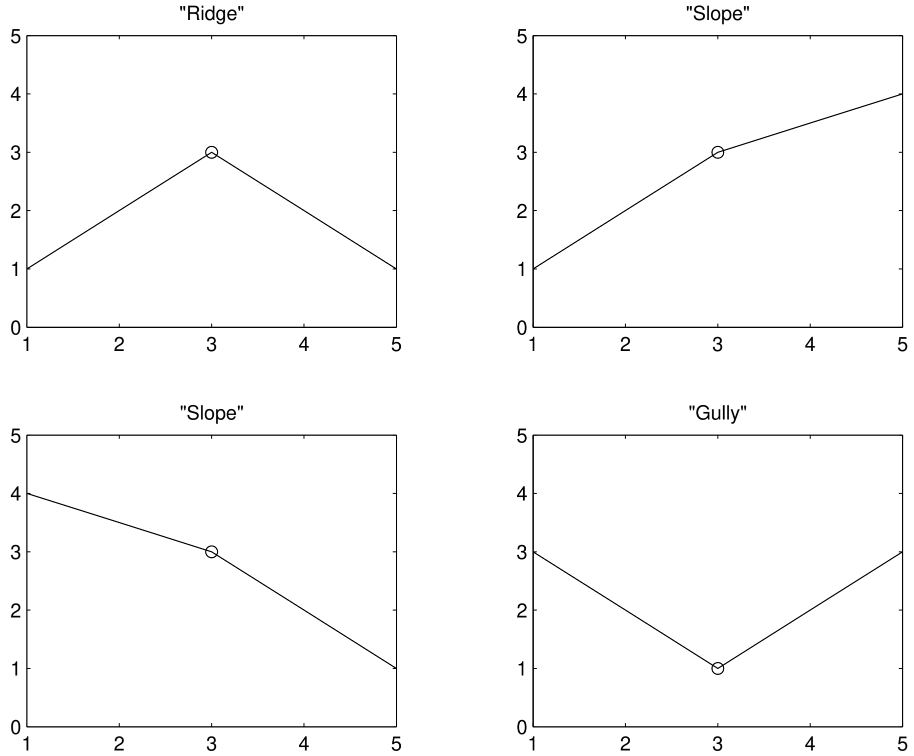


Figure 22: Characterization of non-differentiable points. For clarity and simplicity, a one-parameter case is illustrated.

rectional derivative utilized will be correct in the search direction of the parameter in question. Also, if the non-differentiability is on a ”slope” (see Figure 22), the direction if not the magnitude of the search direction will be correct, so it will be possible to take at least a short step in the search direction and thus arrive at a differentiable point. However, if the non-differentiable point is in a ”gully” (again, see Figure 22), an incorrect search direction will be obtained. In this case, it is possible that no step in the search direction will be satisfy the Armijo criterion, especially if the gully is very steep. Consequently, the algorithm can get stuck at the non-differentiable point and therefore terminate. Of course, it is possible to avoid this potential (but unlikely) problem by a simple implementation modification. By having the algorithm check if any parameters are non-differentiable (and have the ”gully”-characterization) when the algorithm gets stuck, the situation can be rectified. Any stuck search direction components are simply set to zero.

One may also note that, in most well known problems, the non-differentiable hyper-surface seem to be ”ridges”. However, it is possible to construct problems with ”gullies”.

9.2.2 Non-uniqueness of local optima

Related to the issue of non-differentiability is that of non-uniqueness of local optima in the network design problem (the global optimum may also be non-unique, but that is certainly not a problem as long as one of the global solutions is found). A natural consequence of there being separate regions (or cells) corresponding to sets of used routes and often separated by ”ridges” is that each region may have its own local minimum/minima. We have met with this issue in several of the examples we have considered. Note, however, that in these BPR-function examples, we have not yet come upon a problem where we have found more than one local minimum in a region. A very interesting question that arises is whether there are ”reasonable” conditions on the link cost functions and investment cost functions that guarantee that each region has only a single local minimum. If there are such conditions, and real-life networks can be modeled without violating them, it would clearly allow us to use new methods to find the global optimum of the network design objective function. For instance, an algorithm could somehow make intelligent guesses about which regions are likely to hold a good local optima and search each of these regions using our algorithm together with a projection (in order to make sure the algorithm stays in the right region).

Note that one of the conditions would have to be that the investment cost functions are convex, something that may not always be realistic due to real-life economies of scale.

9.2.3 Impact of DSD algorithm and sensitivity algorithm tolerances

Table 31 below shows results for network design on the Harker and Friesz network using a variety of different accuracies (i.e. tolerances) for the DSD and sensitivity analysis algorithms. The network design setup was the same as was used in Suwansirikul et al (1987) – Network 3, case 3. The number of network design iterations was fixed at 20 iterations. The Armijo factor τ was 0.01. The Armijo multiplier κ was 2. An upper bound of 5000 sensitivity iterations was applied for each execution of the sensitivity algorithm. The initial solution used was zero capacity augmentation for all links.

The most immediately apparent feature of the results is that the two bottom executions somehow arrived at much better solutions. However, a study of the solutions shows that they are all in the same vicinity, and that it is not a question of different local optima. It seems then to be pure chance resulting from numerical factors that let the two particular executions pass early through some ”threshold”. If the algorithm was run until termination (i.e. until it got stuck) in each case, the result would likely be different.

DSD tolerance	Sensitivity tolerance	CPU time (s)	Z
1e-10	1e-8	135.65	522.5855174893
1e-10	1e-3	122.01	522.5855165764
1e-8	1e-3	30.74	522.5855175517
1e-7	1e-5	30.28	522.5855176091
1e-5	1e-5	27.93	522.5855201422
1e-6	1e-4	24.72	522.5855203887
1e-5	1e-3	19.63	522.5855191022
1e-6	1e-5	28.85	522.5823572701
1e-4	1e-2	14.47	522.5825254492

Table 31: Value of network design objective function for solutions obtained using different combinations of error tolerances.

Clearly, using a stricter DSD tolerance (for the same sensitivity tolerance) here costs relatively little extra CPU time while improving accuracy. On the other hand, the sensitivity tolerance should be probably be kept less strict than the DSD tolerance.

Note that the objective function values listed in the table above were in each case obtained from the solution by solving the relevant DSD problem using a very strict tolerance – these objective function values will therefore in the many cases differ from those used internally during the network design execution. Of course, the more exact objective function values obtained using a more accurate DSD execution will better represent the accuracy of the network design solution.

9.2.4 Choice of Armijo parameters

Choosing a lower τ (the Armijo factor, see Section 4.2) clearly allows longer steps to be taken, which normally means convergence in fewer iterations. Furthermore, for reasons of limited numerical precision, longer steps usually also means the ability to reach a better final design solution. The explanation for this is that there always comes a point when the step taken becomes smaller than the limits of numerical precision. When this happens, the algorithm cannot make further progress. For these reasons, a τ of 0.01 was used for most of the executions.

Choosing the Armijo multiplier, κ, is a trade off between having accurate line searches and having to solve as few extra equilibrium problems as possible. Since each equilibrium problem is quite expensive, decreasing κ will slow down each iteration significantly. On the other hand, using a very high κ may lead to a situation where many short steps are taken in the same direction (with expensive sensitivity calculations in between) when a single longer step may have been satisfactory with a more reasonable κ. We normally used a κ of 2.

9.2.5 Mechanisms for improving performance

See Section 6.3.5 for a description of these mechanisms.

Parameter 'Freezing' The parameter freezing mechanism saved a lot of CPU-time in the executions for some of the network design problems, notably those on the Harker and Friesz network. Typically, those design parameters that would end up at a bounding value reached those values already after one or two iterations. 'Freezing' these parameters reduced the computational effort spent on sensitivity problem by a lot (sometimes as much as 80%).

Hot start DSD implementation The 'hot start' mechanism for speeding up the calculation of equilibrium solutions during the network design process seemed like a good idea in theory. However, in the few tests that we performed, it did not deliver any significant performance improvements. This mechanism may still be useful with some enhancements (e.g., perhaps a more intelligent starting guess than the previous solution itself can be used). More research is needed.

Variable DSD tolerance This mechanism was also not particularly useful. Although using slightly looser error tolerances in calculating equilibrium solutions during the line searches does save some time, this gain is often offset by the need to calculate the final equilibrium twice (since the higher accuracy is required for the equilibrium solution at next iteration). On the other hand, this mechanism would likely give a greater performance advantage if for some reason we wanted to use a low κ.

10 Conclusions

10.1 Algorithms performance for TAP

In this thesis, we have seen that DSD-type algorithms can produce very accurate solutions to traffic assignment problems. The DSD-algorithms implemented were vastly superior to the Frank-Wolfe algorithm, performance-wise. For large networks, the DSD-implementation utilizing the more advanced Newton-type search direction generation turned out to be significantly better than a DSD-implementation using the simple projection of a point on the gradient. For small networks and for networks with many OD-pairs compared to the number of links, we have found the more computationally intensive Newton-based method to be slightly less efficient than the gradient-based method, although the numbers here are not as clear. In particular, we have observed that at very high accuracies (likely for small problems), numerical imprecision related to the implementation has a significant effect on the algorithms.

We have further seen that the settings used in an implementation can make a large difference in algorithm performance. In particular, one can gain a performance advantage by tweaking parameters in order increase or reduce the relative amount of time spent on solving restricted master problems and shortest route problems, respectively. For a large and complex network with relatively few OD-pairs, the number of shortest-route problems solved should be minimized in order to increase performance. By contrast, for a small network with many OD-pairs, such as the Sioux Falls network, the CPU time-consumption for shortest-route calculations is almost negligible compared to the effort required to solve the restricted master problems. The importance of these factors is of course implementation-specific.

10.2 Sensitivity analysis

We have proposed a new method for sensitivity analysis of traffic equilibria and shown under what conditions it works. We have shown that, unlike older methods, this method will provide a result for any traffic equilibrium satisfying our assumptions (in particular, the assumption of strictly increasing route cost functions), and that the result obtained will be useful. Specifically, unique cost derivatives are always provided, and the link cost derivatives obtained are guaranteed to be unique except for those links that have zero link cost derivative at the current equilibrium. Furthermore, we have shown that the method can be implemented in practice by making use of it for network design.

10.3 Network design

In network design, we have seen that our sensitivity analysis-based method is able to find, very accurately, local optima of the capacity enhancement problem. In fact, the results obtained using our method were more accurate than all previously published results we looked at. Performance-wise, we have observed that it is crucial to solve the regular equilibrium problems very accurately in order to obtain an accurate network design solution. We noted that many researchers using others methods had not done this (although not all the methods used equilibrium solutions during the optimization phase), as the corresponding published results were not completely consistent with the claimed objective function values for the network design problem.

By looking at graphs and using simple reasoning, we have touched at the study of the structure of the network design problem. We have noted that the objective function is non-differentiable exactly on the hyper-surfaces between regions (in the space of feasible enhancements) corresponding to solutions with different sets of cheapest routes. We have noted that each region may have its own local optima, and seen simple examples of such cases. We have also observed that our algorithm may fail to find the best local optima even in a simple case.

An interesting topic for future research is to further investigate the structure of the network design problem in order to try to exploit to construct more efficient methods. In particular, an interesting question is whether there are reasonable conditions of the problem that guarantees that each region of the feasible set will have at most one local optimum.

References

[1] M.G.H. Bell and Y.Iida, *Transportation Network Analysis*, John Wiley & Sons, Chichester, 1997.

[2] H-J Cho, T.E. Smith, T.L. Friesz, *A reduction method for local sensitivity analyses of network equilibrium arc flows*, Transportation Research, Part B, 34 (2000).

[3] A.L. Dontchev and R.T. Rockafellar, *Ample parameterization of variational inclusions*, SIAM Journal on Optimization, 12 (2002).

[4] T.L. Friesz, H-J. Cho, N.J. Mehta, R.L. Tobin, G. Anandalingam *A Simulated Annealing Approach to the Network Design Problem with Variational Inequality Constraints*, 1990.

[5] P.T. Harker and T.L. Friesz, *Bounding the Solution of the Continuous Equilibrium Network Design Problem*, in *Proceedings of the 9th International Symposium on Transportation and Traffic Theory*, VNU Science Press, Delft, Netherlands, 1984.

[6] H.-J. Huang and M.G.H. Bell, *Continuous Equilibrium Network Design Problem with Elastic Demand: Derivative-Free Solution Methods* in Transportation Networks: Recent Methodological Advances, ed. M.G.H. Bell, Elsevier Science, 1998.

[7] T. Larsson and M. Patriksson, *Simplicial decomposition with disaggregated representation for the traffic assignment problem*, Transportation Science, 26 (1992).

[8] A. Migdalas, *Bilevel Programming in Traffic Planning: Models, Methods and Challenge*, Journal of Global Optimization, 7 (1995).

[9] S. Nguyen and C. Dupuis, *An Efficient Method for Computing Traffic Equilibria in Networks with Assymetric Transportation Costs*, Transportation Science, 18 (1984).

[10] M. Patriksson, *The Traffic Assignment Problem - Models and Methods*, Topics in Transportation, VSP BV, Utrecht, The Netherlands, 1994.

[11] P.A. Steenbrink, *Optimization of Transport Networks*, John Wiley & Sons, London, 1974.

[12] S. Suh and T.J. Kim, *Solving nonlinear bilevel programming models of the equilibrium network design problem: a compariative review* in Hierarchical Optimization, G. Anandalingam and T.L. Friesz, eds., vol. 34 of Annals of Operations Research, J.C. Baltzer AG, Basel, Switzerland, 1992.

[13] C. Suwansirikul, T.L. Friesz, R.L. Tobin *Equilibrium Decomposed Optimization: A Heuristic for the Continuous Equilibrium Network Design Problem*, Transportation Science, 21 (1987).

[14] R.L. Tobin and T.L. Friesz, *Sensitivity analysis for equilibrium network flow*, Transportation Science, 22 (1988).

Appendix

A Program Code

A.1 Shell program / user interface

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nw_solver.m
%
% Tool for solving and illustrating traffic assignment problems.
%
% FUNCTION DEPENDENCIES:
%   * Uses a solver function specified in 'solver_fun'
%   * Uses a sensitivity solver function spec. in 'sens_solver'
%   * Uses the function 'nw_drawer' to illustrate the solution.
%
% INPUT
%   * Reads data from four text files:
%
%    'links_file'  : a text file w/ one row for each link;
%                    columns 1/2 = start/end nodes; columns 3+ = cost data on polynomial form
%                    all rows should have the same number of rows, i.e. add extra zeros if some
%                    links have lower degree cost polynomials than others.
%    'demands_file': a text file with one row for each OD-pair;
%                    columns 1/2 = origin/destination nodes; column 3 = demand.
%    'nodes_file'  : a text file with one row for each node;
%                    columns 1/2 = node coord's (x/y) in a planar representation of the network
%    'sens_file'   : a text file with one row for each link;
%                    columns 1 = '-value of link paramater. columns 2+ = cost data on polynomial form
%                    (cost as a function of link parameter. column2 = degree 1 etc.)
%
% OUTPUT
%   * Logs solution data to the file specified in 'sol_log_file'
%
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%

links_file    = 'braess_l.dat';
demands_file  = 'braess_d.dat';
nodes_file    = 'braess_r.dat';
sens_file     = 'sioux_s.dat';

sol_log_file  = 'nw_solution.mat';

solver_fun    = 'nw_dsd_solver';
sens_solver   = 'nw_sens_solver';

enforce_feasibility = 1;      % if 1, eliminates small feasibility errors in solution

max_menu_size = 38;           % max menu items to display simultaneously

%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

has_solution = 0;
has_sens     = 0;

% Iterate UI
while 1

    % Display main menu
    %%%%%%%%%%%%%%%%%%%%%%
    if has_solution & has_sens
        quit_choice = 12;
        load_choice = -1;
        choice = menu('          Please choose an action:           ',...
            'Solve TAP using DSD solver', 'Solve sensitivity problem', 'Show solution info',...
            'Draw network with total link flows', 'Draw network with OD-pair link flows',...
            'Draw network with link costs', 'Overview of routes and route costs',...
            'Draw networks with OD-pair routes', 'Draw network with link flow derivatives',...
            'Draw network with OD-pair link flow derivatives',...
            'Draw networks with route flow derivatives', 'Quit');

    elseif has_solution
```

```
    quit_choice = 9;
    load_choice = -1;
    choice = menu('          Please choose an action:           ',...
        'Solve TAP using DSD solver', 'Solve sensitivity problem', 'Show solution info',...
        'Draw network with total link flows', 'Draw network with OD-pair link flows',...
        'Draw network with link costs', 'Overview of routes and route costs',...
        'Draw networks with OD-pair routes', 'Quit');

elseif exist(sol_log_file) == 2
    load_choice = 2;
    quit_choice = 3;
    choice = menu('          Please choose an action:           ',...
        'Solve TAP using DSD solver', 'Load previous solution from autolog', 'Quit');
else
    load_choice = -1;
    quit_choice = 2;
    choice = menu('          Please choose an action:           ',...
        'Solve TAP using DSD solver', 'Quit');
end


%%%%%%%%%%%%%%%%%%%%%%


if choice == quit_choice
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Quit
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    break;


elseif choice == load_choice
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Load previous solution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    has_solution = 1;
    clear param_data;
    load(sol_log_file);
    if exist('param_data') == 1
        has_sens = 1;
    else
        has_sens = 0;
    end


elseif choice == 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Solve TAP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Load network
    links   = load(links_file);
    nodes   = load(nodes_file);
    demands = sortrows(load(demands_file), [1 2]);
    l_l     = size(links,1);
    l_d     = size(demands,1);
    l_n     = size(nodes,1);

    % Calculate solution
    [routes, num_routes, route_flows, route_costs, link_flows, link_costs]...
        = eval(strcat(solver_fun, '(links, demands, l_n)'));

    % Adjust flows slightly to remove small infeasibilities
    if enforce_feasibility

        % Calculate feasibility error before adjustment
        f_errors = zeros(l_d,1);
        for pq = 1:l_d
            f_errors(pq) = sum(route_flows{pq}) - demands(pq, 3);
        end
        old_feas_error = sum(abs(f_errors))/sum(demands(:,3));

        % Adjust route flows
        for pq = 1:l_d
            [trash,lo_cost_route_idx] = min(route_costs{pq});
            [trash,hi_cost_route_idx] = max(route_costs{pq});
            missing_flow = demands(pq,3) - sum(route_flows{pq});
            if missing_flow > 0
                route_flows{pq}(lo_cost_route_idx) = route_flows{pq}(lo_cost_route_idx) + missing_flow;
            elseif route_flows{pq}(hi_cost_route_idx) >= -missing_flow
                route_flows{pq}(hi_cost_route_idx) = route_flows{pq}(hi_cost_route_idx) - (-missing_flow);
            else
                idxs = find(route_flows{pq} >= -missing_flow);
                [trash,ok_route_idx] = max(route_costs{pq}(idxs));
                route_flows{pq}(idxs(ok_route_idx)) = route_flows{pq}(idxs(ok_route_idx)) - (-missing_flow);
            end
        end
```

```
    % Recalculate link flows
    link_flows = zeros(l_l,1);
    for pq = 1:l_d
        for r = 1:num_routes(pq)
            link_flows(routes{pq}{r}) = link_flows(routes{pq}{r}) + route_flows{pq}(r);
        end
    end

    % Recalculate link costs
    link_costs = links(:,3);
    for degree = 1:(size(links, 2) - 3)
        link_costs = link_costs + links(:,degree+3) .* link_flows.^degree;
    end

    % Recalculate route costs
    for pq = 1:l_d
        for r = 1:num_routes(pq)
            route_costs{pq}(r) = sum(link_costs(routes{pq}{r}));
        end
    end

    % Calculate new feasibility error
    f_errors = zeros(l_d,1);
    for pq = 1:l_d
        f_errors(pq) = sum(route_flows{pq}) - demands(pq, 3);
    end
    feasibility_error = sum(abs(f_errors))/sum(demands(:,3));

    disp(sprintf('Feasibility error reduced from %3.6g to %3.6g.', old_feas_error, feasibility_error));

else

    % Calculate feasibility error
    f_errors = zeros(l_d,1);
    for pq = 1:l_d
        f_errors(pq) = sum(route_flows{pq}) - demands(pq, 3);
    end
    feasibility_error = sum(abs(f_errors))/sum(demands(:,3));

end

% Calculate OD-pair link flows
odpair_link_flows =  zeros(l_l, l_d);
for pq = 1:l_d
    for r = 1:num_routes(pq)
        odpair_link_flows(routes{pq}{r}, pq) = odpair_link_flows(routes{pq}{r}, pq) + route_flows{pq}(r);
    end
end

% Calculate total, average, and OD-pair costs
total_cost   = link_flows' * link_costs;
avg_cost     = total_cost / sum(demands(:,3));
odpair_costs = odpair_link_flows' * link_costs ./  demands(:,3);

% Calculate objective function value
obj_fun_val = 0;
for degree = 1:(size(links,2) - 2)
    obj_fun_val = obj_fun_val + sum(links(:,degree+2) .* link_flows.^degree / degree);
end

% Calculate a lower bound on objective function
% (rfs_c/lfs_c = equilibrium route/link flows assuming costs were constant = link_costs)
rsd = cell(l_d,1);
for pq = 1:l_d
    rfs_c{pq} = zeros(num_routes(pq), 1);
    [trash, cheapest_route] = min(route_costs{pq});
    rfs_c{pq}(cheapest_route) = demands(pq,3);
end
lfs_c = zeros(l_l,1);
for pq = 1:l_d
    for r = 1:num_routes(pq)
        lfs_c(routes{pq}{r}) = lfs_c(routes{pq}{r}) + rfs_c{pq}(r);
    end
end
obj_fun_lb = obj_fun_val + (lfs_c - link_flows)' * link_costs;

% Calculate route cost differences (rrcd = relative route cost difference)
max_route_costs = zeros(l_d,1);
min_route_costs = zeros(l_d,1);
avg_route_costs = zeros(l_d,1);
avg_diff_costs  = zeros(l_d,1);
surplus_cost    = 0;
for pq = 1:l_d
    live_routes         = find(route_flows{pq} > 0);
    min_route_costs(pq) = min(route_costs{pq});
    max_route_costs(pq) = max(route_costs{pq}(live_routes));
    avg_route_costs(pq) = mean(route_costs{pq}(live_routes));
    avg_diff_costs(pq)  = mean(abs(route_costs{pq}(live_routes) - avg_route_costs(pq)));
```

```
        surplus_cost        = surplus_cost + (route_flows{pq}' * (route_costs{pq} - min_route_costs(pq)));
    end
    avg_avg_rrcd     = mean(avg_diff_costs ./ min_route_costs);
    avg_max_rrcd     = mean((max_route_costs - min_route_costs)./min_route_costs);
    max_max_rrcd     = max((max_route_costs - min_route_costs)./min_route_costs);
    rel_surplus_cost = surplus_cost / (link_flows' * link_costs);

    % Log the solution to file
    save(sol_log_file, 'routes', 'num_routes', 'route_flows', 'route_costs', 'link_flows',...
        'link_costs', 'links', 'nodes', 'demands', 'l_n', 'l_l', 'l_d', 'odpair_link_flows',...
        'obj_fun_val', 'obj_fun_lb', 'avg_avg_rrcd', 'avg_max_rrcd', 'max_max_rrcd',...
        'rel_surplus_cost', 'feasibility_error', 'avg_cost', 'total_cost', 'odpair_costs');

    has_solution = 1;
    has_sens     = 0; % make sure new solution isn't matched with old sens. analysis


elseif choice == 2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Sensitivity Analysis
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % load data
    param_data = load(sens_file);

    % Calculate flow derivatives
    [d_link_flows, d_link_costs, d_route_flows, d_route_costs, routes2, free_routes, total_routes...
            init_sum_d_route_cost_diff, sum_d_route_cost_diff]...
        = eval(strcat(sens_solver, '(routes, num_routes, route_costs, route_flows, link_costs',...
        ', link_flows, param_data, links, demands, l_n)'));

    % Calculate OD-pair link flow derivatives
    d_odpair_link_flows =  zeros(l_l, l_d);
    for pq = 1:l_d
        for r = 1:total_routes(pq)
            d_odpair_link_flows(routes2{pq}{r}, pq) = d_odpair_link_flows(routes2{pq}{r}, pq) + d_route_flows{pq}(r);
        end
    end

    % Calculate total, average, and OD-pair costs derivatives
    d_total_cost   = link_flows' * d_link_costs + d_link_flows' * link_costs;
    d_avg_cost     = d_total_cost / sum(demands(:,3));
    d_odpair_costs = (odpair_link_flows' * d_link_costs + d_odpair_link_flows' * link_costs) ./  demands(:,3);

    % Append sensitivity data to log file
    save(sol_log_file, 'param_data', 'routes2', 'd_route_flows', 'd_link_flows', 'd_route_costs',...
        'd_link_costs', 'd_odpair_link_flows', 'total_routes', 'free_routes', 'd_total_cost',...
        'd_avg_cost', 'd_odpair_costs', 'init_sum_d_route_cost_diff', 'sum_d_route_cost_diff', '-append');

    has_sens = 1;


elseif choice == 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Show solution info
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    disp(' ');
    disp('/--------------------------------------------------------------------');
    text = sprintf('| Objective function value              : %0.16g', obj_fun_val);
    disp(text);
    text = sprintf('| Objective function lower bound        : %0.16g', obj_fun_lb);
    disp(text);
    disp('|');
    text = sprintf('| Max relative objective function error  : %0.16g', (obj_fun_val-obj_fun_lb)/obj_fun_lb);
    disp(text);
    disp('|');
    text = sprintf('| Max relative route cost difference     : %0.16g', max_max_rrcd);
    disp(text);
    text = sprintf('| Avg max relative route cost difference : %0.16g', avg_max_rrcd);
    disp(text);
    text = sprintf('| Avg avg relative route cost difference : %0.16g', avg_avg_rrcd);
    disp(text);
    disp('|');
    text = sprintf('| Relative average surplus cost          : %0.16g', rel_surplus_cost);
    disp(text);
    disp('|');
    text = sprintf('| Relative feasibility error             : %0.16g', feasibility_error);
    disp(text);
    disp('|');
    text = sprintf('| Average travel cost                    : %0.16g', avg_cost);
    disp(text);
    if has_sens
        disp('|');
        disp('|--------------------------------------------------------------------');
        text = sprintf('| After/Before d_route_cost error ratio  : %0.16g', sum_d_route_cost_diff/init_sum_d_route_cost_diff);
        disp(text);
        disp('|');
```

```
        text = sprintf('| Average travel cost derivative         : %0.16g', d_avg_cost);
        disp(text);
    end
    disp('|');
    disp('\--------------------------------------------------------------------');


elseif choice == 4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Draw link flows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    nw_drawer(nodes,links,link_flows,'Traffic for all OD-pairs.', 0, 0);


elseif choice == 5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Draw OD-pair link flows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Iterate submenus... lots of code here just to get the menus working intelligently
    first_od = 1;
    if l_d > max_menu_size - 1
        last_od     = max_menu_size - 2;
        back_choice = last_od + 2;
        more_choice = last_od + 1;
    else
        last_od     = l_d;
        back_choice = l_d + 1;
        more_choice = 0;
    end

    while 1
        text = 'choice2 = first_od - 1 + menu(''Please choose an OD-pair:''';
        for pq = first_od:last_od
            text = strcat(text, sprintf(', ''OD-pair %d -> %d''', demands(pq,1), demands(pq,2)));
        end
        if (more_choice > 0)
            text = strcat(text, ', ''More''');
        end
        text = strcat(text, ', ''Back'');');
        eval(text);
        if choice2 == back_choice
            break;
        elseif choice2 == more_choice
            first_od = first_od + max_menu_size-2;
            if last_od + max_menu_size - 1 >= l_d
                last_od = l_d;
                back_choice = l_d + 1;
                more_choice = 0;
            else
                last_od  = last_od  + max_menu_size-2;
                back_choice = last_od + 2;
                more_choice = last_od + 1;
            end
        else
            pq = choice2;
            % Draw diagram
            name = sprintf('Traffic for OD-pair %d -> %d.   Travel cost = %3.6g.',...
                demands(pq,1), demands(pq,2), odpair_costs(pq));
            nw_drawer(nodes,links,odpair_link_flows(:,pq),name,demands(pq,1),demands(pq,2));
        end
    end


elseif choice == 6
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Draw link costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    name = sprintf('Link costs. Average travel cost = %3.6g.', avg_cost);
    nw_drawer(nodes,links,link_costs,name,0,0);


elseif choice == 7
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Overview of routes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    for pq = 1:l_d

        max_flow  = max(route_flows{pq});
        cost_data = 1./route_costs{pq} * max_flow * min(route_costs{pq});

        clf;
        hold on;
        axis([0 num_routes(pq)+1 0 max_flow*1.1]);
        [trash, idx] = sort(-route_flows{pq});
```

```
        i = 1;
        for r = idx'
            fill([i-0.5 i-0.5 i+0.5 i+0.5], [0 cost_data(r) cost_data(r) 0], 'g');
            fill([i-0.2 i-0.2 i+0.2 i+0.2], [0 route_flows{pq}(r) route_flows{pq}(r) 0], 'r');
            i = i + 1;
        end
        title(sprintf('Routes for OD-pair %d -> %d.   Click mouse to continue or hit key to exit...',...
            demands(pq,1), demands(pq,2)));
        xlabel ('Red bars show route flows. Green bars show (route costs)^-^1.');
        if waitforbuttonpress
            break;
        end
    end


elseif choice == 8
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Draw OD-pair routes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Iterate submenus... lots of code here just to get the menus working intelligently
    first_od = 1;
    if l_d > max_menu_size - 1
        last_od     = max_menu_size - 2;
        back_choice = last_od + 2;
        more_choice = last_od + 1;
    else
        last_od     = l_d;
        back_choice = l_d + 1;
        more_choice = 0;
    end

    while 1
        text = 'choice2 = first_od - 1 + menu(''Please choose an OD-pair:''';
        for pq = first_od:last_od
            text = strcat(text, sprintf(', ''OD-pair %d -> %d''', demands(pq,1), demands(pq,2)));
        end
        if (more_choice > 0)
            text = strcat(text, ', ''More''');
        end
        text = strcat(text, ', ''Back'');');
        eval(text);
        if choice2 == back_choice
            break;
        elseif choice2 == more_choice
            first_od = first_od + max_menu_size-2;
            if last_od + max_menu_size - 1 >= l_d
                last_od = l_d;
                back_choice = l_d + 1;
                more_choice = 0;
            else
                last_od  = last_od  + max_menu_size-2;
                back_choice = last_od + 2;
                more_choice = last_od + 1;
            end
        else
            pq = choice2;
            % Display routes
            [trash, idx] = sort(-route_flows{pq});
            for r = idx'
                flow_string = sprintf('flow: %3.6g', route_flows{pq}(r));
                name = sprintf('Route %d (%s)    for OD-pair %d -> %d.    Route cost = %3.6g.',...
                    r, flow_string, demands(pq,1), demands(pq,2), route_costs{pq}(r));
                name = strcat(name, '    (Click mouse to continue or hit key to exit...)');
                route_vector = zeros(l_l,1);
                route_vector(routes{pq}{r}) = 1;
                nw_drawer(nodes,links,route_vector,name,demands(pq,1),demands(pq,2));
                if waitforbuttonpress
                    break;
                end
            end
        end
    end


elseif choice == 9
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Draw link flow derivatives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    nw_drawer(nodes,links,d_link_flows,'Directional derivatives of link flows.', 0, 0);


elseif choice ==  10
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Draw OD-pair link flow derivatives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    % Iterate submenus... lots of code here just to get the menus working intelligently
    first_od = 1;
    if l_d > max_menu_size - 1
        last_od     = max_menu_size - 2;
        back_choice = last_od + 2;
        more_choice = last_od + 1;
    else
        last_od     = l_d;
        back_choice = l_d + 1;
        more_choice = 0;
    end

    while 1
        text = 'choice2 = first_od - 1 + menu(''Please choose an OD-pair:''';
        for pq = first_od:last_od
            text = strcat(text, sprintf(', ''OD-pair %d -> %d''', demands(pq,1), demands(pq,2)));
        end
        if (more_choice > 0)
            text = strcat(text, ', ''More''');
        end
        text = strcat(text, ', ''Back'');');
        eval(text);
        if choice2 == back_choice
            break;
        elseif choice2 == more_choice
            first_od = first_od + max_menu_size-2;
            if last_od + max_menu_size - 1 >= l_d
                last_od = l_d;
                back_choice = l_d + 1;
                more_choice = 0;
            else
                last_od  = last_od  + max_menu_size-2;
                back_choice = last_od + 2;
                more_choice = last_od + 1;
            end
        else
            pq = choice2;
            % Draw diagram
            name = sprintf('Link flow derivatives for OD-pair %d -> %d.   OD-pair cost/derivative: %3.6g / %3.6g.',...
                demands(pq,1), demands(pq,2), odpair_costs(pq), d_odpair_costs(pq));
            nw_drawer(nodes,links,d_odpair_link_flows(:,pq),name,demands(pq,1),demands(pq,2));
        end
    end


elseif choice ==  11
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% choice: Show OD-pair route flow derivatives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Iterate submenus... lots of code here just to get the menus working intelligently
    first_od = 1;
    if l_d > max_menu_size - 1
        last_od     = max_menu_size - 2;
        back_choice = last_od + 2;
        more_choice = last_od + 1;
    else
        last_od     = l_d;
        back_choice = l_d + 1;
        more_choice = 0;
    end

    while 1
        text = 'choice2 = first_od - 1 + menu(''Please choose an OD-pair:''';
        for pq = first_od:last_od
            text = strcat(text, sprintf(', ''OD-pair %d -> %d''', demands(pq,1), demands(pq,2)));
        end
        if (more_choice > 0)
            text = strcat(text, ', ''More''');
        end
        text = strcat(text, ', ''Back'');');
        eval(text);
        if choice2 == back_choice
            break;
        elseif choice2 == more_choice
            first_od = first_od + max_menu_size-2;
            if last_od + max_menu_size - 1 >= l_d
                last_od = l_d;
                back_choice = l_d + 1;
                more_choice = 0;
            else
                last_od  = last_od  + max_menu_size-2;
                back_choice = last_od + 2;
                more_choice = last_od + 1;
            end
        else
            pq = choice2;
            % Display routes
```

```
                for r = 1:free_routes(pq)
                    if d_route_flows{pq}(r) ~= 0
                        name = sprintf('flow derivative: %3.6g. (Free route.)', d_route_flows{pq}(r));
                        name = strcat(name, '     (Click mouse to continue or hit key to exit...)');
                        route_vector = zeros(l_l,1);
                        route_vector(routes2{pq}{r}) = sign(d_route_flows{pq}(r));
                        nw_drawer(nodes,links,route_vector,name,demands(pq,1),demands(pq,2));
                        if waitforbuttonpress
                            break;
                        end
                    end
                end
                for r = free_routes(pq)+1:total_routes(pq)
                    if d_route_flows{pq}(r) ~= 0
                        name = sprintf('flow derivative: %3.6g. (Restricted route.)', d_route_flows{pq}(r));
                        name = strcat(name, '     (Click mouse to continue or hit key to exit...)');
                        route_vector = zeros(l_l,1);
                        route_vector(routes2{pq}{r}) = sign(d_route_flows{pq}(r));
                        nw_drawer(nodes,links,route_vector,name,demands(pq,1),demands(pq,2));
                        if waitforbuttonpress
                            break;
                        end
                    end
                end
            end
        end

    end
end
```

A.2 DSD algorithm TAP solver

```
function [routes, num_routes, route_flows, route_costs, link_flows, link_costs]...
    = nw_dsd_solver_bpr(links, demands, l_n)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function for solving a traffic assignment problem
%      using a disaggregated simplicial decomposition method.
%
% * Input arguments:
%
%      'links'    : a matrix w/ one row for each link;
%                   columns 1/2 = start/end nodes;
%                   columns 3/4/5/6 = A_a / B_a / K_a / n,
%                   where the link cost t_a(f_a) = A_a + B_a * (f_a / K_a)^n
%      'demands'  : a matrix with one row for each OD-pair;
%                   columns 1/2 = origin/destination nodes; column 3 = demands.
%      'l_n'      : the number of nodes in the network.
%
% * Output arguments:
%
%      'routes'         : routes generated when solving the problem. a cell array of cell arrays
%                         of vectors. first cell index = od-pair. second cell index = route number.
%                         vector holds the link numbers of all links on the route.
%      'num_routes'     : a vector with the number of routes (in 'routes') for each od-pair.
%      'route_flows'    : a cell array of vectors, holding the equilibrium flow on each route.
%                         cell index = od-pair. vector index = route number.
%      'route_costs'    : a cell array of vectors, holding the equilibrium cost of each route.
%                         cell index = od-pair. vector index = route number.
%      'link_flows'     : a vector with the equilibrium flow for each link.
%      'link_costs'     : a vector with the equilibrium cost for each link.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% These are declared global so they can be modified "in place" by subfunctions (saves memory).
global routes;
global route_costs;
global route_flows;
global link_costs;
global link_flows;
global num_routes;

%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%

tol_values = [1e-1 1e-2 1e-3 1e-4 1e-6 1e-8]; % sequence of tolerance settings to use

% a failure means not reaching the set tolerance level AND also not being able to add any new routes
max_failures             = 25;   % number of consecutive failures before giving up and terminating.
fail_iter                = 50;   % number of rmp iterations without reaching tolerance before trying to add routes again
```

```
max_total_rmp_iterations = 5000; % total bound on the number of rmp's solved before giving up and terminating.

line_search_flow_tol     = 1e-12*mean(demands(:,3));  % maximum link flow error from a line search

log_to_screen            = 1;  % if 1, a lot of information is displayed during execution

use_newton               = 1;  % use projected Newton instead of proj. grad. point algorithm to generate search directions
%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%
% Initialization     %
%%%%%%%%%%%%%%%%%%%%%%

l_l = size(links,1);    % number of links
l_d = size(demands,1);  % number of od-pairs

routes      = cell(l_d, 1);
route_flows = cell(l_d, 1);
route_costs = cell(l_d, 1);
num_routes  = zeros(l_d, 1);
link_flows  = zeros(l_l, 1);
link_costs  = links(:,3);     % initialize costs to zero congestion costs

% Build node-node incidence matrix A.
% Aij = 0 if there's no link from i to j; otherwise Aij = index of the cheapest such link
A     = sparse(l_n, l_n);
for a = 1:l_l
    node1 = links(a,1);
    node2 = links(a,2);
    if (A(node1, node2) == 0) | (link_costs(a) < link_costs(A(node1, node2)))
        A(node1, node2) = a;
    end
end
%%%%%%%%%%%%%%%%%%%%%%


% Add shortest routes to 'routes'. (modifies 'routes', 'num_routes', 'route_flows', 'route_costs')
add_new_shortest_routes(A, link_costs, demands(:,1:2));

% Put all od_pair flow on shortest route for each od-pair
for pq = 1:l_d
    route_flows{pq}(1) = demands(pq,3);
    link_flows(routes{pq}{1}) = link_flows(routes{pq}{1}) + demands(pq,3);
end

% Calculate link costs
link_costs = calc_link_costs(link_flows, links(:,3:end));

% Update route costs
for pq = 1:l_d
    route_costs{pq}(1) = sum(link_costs(routes{pq}{1}));
end

% Update node-node incidence matrix A.
for a = 1:l_l
    node1 = links(a,1);
    node2 = links(a,2);
    if (link_costs(a) < link_costs(A(node1, node2)))
        A(node1, node2) = a;
    end
end

% Add new shortest routes to 'routes'. (modifies 'routes', 'num_routes', 'route_flows', 'route_costs')
n = add_new_shortest_routes(A, link_costs, demands(:,1:2));

if n == 0
    if log_to_screen
        disp(sprintf('Algorithm terminated with %d route(s), tolerance 0.', l_d));
    end
    break;
end

failures = 0;
tol_idx = 1;
rmp_tol = tol_values(tol_idx);
if log_to_screen
    disp(sprintf('Algorithm started with %d routes. Tolerance set to %0.3g', n+l_d, rmp_tol));
end

for i = 1:max_total_rmp_iterations

    % Solve restricted master problem. (modifies 'route_flows', 'link_flows', 'route_costs', 'link_costs')
    [tol_was_reached, tol_reached, iter] = solve_rmp(routes, links(:,3:end), num_routes, line_search_flow_tol, rmp_tol, fail_iter, use_newton);

    if (tol_was_reached)
        failures = 0;

        % Update node-node incidence matrix A.
```

```
        for a = 1:l_l
            node1 = links(a,1);
            node2 = links(a,2);
            if (link_costs(a) < link_costs(A(node1, node2)))
                A(node1, node2) = a;
            end
        end

        % Add currently shortest routes to 'routes'. (modifies 'routes', 'num_routes', 'route_flows', 'route_costs')
        n = add_new_shortest_routes(A, link_costs, demands(:,1:2));

        if n == 0
            if tol_reached <= tol_values(end)
                if log_to_screen
                    disp(sprintf('%5d iteration(s)... final tolerance (%0.3g) reached.  no routes added => algorithm terminated.', iter, tol_values(end
                end
                break;
            else
               while tol_reached <= rmp_tol
                  tol_idx = tol_idx + 1;
                  rmp_tol = tol_values(tol_idx);
               end
               if log_to_screen
                   disp(sprintf('%5d iteration(s)... tolerance reached.  no routes added => tolerance set to %0.3g', iter, rmp_tol));
               end
            end
        elseif log_to_screen
            disp(sprintf('%5d iteration(s)... tolerance reached. %3d route(s) added', iter, n));
        end

    else  % failed to reach requested tolerance within the set number of iterations => try to add new routes before continuing

        % Update node-node incidence matrix A.
        for a = 1:l_l
            node1 = links(a,1);
            node2 = links(a,2);
            if (link_costs(a) < link_costs(A(node1, node2)))
                A(node1, node2) = a;
            end
        end

        % Add currently shortest routes to 'routes'. (modifies 'routes', 'num_routes', 'route_flows', 'route_costs')
        n = add_new_shortest_routes(A, link_costs, demands(:,1:2));

        if n > 0
            if log_to_screen disp(sprintf('%5d iteration(s)... %d route(s) added', iter, n)); end
            failures = 0;
        else
            failures = failures + 1;
            if log_to_screen disp(sprintf('%5d iteration(s)... no routes added (%d)', iter, failures)); end
            if failures == max_failures
                if log_to_screen
                    disp(sprintf('%5d iterations... algorithm terminated. Required tolerance NOT reached.', iter));
                end
                break;
            end
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [tol_was_reached, tol_reached, iter] = solve_rmp(routes, link_cost_data, num_routes, ls_flow_tol, rmp_tol, max_iter, use_newton)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function solves the restricted master problem over the routes in 'routes'.
% In each iteration, it uses a projected gradient point method (or a projected Newton method,
% if the use_newton flag is set) to generate a search direction and then performs a line search.
%
% * Input arguments:
%
%      'routes'         : the routes over which the problem is solved. a cell array of cell arrays
%                         of vectors. first cell index = od-pair. second cell index = route number.
%                         vector holds link numbers of all links on the route.
%      'link_cost_data' : a matrix with a row for each od-pair. columns contain cost data
%                         on the form specified in the main function header of this file.
%      'num_routes'     : a vector with the number of routes for each od-pair.
%      'ls_flow_tol'    : the max abs link flow error allowed in line searches.
%      'rmp_tol'        : the tolerance to achieve for the RMP before terminating.
%      'max_iter'       : the maximum number of iterations used in solving the RMP.
%      'use_newton'     : if 1, this function uses a proj newton algorithm to generate a search
%                         direction. Standard algorithm is gradient point projection.
%      'route_flows'    : a cell array of vectors, holding the flow on each route.
%                         cell index = od-pair. vector index = route number.
%      'route_costs'    : a cell array of vectors, holding the cost of each route.
%                         cell index = od-pair. vector index = route number.
%      'link_flows'     : a vector with the flow on each link.
```

```
%      'link_costs'     : a vector with the cost for each link.
%
% * Output arguments:
%
%      'tol_was_reached' : 1 if the required tolerance was reach, 0 otherwise
%      'tol_reached'     : the value of the termination coefficient achieved
%      'iter'            : number of iteratons used
%      'route_flows', 'route_costs', 'link_flows' & 'link_costs' have been updated.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

l_l = size(link_cost_data, 1); % number of links
l_d = length(route_flows);     % number of od-pairs

tol_was_reached = 0;
for i = 1:max_iter

    % Generate search direction
    if use_newton
        d_link_costs  = calc_d_link_costs(link_flows, link_cost_data);
        search_direction = generate_search_direction2(routes, num_routes, route_flows, route_costs, d_link_costs);
    else
        search_direction = generate_search_direction(num_routes, route_flows, route_costs);
    end

    % Determine upper bound on step length
    ub_on_step = inf;
    for pq = 1:l_d
        for r = 1:num_routes(pq)
            if search_direction{pq}(r) < 0
                ub_for_this_route = - route_flows{pq}(r) / search_direction{pq}(r);
                if ub_for_this_route < ub_on_step
                    ub_on_step = ub_for_this_route;
                end
            end
        end
    end

    % Calculate search direction in total link flow space
    link_search_direction = zeros(l_l,1);
    for pq = 1:l_d
         for r = 1:num_routes(pq)
            link_search_direction(routes{pq}{r}) = link_search_direction(routes{pq}{r}) + search_direction{pq}(r);
        end
    end

    % Terminate if at optimum before 1st iteration (needed since check of termination criterion is at end)
    if max(abs(link_search_direction))==0
        tol_reached = 0;
        tol_was_reached = 1;
        break;
    end

    % Calculate optimal step length
    if exist('ls_step') == 1
        step_guess = min(ls_step, ub_on_step);
    else
        step_guess = ub_on_step;
    end
    ls_step = line_search(link_flows, link_cost_data, link_search_direction, ub_on_step, step_guess, ls_flow_tol);

    % Calculate new flows and costs
    link_flows = link_flows + ls_step * link_search_direction;
    link_flows = max(0, link_flows);
    link_costs = calc_link_costs(link_flows, link_cost_data);

    for pq = 1:l_d
        route_flows{pq} = route_flows{pq} + ls_step * search_direction{pq};
        for r = 1:num_routes(pq)
            route_costs{pq}(r) = sum(link_costs(routes{pq}{r}));
        end
    end

    % Calculate termination coefficient
    min_route_costs = zeros(l_d,1);
    surplus_cost    = 0;
    for pq = 1:l_d
        min_route_costs(pq) = min(route_costs{pq});
        surplus_cost        = surplus_cost + route_flows{pq}' * (route_costs{pq} - min_route_costs(pq));
    end

    tol_reached = surplus_cost / (link_flows' * link_costs);

    % Check termination criterion
    if tol_reached < rmp_tol
```

```
        tol_was_reached = 1;
        break;
    end
end

iter = i;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function search_direction = generate_search_direction(num_routes, route_flows, route_costs);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function generates a search direction in route flows. It calculates a point on
% the projected gradient to determine the search direction.
%
% * Input arguments:
%
%      'num_routes'     : a vector with the number of routes for each od-pair.
%      'route_flows'    : a cell array of vectors, holding the flow on each route.
%                         cell index = od-pair. vector index = route number.
%      'route_costs'    : a cell array of vectors, holding the cost of each route.
%
% * Output arguments:
%
%      'search_direction' : a cell array of vectors holding the search direction.
%                           cell index = od-pair. vector index = route number.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k = 1;
l_d = size(num_routes,1);
search_direction = cell(l_d,1);

for pq = 1:l_d
        grad_direction = -k * route_costs{pq};

        % break_points(r) = max value for Lagrange multiplier for route r to be used.
        break_points = route_flows{pq} + grad_direction;
        [sorted_bp] = sort(break_points);

        % try if we should use all routes for this od-pair
        d_flows = -sorted_bp(1) + grad_direction;
        if sum(d_flows) <= 0
            j = 1;
        else
            % now we must search to determine how many routes should be used
            % j = first used route, i.e. the j-1 most expensive routes are not used
            ub = num_routes(pq);
            lb = 2;
            while ub~=lb
                j = floor((ub + lb) / 2);
                d_flows = max(-route_flows{pq}, -sorted_bp(j) + grad_direction);
                if sum(d_flows) > 0
                    lb = j+1;
                else
                    ub = j;
                end
            end
            j = ub;
            d_flows = max(-route_flows{pq}, -sorted_bp(j) + grad_direction);
        end

    multiplier = sorted_bp(j) + sum(d_flows) ./ (num_routes(pq)-j+1);
    search_direction{pq} = max(-route_flows{pq}, -multiplier  + grad_direction);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function search_direction = generate_search_direction2(routes, num_routes, route_flows, route_costs, d_link_costs);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function generates a search direction in route flows. It uses a projected Newton method
%    to determine the search direction.
%
% * Input arguments:
%
%      'routes'         : the routes over which the current rmp is being solved. a cell array
%                         of cell arrays of vectors. first cell index = od-pair.
%                         second cell index = route number.
%                         vector holds link numbers of all links on the route.
%      'num_routes'     : a vector with the number of routes for each od-pair.
%      'route_flows'    : a cell array of vectors, holding the flow on each route.
%                         cell index = od-pair. vector index = route number.
```

```
%      'route_costs'    : a cell array of vectors, holding the cost of each route.
%      'd_link_costs'   : a vector with the current link cost derivatives w.r.t. link flows
%
% * Output arguments:
%
%      'search_direction' : a cell array of vectors holding the search direction.
%                           cell index = od-pair. vector index = route number.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k = 1;  % constant used to determine weighting of second-order information

l_d = size(num_routes,1);  % number of OD-pairs
search_direction = cell(l_d,1);
for pq = 1:l_d
    if num_routes(pq) == 1
        search_direction{pq} = 0;  % if just one route, no calculations necessary
    else

        % Calculate route cost derivatives
        d_route_costs = zeros(num_routes(pq),1);
        for r = 1:num_routes(pq)
            d_route_costs(r) = sum(k .* d_link_costs(routes{pq}{r}));
        end

        % break_points(r) = max value for Lagrange multiplier for route r to be used.
        break_points = d_route_costs .* route_flows{pq} - route_costs{pq};
        [sorted_bp, idxs] = sort(break_points);

        finished = 0;

        % If some route(s) have route cost derivatives of zero, these routes must be
        % either unused or have exactly route_cost = - the value of the Lagrange mult.
        zers_idxs = find(d_route_costs(idxs) == 0);
        if ~isempty(zers_idxs)

            % try multiplier value of the break point of route zers_idxs(end), else put lb = zers_idxs(1)+1
            d_route_costs(idxs(zers_idxs)) = eps; % to prevent division by zero
            d_flows = max(-route_flows{pq}, -(sorted_bp(zers_idxs(end)) + route_costs{pq}) ./ d_route_costs);
            d_flows(idxs(zers_idxs)) = -route_flows{pq}(idxs(zers_idxs));

            if sum(d_flows) <= 0  % this means there should be some flow on the link at zers_idxs(end)
                d_flows(idxs(zers_idxs(end))) = d_flows(idxs(zers_idxs(end))) - sum(d_flows);
                search_direction{pq} = d_flows;
                multiplier = sorted_bp(zers_idxs(end));
                finished = 1;
            else  % no zero-route cost derivative links should be used. We must still search for the multiplier value.
                lb = zers_idxs(end)+1;
                ub = num_routes(pq);
            end
        else  % this is just like the gradient point projection case (see function above)
            ub = num_routes(pq);
            lb = 2;
            d_flows = max(-route_flows{pq}, -(sorted_bp(1) + route_costs{pq}) ./ d_route_costs);
        end
        if ~finished
            if sum(d_flows) <= 0
                j = 1;
            else

                % now we must search to determine how many routes should be used
                % j = first used route, i.e. the j-1 most expensive routes are not used
                while ub ~= lb
                    j = floor((ub + lb) / 2);
                    d_flows = max(-route_flows{pq}, -(sorted_bp(j) + route_costs{pq}) ./ d_route_costs);
                    if sum(d_flows) > 0
                        lb = j+1;
                    else
                        ub = j;
                    end
                end
                j = ub;
                d_flows = max(-route_flows{pq}, -(sorted_bp(j) + route_costs{pq}) ./ d_route_costs);
            end

            multiplier = sorted_bp(j) + sum(d_flows) ./ sum(1./d_route_costs(idxs(j:num_routes(pq))));
            search_direction{pq} = max(-route_flows{pq}, -(multiplier + route_costs{pq}) ./ d_route_costs);

        end
        % A rough feasibility filter, probably never used.
        if (sum(search_direction{pq}) > 10^-8) | (sum(search_direction{pq}) < -10^-8)
            [v,q] = max(search_direction{pq});
            search_direction{pq}(q) = search_direction{pq}(q) - sum(search_direction{pq});
        end
    end
```

```
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function ls_step = line_search(link_flows, link_cost_data, link_sd, ub, guess, flow_error_tol)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates the optimal step length in the direction 'search_direction'.
%
% * Input arguments:
%
%      'link_flows'     : a vector with the current flow for each link.
%      'link_cost_data' : a matrix with a row for each od-pair. columns contain cost data
%                         on polynomial form for the od-pairs. Column 1 = 0-degree coeff. etc.
%      'link_sd'        : link search direction: a vector with a value for each link defining
%                         the direction of the line search
%      'ub'             : the upper bound on the step length
%      'guess'          : a starting guess for the step length
%      'flow_error_tol' : the maximum abs link flow error in the step allowed for termination.
%
% * Output arguments:
%
%      'ls_step'        : the calculated step length
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

max_iter = 80;
lb = 0;
ls_step = guess;

% max abs error allowed in the step length
step_error_tol = flow_error_tol ./ max(abs(link_sd));

% first check if we should all way to the upper bound
if link_sd' * calc_link_costs(link_flows + ub * link_sd, link_cost_data) <= 0
    ls_step = ub;
    return;
end

% otherwise perform bisection search
i = 0;
while ((ub - lb) > step_error_tol) & (i < max_iter);
    i = i + 1;
    old_step = ls_step;
    if link_sd' * calc_link_costs(link_flows + ls_step * link_sd, link_cost_data) > 0
        ub = old_step;
        ls_step = (ub + lb) / 2;
    else
        lb = old_step;
        ls_step = (ub + lb) / 2;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function link_costs = calc_link_costs(link_flows, link_cost_data)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates link costs given the flow on each link.
%
% * Input arguments:
%
%      'link_flows'     : a vector with the current flow for each link.
%      'link_cost_data' : a matrix with a row for each link.
%                         columns 3/4/5/6 = A_a / B_a / K_a / n,
%                         where the link cost t_a(f_a) = A_a + B_a * (f_a / K_a)^n
%
% * Output arguments:
%
%      'link_costs'     : the costs of all links in the network
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

link_costs = link_cost_data(:,1) + link_cost_data(:,2) .* (link_flows ./ link_cost_data(:,3)).^link_cost_data(:,4);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function d_link_costs = calc_d_link_costs(link_flows, link_cost_data)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates link cost derivatives given the flow on each link.
%
% * Input arguments:
%
%      'link_flows'     : a vector with the current flow for each link.
%      'link_cost_data' : a matrix with a row for each od-pair. columns contain cost data
%                         on polynomial form for the od-pairs. Column 1 = 0-degree coeff. etc.
%
% * Output arguments:
%
%      'd_link_costs'   : the derivatives (w.r.t. link flow on the respective links) of the
%                         link costs of all links in the network
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

d_link_costs = link_cost_data(:,2) .* link_cost_data(:,4) .* link_cost_data(:,3) ...
    .* (link_flows ./ link_cost_data(:,3)).^(link_cost_data(:,4) - 1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function [num_new_routes] = add_new_shortest_routes(A, link_costs, od_pairs)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates the cheapest routes for all od-pairs and adds any of these (with zero
% flow) that are new (i.e. not in 'routes'). Also calculates route costs for the new routes.
%
% * Input arguments:
%
%      'A'              : a node-node incidence matrix. Aij = 0 if there is no link between node
%                         i and node j, otherwise Aij = link number of cheapest link from i to j.
%      'link_costs'     : a vector with the current cost for each link.
%      'od_pairs'       : a matrix with a row for each od-pair. Columns 1/2 = origin/dest. nodes.
%      'routes'         : the routes over which the problem is solved. a cell array of cell arrays
%                         of vectors. first cell index = od-pair. second cell index = route number.
%                         vector holds link numbers of all links on the route.
%      'num_routes'     : a vector with the number of routes for each od-pair.
%      'route_flows'    : a cell array of vectors, holding the flow on each route.
%                         cell index = od-pair. vector index = route number.
%      'route_costs'    : a cell array of vectors, holding the cost of each route.
%                         cell index = od-pair. vector index = route number.
%
% * Output arguments:
%
%      'num_new_routes' : a vectow with the number of routes added to 'routes'
%      'routes', 'num_routes', 'route_flows', & 'route_costs' have been updated.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global routes;
global num_routes;
global route_flows;
global route_costs;

l_d = size(od_pairs, 1);
l_n = size(A,1);

num_new_routes = 0;
current_origin = 0;


adjacent_node_list = cell(l_n);
for i = 1:l_n
    adjacent_node_list{i} = find(A(i,:)>0);
end


for pq = 1:l_d
    % generate a cheapest tree (unless we already have one) for the origin of od-pair pq
    % prev_list = list of predecessors for each node in the cheapest tree
    if od_pairs(pq,1) ~= current_origin % we don't have a prev_list yet
        current_origin = od_pairs(pq,1);
        prev_list      = nw_dijkstra(A, link_costs, current_origin, adjacent_node_list);
    end

    % calculate cheapest route for od-pair pq using cheapest tree
    current_dest = od_pairs(pq,2);
    new_route    = [];
    while current_dest ~= current_origin;
        temp         = prev_list(current_dest);
        new_route    = [A(temp, current_dest); new_route];
```

```
        current_dest = temp;
    end

    % determine if route is already listed in 'routes'
    is_old = 0;
    for r = 1:num_routes(pq)
        if isequal(routes{pq}{r}, new_route)
            is_old = 1;
            break;
        end
    end

    % if the route is new, add it to 'routes'
    if is_old == 0
        num_new_routes             = num_new_routes + 1;
        num_routes(pq)             = num_routes(pq) + 1;
        routes{pq}{num_routes(pq)} = new_route;
        route_flows{pq}            = [route_flows{pq}; 0];
        route_costs{pq}            = [route_costs{pq}; sum(link_costs(new_route))];
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function prev_list = nw_dijkstra(A, link_costs, origin, adjacent_node_list)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates the cheapest routes from one node (the origin) to all other nodes.
% It uses Dijkstras algorithm
%
% * Input arguments:
%
%      'A'              : a node-node incidence matrix. Aij = 0 if there is no link between node
%                         i and node j, otherwise Aij = link number of cheapest link from i to j.
%      'link_costs'     : a vector with the current cost for each link.
%      'origin'         : the origin node
%      'adjacent_node_list': node-node adjacency matrix
%
% * Output arguments:
%
%      'prev_list'      : a vector indexed by node. each element holds the number of the last link
%                         used to get to that particular node in the shortest route from the
%                         origin to that node.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


l_n       = size(A,1);            % number of nodes
labels    = zeros(l_n,1) + inf;   % costs of cheapest routes from origin to each node
fixed     = zeros(l_n,1);         % bool vector specifying which labels are fixed(certain)
prev_list = zeros(l_n,1);

labels(origin) = 0;

for fixed_labels = 1:(l_n-1)
    % choose a node to fix the label for
    not_fixed_idxs = find(fixed ~= 1);
    [trash, node_idx_idx] = min(labels(not_fixed_idxs));
    current_node = not_fixed_idxs(node_idx_idx);

    fixed(current_node) = 1;

    % update labels. adjacent_links(j) = idx of cheapest link from current node to node j, or 0
    current_label = labels(current_node);
    for node = adjacent_node_list{current_node}
        alt_cost = current_label + link_costs(A(current_node, node));
        if alt_cost < labels(node)
            labels(node)    = alt_cost;
            prev_list(node) = current_node;
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.3 Sensitivity analysis solver

```
function [d_link_flows, d_link_costs, d_route_flows, d_route_costs, routes2,...
         free_routes, total_routes, init_sum_route_cost_diff, sum_route_cost_diff]...
     = nw_sens_solver_bpr(routes, num_routes, route_costs, route_flows, link_costs, link_flows, d_rho, links, demands, l_n);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function for performing sensitivity analysis on a traffic equilibrium
%
% * Input arguments:
%
%      'links'      : a matrix w/ one row for each link;
%                     columns 1/2 = start/end nodes;
%                     columns 3/4/5/6 = A_a / B_a / K_a / n,
%                     where the link cost t_a(f_a) = A_a + B_a * (f_a / K_a)^n
%      'demands'     : a matrix with one row for each OD-pair;
%                     columns 1/2 = origin/destination nodes; column 3 = demands.
%      'l_n'         : the number of nodes in the network.
%      'routes'      : routes generated when originally solving the problem. a cell array of cell arrays
%                       of vectors. first cell index = od-pair. second cell index = route number.
%                       vector holds the link numbers of all links on the route.
%      'num_routes'  : a vector with the number of routes (in 'routes') for each od-pair.
%      'route_flows' : a cell array of vectors, holding the equilibrium flow on each route.
%                       cell index = od-pair. vector index = route number.
%      'route_costs' : a cell array of vectors, holding the equilibrium cost of each route.
%                       cell index = od-pair. vector index = route number.
%      'link_flows'  : a vector with the equilibrium flow for each link.
%      'link_costs'  : a vector with the equilibrium cost for each link.
%
% * Output arguments:
%      'd_link_flows', 'd_link_costs', 'd_route_flows', 'd_route_costs' : directional derivatives.
%      'routes2'     : all routes included in the sensitivity analysis (!= 'routes').
%      'free_routes' : the number of positive-eq-flow routes for each OD-pair.
%      'total_routes': the total number of routes for each OD-pair.
%      'init_sum_route_cost_diff', 'sum_route_cost_diff' : termination coefficient data
%
%
% Magnus Josefsson,  2003/02
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% These are declared global so they can be modified "in place" by subfunctions.
% note that these take the place of the original 'routes', 'link_flows' etc. in the dsd solver code.
% However, since these identifiers are used by inparameters here, we must rename them.
global routes2;
global num_routes2;
global route_costs2;
global route_flows2;
global link_costs2;
global link_flows2;


%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%
rmp_tol  = 1e-7;   % termination tolerance
max_iter = 5000;

epsi_flow_coeff = 1e-6;
epsi_cost_coeff = 1e-6;

log_to_screen = 0;
%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%
% Initialization     %
%%%%%%%%%%%%%%%%%%%%%%

l_l = size(links,1);
l_d = size(demands,1);

route_conversion_idx = cell(l_d, 1);
routes2        = cell(l_d, 1);
route_flows2  = cell(l_d, 1);
route_costs2  = cell(l_d, 1);
free_routes   = zeros(l_d, 1);
total_routes  = zeros(l_d, 1);
link_flows2   = zeros(l_l, 1);
link_costs2   = zeros(l_l, 1);

%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

link_cost_data2 = [calc_d_link_costs_d_rho(link_flows, links(:,3:end), d_rho), calc_d_link_costs(link_flows, links(:,3:end))];

% Build node-node incidence matrix A.
% Aij = 0 if there's no link from i to j; otherwise Aij = index of the cheapest such link
A     = sparse(l_n, l_n);
for a = 1:l_l
    node1 = links(a,1);
    node2 = links(a,2);
```

```
    if (A(node1, node2) == 0) | (link_costs(a) < link_costs(A(node1, node2)))
        A(node1, node2) = a;
    end
end

current_origin = 0;
for pq = 1:l_d
    if demands(pq, 1) ~= current_origin
        current_origin = demands(pq, 1);
        labels = dijkstra(A, link_costs, current_origin);
    end

    route_conversion_idx{pq} = find(route_flows{pq} > epsi_flow_coeff * demands(pq,3));
    for j = 1:length(route_conversion_idx{pq})
        routes2{pq}{end+1} = routes{pq}{route_conversion_idx{pq}(j)};
    end

    max_cost = min(route_costs{pq}) * (1 + epsi_cost_coeff);
    cheap_routes = find_cheap_routes(max_cost, labels, links, link_costs, demands(pq,1), demands(pq,2));

    free_routes(pq) = length(routes2{pq});
    for i = 1:length(cheap_routes)
        is_new_route = 1;
        for j = 1:free_routes(pq)
            if (length(cheap_routes{i}) ==  length(routes2{pq}{j})) & (cheap_routes{i} == routes2{pq}{j})
                is_new_route = 0;
                break;
            end
        end
        if is_new_route
            routes2{pq}{end+1} = cheap_routes{i};
        end
    end
    total_routes(pq) = length(routes2{pq});

    route_flows2{pq} = zeros(total_routes(pq),1);
end

link_costs2  = calc_link_costs(link_flows2, link_cost_data2);

for pq = 1:l_d
    route_costs2{pq} = [];
    for r = 1:total_routes(pq)
        route_costs2{pq} = [route_costs2{pq}; sum(link_costs2(routes2{pq}{r}))];
    end
end

if log_to_screen
    disp(sprintf('Algorithm started. %d routes (%d free). Tolerance: %0.3g', sum(total_routes), sum(free_routes), rmp_tol));
end

[tol_was_reached, init_sum_route_cost_diff, sum_route_cost_diff]...
    = solve_rmp(routes2, link_cost_data2, free_routes, total_routes, rmp_tol, max_iter);

if log_to_screen
    if tol_was_reached == 1
        disp('Algorithm terminated. Required tolerance reached.');
    elseif tol_was_reached == 0
        disp('Algorithm terminated. Required tolerance NOT reached.');
    elseif tol_was_reached == -1
        disp('Algorithm terminated. Required tolerance NOT reached (Algorithm stuck).');
    end
end

d_route_flows = route_flows2;
d_link_flows  = link_flows2;
d_route_costs = route_costs2;
d_link_costs  = link_costs2;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [tol_was_reached, init_sum_route_cost_diff, sum_route_cost_diff]...
    = solve_rmp(routes, link_cost_data, free_routes, total_routes, rmp_tol, max_iter)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function solves the restricted master problem over the routes in 'routes'.
%
% * Input arguments:
%
%      'routes'         : the routes over which the problem is solved. a cell array of cell arrays
%                         of vectors. first cell index = od-pair. second cell index = route number.
%                         vector holds link numbers of all links on the route.
%      'link_cost_data' : a matrix with a row for each od-pair. columns contain cost data
%                         on polynomial form for the od-pairs. Column 1 = 0-degree coeff. etc.
%      'free_routes'    : a vector with the number of free routes for each od-pair.
%      'total_routes'   : a vector with the total number of routes for each od-pair.
%      'rmp_tol'        : the tolerance coefficient for the RMP.
%      'max_iter'       : the maximum number of iterations before terminating.
```

```
%
% * Output arguments:
%
%      'route_flows2', 'route_costs2', 'link_flows2' & 'link_costs2' have been updated.
%      'tol_was-reached': specifies whether the requested tolerance was reached.
%      'init_sum_route_cost_diff', 'sum_route_cost_diff' : termination coefficient data.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global route_flows2;
global route_costs2;
global link_flows2;
global link_costs2;

l_l = size(link_costs2, 1);
l_d = length(route_flows2);

tol_was_reached = 0;

init_sum_route_cost_diff = 0;

for pq = 1:l_d
    min_route_costs          = min(route_costs2{pq});
    live_route_indecis       = [(1:free_routes(pq))];
    init_sum_route_cost_diff = init_sum_route_cost_diff + sum(route_costs2{pq}(live_route_indecis) - min_route_costs);
end

sum_route_cost_diff = init_sum_route_cost_diff;

for i = 1:max_iter
    search_direction = generate_search_direction(free_routes, total_routes, route_flows2, route_costs2);

    % Determine upper bound on step length
    ub_on_step = inf;
    for pq = 1:l_d
        for r = free_routes(pq)+1:total_routes(pq)
            if search_direction{pq}(r) < 0
                ub_for_this_route = - route_flows2{pq}(r) / search_direction{pq}(r);
                if ub_for_this_route < ub_on_step
                    ub_on_step = ub_for_this_route;
                end
            end
        end
    end

    % Calculate search direction in total link flow space
    link_search_direction = zeros(l_l,1);
    for pq = 1:l_d
        for r = 1:total_routes(pq)
            link_search_direction(routes{pq}{r}) = link_search_direction(routes{pq}{r}) + search_direction{pq}(r);
        end
    end

    if max(link_search_direction) == 0
        break;
    end

    % Calculate optimal step length
    if exist('ls_step') == 1
        step_guess = min(ls_step, ub_on_step);
    else
        step_guess = 1;
    end
    ls_step = line_search(link_flows2, link_cost_data, link_search_direction, ub_on_step, step_guess);

    % Calculate new flows and costs
    old_flow    = link_flows2;
    link_flows2 = link_flows2 + ls_step * link_search_direction;
    link_costs2 = calc_link_costs(link_flows2, link_cost_data);
    for pq = 1:l_d
        route_flows2{pq} = route_flows2{pq} + ls_step * search_direction{pq};
        for r = 1:total_routes(pq)
            route_costs2{pq}(r) = sum(link_costs2(routes{pq}{r}));
        end
    end

    sum_route_cost_diff = 0;

    for pq = 1:l_d
        min_route_costs     = min(route_costs2{pq});
        live_route_indecis  = [(1:free_routes(pq)), free_routes(pq) + find(route_flows2{pq}(free_routes(pq)+1:end) > 0)'];
        sum_route_cost_diff = sum_route_cost_diff + sum(route_costs2{pq}(live_route_indecis) - min_route_costs);
    end

    obj_fun_val = 0;
```

```
        for degree = 1:(size(link_cost_data,2))
            obj_fun_val = obj_fun_val + sum(link_cost_data(:,degree) .* link_flows2.^degree / degree);
        end

    %Check termination criterium
    if sum_route_cost_diff / init_sum_route_cost_diff < rmp_tol
        tol_was_reached = 1;
        break;
    end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function search_direction = generate_search_direction(free_routes, total_routes, route_flows, route_costs);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function generates a search direction in route flows. It uses a projected gradient
%    to determine the search direction.
%
% * Input arguments:
%
%      'free_routes'    : a vector with the number of free routes for each od-pair.
%      'total_routes'   : a vector with the total number of routes for each od-pair.
%      'route_flows'    : a cell array of vectors, holding the flow on each route.
%                         cell index = od-pair. vector index = route number.
%      'route_costs'    : a cell array of vectors, holding the cost of each route.
%
% * Output arguments:
%
%      'search_direction' : a cell array of vectors holding the search direction.
%                           cell index = od-pair. vector index = route number.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

l_d = length(total_routes);
search_direction = cell(l_d,1);

for pq = 1:l_d

    if total_routes(pq) == free_routes(pq)
        multiplier = -mean(route_costs{pq});
        search_direction{pq} = -multiplier - route_costs{pq};
    else

        break_points = route_flows{pq}(free_routes(pq)+1:end) - route_costs{pq}(free_routes(pq)+1:end);
        [sorted_bp, idx] = sort(break_points);

        if sum(-sorted_bp(1) - route_costs{pq}) <= 0
            multiplier = -mean(route_costs{pq});
        elseif sum(-sorted_bp(end) - route_costs{pq}(1:free_routes(pq))) - sum(route_flows{pq}(free_routes(pq)+1:end)) >= 0
            multiplier = -mean(route_costs{pq}(1:free_routes(pq))) - sum(route_flows{pq}(free_routes(pq)+1:end)) ./ free_routes(pq);
        else
            ub = total_routes(pq) - free_routes(pq);
            lb = 2;
            while ub~=lb
                j = floor((ub + lb) / 2);
                d_flows = -sorted_bp(j) - route_costs{pq};
                d_flows = [d_flows(1:free_routes(pq)); max(-route_flows{pq}(free_routes(pq)+1:end), d_flows(free_routes(pq)+1:end))];
                if sum(d_flows) > 0
                    lb = j+1;
                else
                    ub = j;
                end
            end
            j = ub;
            d_flows = -sorted_bp(j) - route_costs{pq};
            d_flows = [d_flows(1:free_routes(pq)); max(-route_flows{pq}(free_routes(pq)+1:end), d_flows(free_routes(pq)+1:end))];
            multiplier = sorted_bp(j) + sum(d_flows) ./ (total_routes(pq)-j+1);
        end
        search_direction{pq} = -multiplier - route_costs{pq};
        search_direction{pq}(free_routes(pq)+1:end) = max(-route_flows{pq}(free_routes(pq)+1:end), search_direction{pq}(free_routes(pq)+1:end));
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function ls_step = line_search(link_flows, link_cost_data, link_sd, ub, guess)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates the optimal step length in the direction 'search_direction'.
%
% * Input arguments:
%
%      'link_flows'     : a vector with the current flow for each link.
%      'link_cost_data' : a matrix with a row for each od-pair. columns contain cost data
```

```
%                          on polynomial form for the od-pairs. Column 1 = 0-degree coeff. etc.
%      'link_sd'        : link search direction: a vector with a value for each link defining
%                          the direction of the line search
%      'ub'             : the upper bound on the step length
%      'guess'          : a starting guess for the step length
%
% * Output arguments:
%
%      'ls_step'           : the calculated step length
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lb   = 0;
ls_step = guess;%ub/2;
max_iter = 80;
flow_error_tol = 0;

flow_error_tol = flow_error_tol ./ max(abs(link_sd));

if link_sd' * calc_link_costs(link_flows + ub * link_sd, link_cost_data) <= 0
    ls_step = ub;
    return;
end

i = 0;
while ((ub - lb) > flow_error_tol) & (i < max_iter);
    i = i + 1;
    old_step = ls_step;
    if link_sd' * calc_link_costs(link_flows + ls_step * link_sd, link_cost_data) > 0
        ub = old_step;
        ls_step = (ub + lb) / 2;
    else
        lb = old_step;
        ls_step = min((ub + lb) / 2, 2*lb);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function cheap_routes = find_cheap_routes(max_cost, labels, links, link_costs, origin, dest)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function returns all routes from origin to dest with cost less than or equal to max_cost.
%
% * Input arguments:
%
%      'labels'         : Dijksta algorithm labels.
%      'link_costs'     : a vector with the current cost for each link.
%      'origin'         : the origin node.
%      'dest'           : the destination node.
%
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cheap_routes = [];

for i = find(links(:,2) == dest)'
    if labels(links(i,1)) + link_costs(i) <= max_cost
        if links(i,1) == origin
            cheap_routes{end+1} = [i];
        else
            new_routes = find_cheap_routes(max_cost - link_costs(i), labels, links, link_costs, origin, links(i,1));
            for j = 1:length(new_routes)
                cheap_routes{end+1} = [new_routes{j}; i];
            end
        end
    end
end


function labels = dijkstra(A, link_costs, origin)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates the cheapest routes from one node (the origin) to all other nodes.
%
% * Input arguments:
%
%      'A'              : a node-node incidence matrix. Aij = 0 if there is no link between node
%                          i and node j, otherwise Aij = link number of cheapest link from i to j.
%      'link_costs'     : a vector with the current cost for each link.
%      'origin'         : the origin node
```

```
%
% * Output arguments:
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


l_n       = size(A,1);
labels    = zeros(l_n,1) + inf;
fixed     = zeros(l_n,1);
prev_list = zeros(l_n,1);

labels(origin) = 0;

for fixed_labels = 1:(l_n-1)
    % choose a node to fix the label for
    idx = find(fixed ~= 1);
    [trash, idx2] = min(labels(idx));
    current_node = idx(idx2);
    fixed(current_node) = 1;

    % update labels
    current_label = labels(current_node);
    current_links = A(current_node, :);
    idx3 = find(current_links > 0);

    for node = idx3 % iterate over all nodes with a link from current_node
        alt_cost = current_label + link_costs(current_links(node));
        if alt_cost < labels(node)
            labels(node)    = alt_cost;
            prev_list(node) = current_node;
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function link_costs = calc_link_costs(link_flows, link_cost_data)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates link costs given the flow on each link.
%
% * Input arguments:
%
%      'link_flows'     : a vector with the current flow for each link.
%      'link_cost_data' : a matrix with a row for each od-pair. columns contain cost data
%                         on polynomial form for the od-pairs. Column 1 = 0-degree coeff. etc.
%
% * Output arguments:
%
%      'link_costs'     : the costs of all links in the network
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

link_costs = link_cost_data(:,1);
for degree = 1:(size(link_cost_data, 2) - 1)
    link_costs = link_costs + link_cost_data(:,degree+1) .* link_flows.^degree;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function d_link_costs = calc_d_link_costs(link_flows, link_cost_data)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates link cost derivatives given the flow on each link.
%
% * Input arguments:
%
%      'link_flows'     : a vector with the current flow for each link.
%
% * Output arguments:
%
%      'd_link_costs'   : the derivatives (w.r.t. link flow on the respective links) of the
%                         link costs of all links in the network
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

d_link_costs = link_cost_data(:,2) .* link_cost_data(:,4) ./ link_cost_data(:,3) ...
    .* (link_flows ./ link_cost_data(:,3)).^(link_cost_data(:,4) - 1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function d_link_costs = calc_d_link_costs_d_rho(link_flows, link_cost_data, d_rho)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function calculates link cost derivatives w.r.t. parameters rho and multiplies by d_rho
%
% * Input arguments:
%
%
% * Output arguments:
%
%      'd_link_costs'   : the derivatives (w.r.t. parameters rho for the respective links) of the
%                         link costs of all links in the network multiplied by all d_rho
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

d_link_costs = zeros(length(link_flows),1);
for i = 1:length(link_flows)
    if d_rho(i) ~= 0
        d_link_costs(i) = d_rho(i) * -link_cost_data(i,2) .* link_cost_data(i,4) ./ link_cost_data(i,3)...
            .* (link_flows(i) ./ link_cost_data(i,3)).^(link_cost_data(i,4));
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.4 Frank-Wolfe/PARTAN TAP Solvers

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function for solving a traffic asssignment problem using the Frank-Wolfe algorithm.
%
% * Uses the function file 'nw_linear_tap'.
% * Input arguments:
%
%      'links'   : a matrix w/ one row for each link;
%                  columns 1/2 = start/end nodes; columns 3+ = cost data on polynomial form
%      'demands' : a matrix with one row for each OD-pair;
%                  columns 1/2 = origin/destination nodes; column 3 = demands.
%      'l_n'     : the number of nodes in the network.
%
% * Output arguments:
%
%      'flows'   : a matrix with a column for each OD-pair flow.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function flows = nw_fw_solver(links, demands, l_n)

%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%

max_iterations = 1000;
funtol= 1e-5;

%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
global times;
global Ts;

orig_time = cputime;
times = [];
Ts    = [];

l_l = length(links);
l_d = length(demands);

f    = nw_linear_tap (links(:,1:2), links(:,3), demands, l_n);
lbd  = 0;
step = 0.5;
i    = 0;

while i < max_iterations;
    i = i + 1;
    y   = nw_linear_tap (links(:,1:2), nw_costfun(f, links), demands, l_n);
    lbd = max(lbd, nw_objfun_simple(f, links) + (sum(y,2) - sum(f,2))' * nw_costfun(f, links));
    if (nw_objfun_simple(f, links) - lbd) / lbd < funtol
```

```
        break;
    end
    step = nw_line_search(f, y, links, step);
    f = f + step * (y - f);
    T = nw_objfun_simple(f, links);
    Ts = [Ts;T];
    times = [times;cputime-orig_time];
    error = (T - lbd) / lbd;
    if error < funtol
        break;
    end
end
iterations               = i
objective_function_value = nw_objfun_simple(f, links)
lower_bound              = lbd
maximal_error            = error

Aeq = zeros(l_n*l_d, l_l*l_d);
for i = 1:l_l
   for j = 1:l_d
      Aeq((j-1)*l_n + links(i,1), (j-1)*l_l + i) = -1;
      Aeq((j-1)*l_n + links(i,2), (j-1)*l_l + i) =  1;
  end
end

beq = zeros(l_n * l_d, 1);
for i = 1:l_d
   beq((i-1)*l_n + demands(i,1)) = -demands(i,3);
   beq((i-1)*l_n + demands(i,2)) = demands(i,3);
end

feasibility_error = max(Aeq*reshape(f, l_l*l_d,1) - beq)./max(demands(i,3))

flows = f;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function t = nw_objfun_simple(flows, links)

% sum flows for all commodities
x = sum(flows, 2);

% calculate objective function
t = 0;
for j = 3:size(links,2)
   t = t + sum(links(:,j) .* x.^(j-2) / (j-2));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function costs = nw_costfun(flows, links)

l_l = length(links);
l_d = size(flows,2);

% sum flows for all commodities
x = sum(flows, 2);

% calculate costs function
costs = zeros(l_l,1);
for j = 3:size(links,2)
   costs = costs + links(:,j) .* x.^(j-3);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function step = nw_line_search(f, y, links, last_step)

step = last_step;
ub       = 1;
lb       = 0;
tolvalue = step * 0.01;

while (ub - lb) > tolvalue
    if (sum(y,2) - sum(f,2))' * nw_costfun(step*y + (1-step)*f, links) > 0
        ub = step;
    else
        lb = step;
    end
    step = (ub + lb) / 2;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function for solving a traffic asssignment problem using the PARTAN method.
%
% * Uses the function file 'nw_linear_tap'.
% * Input arguments:
```

```
%
%      'links'   : a matrix w/ one row for each link;
%                  columns 1/2 = start/end nodes; columns 3+ = cost data on polynomial form
%      'demands' : a matrix with one row for each OD-pair;
%                  columns 1/2 = origin/destination nodes; column 3 = demands.
%      'l_n'     : the number of nodes in the network.
%
% * Output arguments:
%
%      'flows'   : a matrix with a column for each OD-pair flow.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function flows = nw_partan_solver(links, demands, l_n)

%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%

max_iterations = 1000;
funtol= 1e-9;

%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

l_l = size(links,1);
l_d = size(demands,1);

f     = nw_linear_tap (links(:,1:2), links(:,3), demands, l_n);
f_old = f;
lbd   = 0;
step1 = 0.5;
i     = 0;

while i < max_iterations
    i   = i + 1;
    y   = nw_linear_tap(links(:,1:2), nw_costfun(sum(f,2), links(:,3:end)), demands, l_n);
    p   = y - f;
    lbd = max(lbd, nw_objfun_simple(f, links) + (sum(p,2))' * nw_costfun(sum(f,2), links(:,3:end)));
    if (nw_objfun_simple(f, links) - lbd) / lbd < funtol
        break;
    end
    step1 = nw_line_search1(f, p, links, step1, 1e-7);
    f2    = f + step1 * p;
    p2    = f_old - f2;
    temp  = f;
    idx1  = (p2 > 0);
    idx2  = (p2 < 0);
    lb    = -min(f2(idx1)./p2(idx1));
    ub    =  min(-f2(idx2)./p2(idx2));
    step2 = nw_line_search2(f2, p2, links, 0, lb, ub, 1e-5);
    f     = f2 + step2 * p2;
    f_old = temp;
    error = (nw_objfun_simple(f, links) - lbd) / lbd;

    if error < funtol
        break;
    end
end


iterations               = i
objective_function_value = nw_objfun_simple(f, links)
lower_bound              = lbd
maximal_error            = error

Aeq = zeros(l_n*l_d, l_l*l_d);
for i = 1:l_l
   for j = 1:l_d
      Aeq((j-1)*l_n + links(i,1), (j-1)*l_l + i) = -1;
      Aeq((j-1)*l_n + links(i,2), (j-1)*l_l + i) =  1;
  end
end

beq = zeros(l_n * l_d, 1);
for i = 1:l_d
   beq((i-1)*l_n + demands(i,1)) = -demands(i,3);
   beq((i-1)*l_n + demands(i,2)) = demands(i,3);
end

feasibility_error = max(Aeq*reshape(f, l_l*l_d,1) - beq)./max(demands(i,3))

flows = f;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function t = nw_objfun_simple(flows, links)

% sum flows for all commodities
x = sum(flows, 2);

% calculate objective function
t = 0;
for j = 3:size(links,2)
   t = t + sum(links(:,j) .* x.^(j-2) / (j-2));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function costs = nw_costfun(flows, link_cost_data)

% calculate costs function
costs = link_cost_data(:,1);
for j = 2:size(link_cost_data, 2)
   costs = costs + link_cost_data(:,j) .* flows.^(j-1);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function step = nw_line_search1(f, p, links, step_guess, tolvalue)

ub   = 1;
lb   = 0;
step = step_guess;

f_agg    = sum(f,2);
p_agg    = sum(p,2);
p_agg_tr = p_agg';

while (ub - lb) > tolvalue
    if p_agg_tr * nw_costfun(f_agg + step*p_agg, links(:, 3:end)) > 0
        ub = step;
        step = 0.5 * ub + 0.5 * lb;
    else
        lb = step;
        step = min(2*step, 0.5*ub + 0.5*lb);
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function step = nw_line_search2(f, p, links, step_guess, lb, ub, tolvalue)

step = step_guess;

f_agg    = sum(f,2);
p_agg    = sum(p,2);
p_agg_tr = p_agg';

while (ub - lb) > tolvalue
    if p_agg_tr * nw_costfun(f_agg + step*p_agg, links(:, 3:end)) > 0
        ub = step;
        step = max(step-0.5,0.5*ub+0.5*lb);
        %step = (ub+lb)*0.5;
    else
        lb = step;
        step = min(step+0.5,0.5*ub+0.5*lb);
        %step = (ub+lb)*0.5;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function for solving a linear (constant costs) traffic asssignment problem.
%
% * Input arguments:
%
%       'links'   : a matrix with one row for each link in the network;
%                   columns 1/2 = start/end nodes
%       'costs'   : a vector with the cost for each link.
%       'demands' : a matrix with one row for each OD-pair (rows should be sorted by origin node);
%                   columns 1/2 = origin/destination nodes; column 3 = demands.
%       'l_n'     : the number of nodes in the network
%
% * Output arguments:
%
%       'flows'   : a matrix with a column for each OD-pair link flow vector.
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function flows = nw_linear_tap (links, costs, demands, l_n)

l_d       = size(demands,1);
```

```
l_l      = size(links,1);
flows    = zeros(l_l, l_d);
prev_list = zeros(l_n, 1);

% Construct node-node incidence matrix A.
% Aij = 0 if there's no link from i to j; otherwise Aij = index of the cheapest such link
A = sparse(l_n, l_n);
for i = 1:l_l
    j = links(i,1);
    k = links(i,2);
    if (A(j, k) == 0) | (costs(i) < costs(A(j, k)))
        A(j, k) = i;
    end
end


current_origin = 0;
for j = 1:l_d
    if demands(j,1) ~= current_origin % we don't have a prev_list yet
        current_origin = demands(j,1);
        prev_list = nw_dijkstra(A, costs, current_origin);
    end
    current_dest = demands(j,2);
    while current_dest ~= current_origin;
        temp              = prev_list(current_dest);
        addedlink         = A(temp, current_dest);
        flows(addedlink,j) = flows(addedlink,j) + demands(j,3);
        current_dest      = temp;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function prev_list = nw_dijkstra(A, costs, origin)

l_n      = size(A,1);
labels   = zeros(l_n,1) + inf;
fixed    = zeros(l_n,1);
prev_list = zeros(l_n,1);

labels(origin) = 0;

for j = 1:(l_n - 1)
    % choose a node to fix the label for
    idx = find(fixed ~= 1);
    [trash, idx2] = min(labels(idx));
    current_node = idx(idx2);
    fixed(current_node) = 1;

    % update labels
    current_label = labels(current_node);
    current_links = A(current_node, :);
    idx = find(current_links > 0);

    for i = idx
        alt_cost = current_label + costs(current_links(i));
        if alt_cost < labels(i)
           labels(i)    = alt_cost;
           prev_list(i) = current_node;
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.5 Network design solver

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nw_bilevel_bpr.m
%
% Tool for solving network design problems.
%
% FUNCTION DEPENDENCIES:
%   * Uses a solver function specified in 'solver_fun'
%   * Uses a sensitivity solver function spec. in 'sens_solver'
%
% INPUT
%   * Reads data from four text files:
%
%    'links_file'  : a text file w/ one row for each link;
%                    columns 1/2 = start/end nodes; columns 3+ = cost data on polynomial form
%                    all rows should have the same number of rows, i.e. add extra zeros if some
%                    links have lower degree cost polynomials than others.
%    'demands_file': a text file with one row for each OD-pair;
%                    columns 1/2 = origin/destination nodes; column 3 = demand.
%    'nodes_file'  : a text file with one row for each node;
```

```
%                    columns 1/2 = node coord's (x/y) in a planar representation of the network
%    'bilevel_file': a text file with one row for each link;
%                    columns 1/2 = lower and upper bound on capacity increases.
%                    columns 3+ = investment cost data on polynomial form
%
% OUTPUT
%   * Logs solution data to the file specified in 'sol_log_file'
%
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%

links_file    = 'braess2_lbpr.dat';
demands_file  = 'braess2_d.dat';
nodes_file    = 'braess2_r.dat';
bilevel_file  = 'braess2_b.dat';

sol_log_file  = 'nw_sol_bilevel.mat';

solver_fun    = 'nw_dsd_solver_bpr';
sens_solver   = 'nw_sens_solver_bpr';

armijo_factor = 0.01; % tau
armijo_multi  = 2;    % kappa
q_factor = 8;         % used to initialize step length

recalc_on = 0;        % recalculate T after step length is chosen

turns_to_freeze = 3;
turns_frozen    = 5;
turns_to_refreeze=1;

max_iterations = 2000;
%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

% Load network
links   = load(links_file);
nodes   = load(nodes_file);
demands = sortrows(load(demands_file), [1 2]);
bilevel_data = load(bilevel_file);
min_bounds   = bilevel_data(:,1);
max_bounds   = bilevel_data(:,2);

% active parameters
aps = find(max_bounds > min_bounds);

% investment cost data
build_costs  = bilevel_data(aps,3:end);

l_l     = size(links,1);
l_d     = size(demands,1);
l_n     = size(nodes,1);
l_ap    = size(aps,1);

obj_grad  = zeros(l_ap, 1);
bounded_b = zeros(l_ap, 1);
bounded_a = zeros(l_ap, 1);
frozen    = zeros(l_ap, 1);

iter = 0;

init_rho = zeros(l_ap,1);

rho = init_rho;

while 1

    mod_links = links;
    mod_links(aps,5) = links(aps,5) + rho;

    if (iter == 0) | (recalc_on)
        % Calculate solution
        temp_time = cputime;
        [routes, num_routes, route_flows, route_costs, link_flows, link_costs]...
            = eval(strcat(solver_fun, '(mod_links, demands, l_n)'));
    end

    % calculate objective function
    obj_fun = link_flows' * link_costs + nw_calc_build_costs(build_costs, rho);

    iter = iter + 1;
```

```
if iter > max_iterations
    break;
end

dsd_iters(iter+1) = 0;
sens_iters(iter+1) = 0;

% temporarily freeze/unfreeze inactive parameters
for i = 1:l_ap
    if (bounded_a(i) == turns_to_freeze) | (bounded_b(i) == turns_to_freeze)
        frozen(i) = turns_frozen;
        bounded_a(i) = turns_to_freeze - turns_to_refreeze;
        bounded_b(i) = turns_to_freeze - turns_to_refreeze;
    elseif frozen(i) > 0
        frozen(i) = frozen(i) - 1;
    end
end

% Calculate gradient
for i = 1:l_ap
    if frozen(i) > 0
        obj_grad(i) = 0;
    else
        d_rho = zeros(l_l,1);
        d_rho(aps(i)) = 1;
        temp_time = cputime;
        [d_link_flows, d_link_costs] = eval(strcat(sens_solver, '(routes, num_routes, route_costs, route_flows, link_costs',...
            ', link_flows, d_rho, mod_links, demands, l_n)'));
        sens_iters(iter+1) = sens_iters(iter+1) + 1;
        obj_grad(i) = d_link_costs' * link_flows + d_link_flows' * link_costs + nw_calc_d_build_costs(build_costs(i,:), rho(i));
    end
end

% make an initial projection of the gradient, so it is normalized properly for short steps
for i = 1:l_ap
    if frozen(i) > 0
        ;
    elseif (obj_grad(i) > 0) & (rho(i) <= min_bounds(aps(i)))
        bounded_b(i) = bounded_b(i) + 1;
        obj_grad(i) = 0;
    elseif (obj_grad(i) < 0) & (rho(i) >= max_bounds(aps(i)))
        bounded_a(i) = bounded_a(i) + 1;
        obj_grad(i) = 0;
    else
        bounded_b(i) = 0;
        bounded_a(i) = 0;
    end
end

% normalize objective gradient
obj_grad_norm = norm(obj_grad);
if obj_grad_norm == 0
    break;
end
norm_obj_grad = obj_grad ./ obj_grad_norm;

% calculate max steplength
max_step = 0;
for i = 1:l_ap
    if norm_obj_grad(i) > 0
        max_step = max(max_step, (rho(i) - min_bounds(aps(i))) ./ norm_obj_grad(i));
    elseif norm_obj_grad(i) < 0
        max_step = max(max_step, (rho(i) - max_bounds(aps(i))) ./ norm_obj_grad(i));
    end
end

if max_step == 0
    if max(frozen) == 0
        break;
    else
        frozen = zeros(l_ap,1);
        bounded_a = zeros(l_ap,1);
        bounded_b = zeros(l_ap,1);
    end
else

    % make sure step length is at most max_step
    if iter == 1
        step_length = max_step/q_factor;
    elseif step_length > max_step
        step_length = max_step/q_factor;
    end

    found_ok_point = 0;
    found_not_ok_point = 0;
    ok_point = 0;
    sub_iter = 0;
```

```
        % determine step length
        while (found_ok_point == 0) | (found_not_ok_point == 0)

            sub_iter = sub_iter + 1;

            % calculate rho resulting from current step length
            new_rho  = min(max_bounds(aps), max(min_bounds(aps), rho + step_length .* -norm_obj_grad));

            % calculate objective function value resulting from current step length
            mod_links(aps,5) = links(aps,5) + new_rho;
            temp_time = cputime;
            [routes, num_routes, route_flows, route_costs, link_flows, link_costs]...
                = eval(strcat(solver_fun, '(mod_links, demands, l_n)'));
            new_obj_fun = link_flows' * link_costs + nw_calc_build_costs(build_costs, new_rho);

            % perform armijo test
            if new_obj_fun + armijo_factor * step_length * obj_grad_norm <= obj_fun
                found_ok_point = 1;
                ok_point = new_rho;
                ok_link_flows = link_flows;
                ok_link_costs = link_costs;
                ok_routes = routes;
                ok_num_routes = num_routes;
                ok_route_costs = route_costs;
                ok_route_flows = route_flows;
                if found_not_ok_point == 0
                    if step_length < max_step
                        step_length = step_length * armijo_multi;
                    else
                        break;
                    end
                end
            else
                found_not_ok_point = 1;
                step_length = step_length / armijo_multi;
                if new_rho == rho % break if step is so short that it gives the same point as before
                    ok_point = new_rho;
                    ok_link_flows = link_flows;
                    ok_link_costs = link_costs;
                    ok_routes = routes;
                    ok_num_routes = num_routes;
                    ok_route_costs = route_costs;
                    ok_route_flows = route_flows;
                    break;
                end
            end
        end
        if rho == ok_point; % terminate if no step was taken
            if max(frozen) == 0
                break;
            else
                frozen = zeros(l_ap,1);
                bounded_a = zeros(l_ap,1);
                bounded_b = zeros(l_ap,1);
            end
        else
            rho = ok_point;
            link_flows = ok_link_flows;
            link_costs = ok_link_costs;
            routes = ok_routes;
            num_routes = ok_num_routes;
            route_costs = ok_route_costs;
            route_flows = ok_route_flows;
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function build_costs = nw_calc_build_costs(build_cost_data, param_value)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function
%
% * Input arguments:
%       'build_cost_data'   : the investment cost functions on polynomial form
%       'param_value'       : the current values of y
%
% * Output arguments:
%       'build_costs'       : the total investment cost
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

build_costs = 0;
for degree = 1:(size(build_cost_data, 2))
    build_costs = build_costs + sum(build_cost_data(:,degree) .* param_value.^degree);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function d_build_costs = nw_calc_d_build_costs(build_cost_data, param_value)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function
%
* Input arguments:
%       'build_cost_data'   : the investment cost functions on polynomial form
%       'param_value'       : the current values of y
%
% * Output arguments:
%       'd_build_costs'     : the total investment cost derivative
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

d_build_costs = build_cost_data(:,1);
for degree = 1:(size(build_cost_data, 2) - 1)
    d_build_costs = d_build_costs + build_cost_data(:,degree+1) .* param_value.^degree .* (degree+1);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B Network/solution illustrating tool

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function for illustrating solutions to traffic assignment problems.
%
%    Input arguments:
%      'nodes'   : a matrix w/ one row per node;
%                  columns 1/2 = node coord's (x/y) in a planar representation of the network.
%      'links'   : a matrix w/ one row per link;
%                  columns 1/2 = origin/destination nodes of the link.
%      'data'    : a vector with the data to be graphed for each link
%      'name'    : title of diagram
%      'node_o'  : 0 or specifies a node to be marked as origin
%      'node_d'  : 0 or specifies a node to be marked as destination
%
% Magnus Josefsson,  2002/11
% Chalmers University of Technology
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [] = nw_drawer(nodes, links, data, name, node_o, node_d)

%%%%%%%%%%%%%%%%%%%%%%
% Constants          %
%%%%%%%%%%%%%%%%%%%%%%

screen_format = 0.7168; % format (y/x) of the screen

% the following constants are specified relative to a window 1 unit high
margin        = 0.10;   % distance between the farthest out node and the edge of the window 0.25
node_radius   = 0.035;  % radius of a node 0.55
text_offset   = 0.012;  % offset from the node center that the node number text starts
maxdata_width = 0.03;   % width of the traffic rectangle for the link with the greatest 'data'-value
rect_offset   = 0.025;  % distance (along the link) between the node and the data rectangle
epsilon_width = 0.001;  % smallest width data rectangle that will be drawn
arrow_length  = 0.025;   % length of the two shorter 'arrow'-lines

theta         = 0.5;    % (radians) angle between a link and the two lines that form its arrow 0.3

link_width    = 1;      % (pixels) width of the lines drawn to represent links.
arrow_width   = 2;      % (pixels) width of the lines drawn to represent arrows. 3

data_repr_exp = 1;      % exponent relating the 'data'-value for a link to the width of the rectangle
                        % representing the data in the graph (1 => proportional)

neg_color     = [0.5,0.1,0.1]; % color for negative data values
pos_color     = [0.8,0.8,1.0]; % color for positive data values
node_color    = 'k';           % color for circle around node
```

```
node_o_color  = [0.9,0.7,0.7]; % color for origin node
node_d_color  = [0.7,0.9,0.7]; % color for destination node
node_bg_color = [0.8,0.8,0.8]; % color for regular node
link_color    = 'k';           % color for links

%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

clf;
hold on;

% make sure links are sorted
[trash, linksort_idxs] = sortrows(links, [1 2]);
trash = [];

x = nodes(:, 1);
y = nodes(:, 2);

% rescale x- & y-coordinates to fit the network on a window 1 unit high
x_range = (max(x)-min(x));
y_range = (max(y)-min(y));
if x_range > 0
    x = x / screen_format * (1 - 2*margin) / x_range;
end
if y_range > 0
    y = y * (1 - 2*margin) / y_range;
end

% display the window
axis([(min(x) - margin) (max(x) + margin) (min(y) - margin) (max(y) + margin)]);
axis equal;

% draw the nodes
t      = (0:0.1:6.3)';
circle = node_radius .* [cos(t), sin(t)];
for i = 1:length(x)
    if i == node_o
        fill(circle(:,1) + x(i), circle(:,2) + y(i), node_o_color);
        plot(circle(:,1) + x(i), circle(:,2) + y(i), node_color);
        text(x(i) - (2+(i>9))*text_offset, y(i), strcat('O:', int2str(i)));
    elseif i == node_d
        fill(circle(:,1) + x(i), circle(:,2) + y(i), node_d_color);
        plot(circle(:,1) + x(i), circle(:,2) + y(i), node_color);
        text(x(i) - (2+(i>9))*text_offset, y(i), strcat('D:', int2str(i)));
    else
        fill(circle(:,1) + x(i), circle(:,2) + y(i), node_bg_color);
        plot(circle(:,1) + x(i), circle(:,2) + y(i), node_color);
        text(x(i) - (1+(i>9))*text_offset, y(i), int2str(i));
    end
end



% draw the network and data
%  link_start(/end)_1 : coordinates of centers of start(/end) node
%  link_start(/end)_2 : coord's of intersections between link and node circles
%  link_start(/end)_3 : as above, but a distance further up(/down) the link
%  link_start(/end)_4 : as above, but also a distance to the right of the link
curr_link      = [0 0];
old_data_width = 0;
maxdata        = max(abs(data));

for i = 1:size(links,1)
    j = linksort_idxs(i);

    % calculate coordinates
    link_start     = [x(links(j,1)), y(links(j,1))];
    link_end       = [x(links(j,2)), y(links(j,2))];
    link_direction = (link_end - link_start) ./ norm(link_end - link_start);
    link_start2    = link_start  + link_direction .* node_radius;
    link_end2      = link_end    - link_direction .* node_radius;
    link_start3    = link_start2 + link_direction .* rect_offset;
    link_end3      = link_end2   - link_direction .* rect_offset;

    if maxdata ~= 0
        data_width     = maxdata_width * (abs(data(j)) / maxdata) .^ data_repr_exp;
    else
        data_width     = 0;
    end
    perp_direction = [link_direction(2), -link_direction(1)];
    link_start4    = link_start3 + data_width .* perp_direction;
    link_end4      = link_end3   + data_width .* perp_direction;

    if [links(j,1) links(j,2)] == curr_link
        link_start4 = link_start4 + old_data_width .* perp_direction;
        link_end4   = link_end4   + old_data_width .* perp_direction;
        link_start3 = link_start3 + old_data_width .* perp_direction;
        link_end3   = link_end3   + old_data_width .* perp_direction;
```

```
    end
    curr_link = [links(j,1) links(j,2)];
    old_data_width = data_width;

    % draw data
    if data(j) < 0
        data_color = neg_color;
    else
        data_color = pos_color;
    end
    if data_width > epsilon_width
        fill([link_start3(1); link_end3(1); link_end4(1); link_start4(1)],...
            [link_start3(2); link_end3(2); link_end4(2); link_start4(2)], data_color);
    end

    % draw links
    h = line([link_start2(1); link_end2(1)], [link_start2(2); link_end2(2)]);
    set(h, 'LineWidth', link_width, 'Color', link_color);

    % calculate & draw arrows
    arrow_direct1  = -link_direction * [cos(theta), -sin(theta); sin(theta), cos(theta)];
    arrow_direct2  = -link_direction * [cos(theta), sin(theta); -sin(theta), cos(theta)];
    arrow1         = arrow_length .* arrow_direct1;
    arrow2         = arrow_length .* arrow_direct2;
    h = line([link_end2(1); link_end2(1) + arrow1(1)], [link_end2(2); link_end2(2) + arrow1(2)]);
    set(h, 'LineWidth', arrow_width, 'Color', link_color);
    h = line([link_end2(1); link_end2(1) + arrow2(1)], [link_end2(2); link_end2(2) + arrow2(2)]);
    set(h, 'LineWidth', arrow_width, 'Color', link_color);
end

% Display diagram title
title(name);
```