CORE
Metadata, citation and similar papers at core.ac.uk
Provided by NORA - Norwegian Open Research Archives

System optimum vs. user equilibrium in static and dynamic traffic routing

by

Jon Marius Venstad

Thesis
for the degree of
Master of Mathematics

(Master i Matematikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

Juni 2007

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Contents

1 Introduction **4**

1.1 Transportation planning 4

1.2 This text 5

2 Background theory **7**

2.1 Graphs 7

2.1.1 Walks and paths 7

2.1.2 Trees 8

2.1.3 Distance, weighted graphs 9

2.1.4 Capacity, flows 10

2.2 Optimization 12

2.2.1 Convex sets, cones 13

2.2.2 Halfspaces and polyhedron 14

2.2.3 Linear Programming 16

2.2.4 The dual problem 17

2.2.5 Convex optimization 18

2.2.6 Complexity 18

3 The traffic routing problem **20**

3.1 Traffic model 20

3.2 Static system optimum 22

3.3 Static user equilibrium 24

3.4 Dynamic system optimum 25

3.5 Dynamic user equilibrium 28

4 Existing work and solution algorithms **29**

4.1 Distance and shortest paths 29

4.2 Weighted shortest path 31

4.3 The Simplex algorithm 34

4.3.1 Basic idea 34

4.3.2 Basic and non-basic variables 35

4.3.3 Correctness and complexity 38

4.4 Matrix notation, bases 38

4.5 Network flow 39

4.5.1 Multi-commodity flows 41

4.5.2 Maximum flows 42

5 Analysis 44
5.1 Special cases and simplifications 44
5.1.1 Number of commodities 44
5.1.2 Special latency functions 45
5.1.3 Simplified networks 47
5.1.4 Alternative optimality criteria 49
5.2 System optimal vs. user equilibrium 50
5.3 The dynamic case, simplified latency model 61
5.3.1 System optimal planning 61
5.3.2 Existence and uniqueness of the system optimal solution 62
5.3.3 The time discrete graph 63
5.3.4 System optimal planning by use of the augmented graph 64
5.3.5 Variable preferred arrival time 65
5.3.6 Properties of the augmented graph 66
5.3.7 Examples of the augmented graph 66
5.3.8 Multi-commodity planning 71
5.4 Chain decomposable flows 72
5.4.1 System optimal solution with chain flows 73

6 Summary 87

7 Appendix 89

1 Introduction

Optimization is a large discipline of mathematics, and loosely said concerns finding the best solution to some given problem. Numerous branches exist within the field of optimization, such as linear optimization, convex optimization, and integer optimization. Typically an optimization problem has a function to minimize or maximize over a given domain, and the problem might be easy or hard to solve, depending on both the objective function and the feasible domain. Some problems might be solved in a time polynomially proportional to a measure of the size of the problem, while other problems have not yet been, or can never be, solved faster than exponentially proportional to the size of the problem. When solving a problem by hand any big problem can become almost impossible to solve because of its sheer size, but with the aid of fast computers mathematicians today can solve bigger and bigger problems. In particular more and more real world problems can be solved to optimality with the emerging possibilities.

1.1 Transportation planning

One of the branches of real world problems that certainly benefits from optimization theory is transportation planning. In countless scenarios one or more kinds of commodities are to be transported between different locations, and it is often desirable to find the best way of doing this. What the defines the *best way* can be different from problem to problem, but some examples are:

- Routing traffic through a city, with as little congestion as possible.
- Routing internet traffic, with as little delay as possible.
- Routing containers between harbors, while transporting empty ones as short distances as possible.

Now in the case of routing traffic through a city, another possible criterion to consider could be minimizing the total travel time of all the commuters, and yet another could be minimizing the difference between desired and actual departure and arrival times for all commuters. And of course a combination of any of these criteria could be used. This is then the quantity we wish to minimize in our problem, and a function to compute this quantity is required for any mathematical optimization to be done. The mathematical model of our problem is what allows us to do

this calculation; it is a mathematical representation of our problem, and as such has a way of representing the traffic throughout the network in a precise and quantitative way. We can then use the mathematical model to check which different traffic routings are the best with regard to our choice of minimization goal. It is very important that the chosen mathematical model has properties that resemble those of the original problem. Often finding a good mathematical model is not very hard, but finding one that is not too complex for efficient optimization to take place might be harder. Typically this transportation kind of problem is regarded as a network problem where the network is represented as a graph, and each edge in the graph has a cost associated to it that depends on the amount of traffic flowing along it.

Another interesting viewpoint in traffic problems is that of each commuter, assuming the users of the network behave according to the egoistic goal of minimizing their own travel time in the network. This is by many considered the situation that will occur in a real world traffic network, and the "solution" we get from this approach can differ from the solution to the similar optimization problem of e.g. least travel time. Interestingly the user approach yields a solution that is often much worse in terms of total travel time. Bridging the gap between these two solutions to the traffic flow problem might, at least for the environmentalists, be of great interest.

1.2 This text

In this thesis we will have a look at the problem of optimally routing traffic through a network that does not have enough capacity for all the traffic to follow the same, fastest route. I will also try to compare the optimal solution to the user solution that is assumed to occur if no measures are taken to direct the traffic. In order to do this I will need a mathematical model for both problems, which might be slightly different from one another. The models used to represent the networks will in both cases be directed graphs with cost functions along each of the arcs. In addition I will need optimization theory to find the optimal solution to the given problems. In the simplest case we can use linear optimization, but might need other areas for the general case.

The outline of the thesis will be as follows: I will give the basic terminology of the text in section 2, and then go on to describe our problem and different varieties of it in section 3. Section 4 will be used to examine theoretical results that may be applied to our problems. In section 5

I will conduct my own analysis of the problems, using the theory from the previous section, and section 6 will contain a short discussion of what I have achieved in the thesis.

2 Background theory

This section will contain an overview of terminology and concepts used in the rest of the text. This will be from the fields of graph theory and optimization (in particular linear optimization).

2.1 Graphs

A graph is a structure used to describe how different entities are related to each other, through the means of representing each entity and each relation by *nodes* and *edges*, respectively. More precisely an *undirected graph* (or just *graph*) G consists of a set of nodes V and a set E of pairs of these nodes. Each edge $e = (v_1, v_2)$ or v_1v_2 represents a connection between these two nodes. The nodes are said to be the *endpoints* of e, and v_1 and v_2 are said to be *adjacent*. The edge is also *incident to* each of the nodes, and vice versa. Two edges are *adjacent* if they are incident to a common node.

If we demand that the set of pairs of nodes be a set of *ordered* pairs, we obtain a *directed graph*, or *digraph* for short. It is then also common to name the nodes *vertices* and the edges *arcs* instead, and the set of arcs is called A instead of E. In this case an arc $a = v_1v_2$ represents a connection *from* v_1 *to* v_2, but not the other way. These nodes are called the *source* and *target* of a respectively. Note that removing the direction of the arcs in a directed graph simply yields an undirected graph often referred to as the *underlying (undirected) graph*. When using the term graph without qualification we might mean undirected or directed graph, based on the context.
Throughout this section I will give definitions for undirected graphs, with supplements for the directed case where needed .

The *degree* of a node is the number of edges incident to it. For a vertex we distinguish the *indegree*, which is the number of arcs entering the vertex, and the *outdegree*, which is the number of arcs leaving the vertex.

2.1.1 Walks and paths

A *walk* in a graph is an alternating sequence of nodes and edges
$(v_0, v_0v_1, v_1, \dots, v_{n-1}v_n, v_n)$ starting and ending with a node, such that for each edge in the walk the preceding and succeeding nodes are the endpoints of that edge. For a digraph they must be the source and target of the arc, respectively. A walk is *closed* if it begins and ends with the

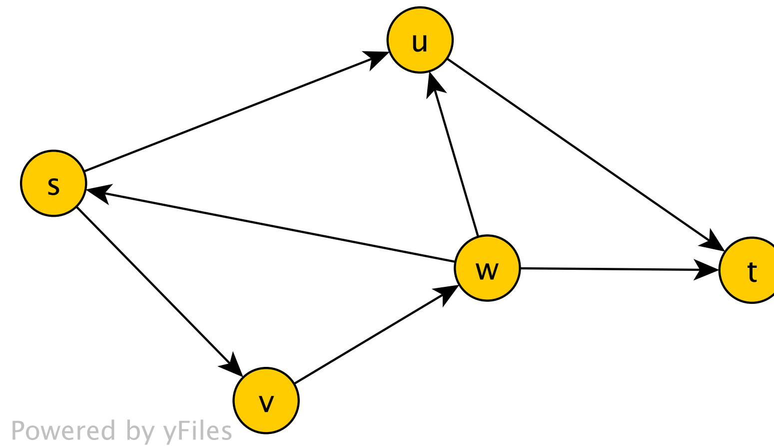

Figure 1: An example graph

same node, and *open* if not.
A walk in which each node is present only once (except possibly the first and last nodes, which may be the same) is called *simple*, and are walks that often occur naturally as solutions to various problems. Think for instance of the problem of finding a shortest path through a graph. Intuitively we may believe this to always be a simple walk, and this will indeed be proven true later in the text. (At least for graphs where such a path exists.)
A walk that is both open and simple is called a *path*, whereas a closed and simple walk is called a *cycle*. A cycle with only one edge is called a *loop*. A (directed) graph containing no (directed) cycles is called *acyclic*.
A (directed) graph is *(strongly) connected* if for any node there exists a path to any other node, and each subgraph $H = (U, F), U \subset V, F = U \cap E$ such that U is connected is called a *component* of G. A directed graph where for any pair of vertices there exists a path from one of the nodes to the other is *weakly connected*.
All graphs in this text are henceforth assumed to be connected, unless otherwise stated.

2.1.2 Trees

Contained in the set of all possible graphs are several interesting subsets or classes of graphs. Among the most important of these are *trees*. A tree is a connected graph with no cycles. This, however, implies a few

other properties that the trees must have.

Theorem 2.1 *For a graph $G = (V, E)$ the following are equivalent:*

a) G is connected and acyclic.

b) G is connected, and $|V| = |E| + 1$.

c) Between any pair of nodes in G there exists exactly one unique path.

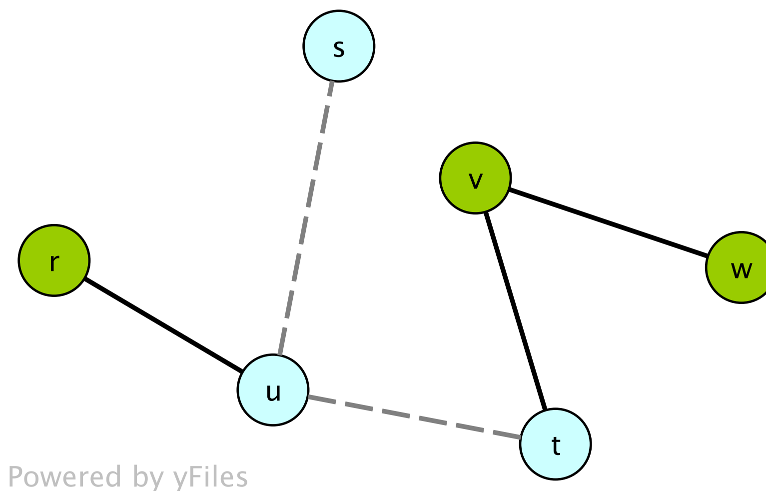

Figure 2: A tree with an $s-t$-path highlighted

2.1.3 Distance, weighted graphs

The notion of distance comes to mind when thinking of a traffic network, be it distance in terms of travel time or in terms of spatial distance. There is also a corresponding definition of distance in graphs.

Definition 2.1 *The length of a walk is equal to the number of edges in it.*

This leads us to the following definition of distance.

Definition 2.2 *The distance between two nodes is equal to the length of a shortest walk between them. If no such walk exists the distance is defined to be ∞.*

This may be used to define a metric in any undirected graph.

i $d(v_1, v_2) = 0 \iff v_1 = v_2$
Follows from the definition.

ii $d(v_1, v_2) = d(v_2, v_1)$
Any path from v_1 to v_2 is also a path from v_2 to v_1 with the same length, when reversed.

iii $d(v_1, v_2) + d(v_2, v_3) \geq d(v_1, v_3)$
Any path from v_1 to v_2 with length l_1 can be combined with a path from v_2 to v_3 with length l_2 to form a path from v_1 to v_3 with length $l_1 + l_2$.

In a directed graph this measure of distance only yields a metric on the underlying undirected graph, because the symmetry requirement fails. We will still define the *distance* between vertices v_1 and v_2 in a directed graph as the length of a shortest path from v_1 to v_2.

Expanding the definition of a graph to also include a function $l : E \to R$ we get a *weighted graph*, where $l(e)$ is the length (or weight) of edge e. These weights are often assumed to be non-negative, i.e. $l : E \to R^+$, and this will also be the case in this text. Now we can give another definition of length and distance in a graph:

Definition 2.3 *The (weighted) length of a walk is equal to the sum of the lengths of the edges in it.*

If we assume the length of each edge to be 1 we see that we recover the first definition of length. We will hereby mean the weighted length/distance whenever we say length/distance.

2.1.4 Capacity, flows

In a road network, the internet, or in any other real life network in which commodities are transported there is some kind of limit to how much stuff can be moved around during unit of time. In a graph this capacity constraint is easily added as another function $c : E \to R^+$ where $c(e)$ denotes the maximum amount of commodity that can be moved along the edge e in one time unit. We will, however, also be interested in the direction of flow along each edge, and because of this we hereby switch our attention over to the directed graphs for the rest of the text.
When working with flow in graphs we also have a function $f : A \to R^+$

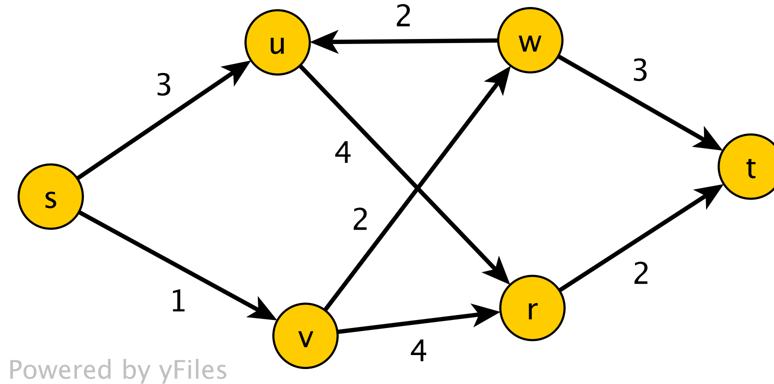

Figure 3: The distance from s to t is 6.

where $f(a)$ denotes the flow currently assigned to arc a. The cost of assigning a flow $f(a)$ along the arc a is

$$f(a)l(a) \tag{1}$$

and the total cost of a flow f is then defined:

Definition 2.4 *The cost of a flow f is*

$$\sum_{a \in A} f(a)d(a)$$

Thinking of a flow situation in which the picture is not altered over time we realize that for each vertex the inflow and the outflow must be equal. Now some nodes may have an innate supply of commodity, such that the flow out into the graph from such a node node is greater than the flow into it. This is a node with a positive *supply*, and it is called a *source*. In the opposite case the node is a *sink*, with a negative supply. We define the supply function $b : V \to R$ such that for each source s we have $b(s) > 0$, for each sink t we have $b(t) < 0$ and for all other nodes v we have $b(v) = 0$. We can then characterize a *balanced flow*.

Definition 2.5 *The flow f is balanced if for each vertex $v \in V$ we have*

$$\sum_{a \in \delta_{out}(v)} f(a) = b(v) + \sum_{a \in \delta_{in}(v)} f(a) \tag{2}$$

where $\delta_{out}(v)$ denotes the leaving arcs of v and $\delta_{in}(v)$ denotes the entering arcs.

This is also called the *flow conservation* property, and will be assumed to hold for any flow f unless otherwise stated.

Definition 2.6 *A flow that satisfies both the flow conservation property and also the capacity constraint*

$$0 \leq f(a) \leq c(a) \ \forall a \in A \tag{3}$$

is called a feasible flow.

For a feasible flow we also talk about the *value* of the flow, which is

Definition 2.7 *The value of a $s-t$-flow f is*

$$\sum_{a \in \delta_{out}(s)} f(a) = \sum_{a \in \delta_{in}(t)} f(a)$$

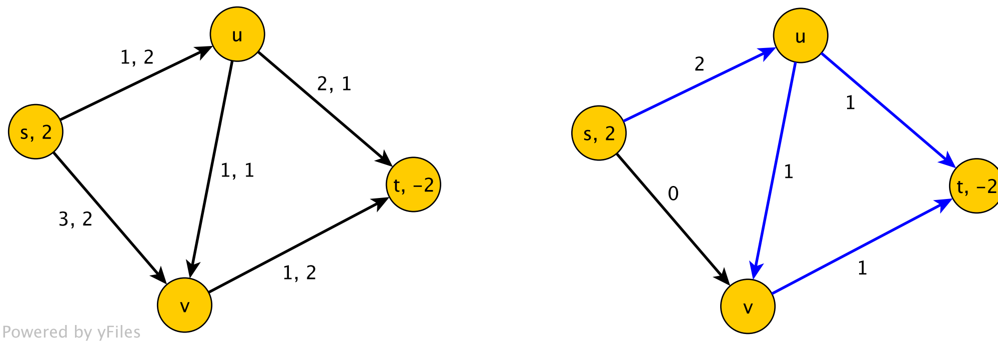

Figure 4: A graph with costs and capacities on each arc, and a feasible flow in the same graph.

2.2 Optimization

The most general form of an optimization problem may be written

$$max\{f(x) : x \in D\}$$

or

$$min\{f(x) : x \in D\}$$

where D is some domain called the feasible region. Multiplying $f(x)$ with -1 we see that the two problems are really the same. About such a general problem there is not much to be said, and thus optimization problems are divided into several categories according to the form of both f and D:

- If both f and D are convex we have *convex optimization*
- If f is also linear and D is a polyhedron we have *linear optimization* or *linear programming*
- If D is a finite set we have *combinatorial optimization*
- If D is also the integer points of a polyhedron we have *integer programming*

We will in this section look at minimizing a *linear function* over a polyhedron, which is then *linear programming*. Our aim is to see that all polyhedron are a combination of *polytopes* and *finitely generated cones*, and that a linear function over such a domain obtains its minimum value in a vertex of the domain, unless the minimum is unbounded.

2.2.1 Convex sets, cones

An import kind of set is the *convex set*.

Definition 2.8 *A set $C \subset R^n$ is convex if for any pair of points $c, d \in C$ and for any $0 \leq \lambda \leq 1$ we have*

$$\lambda c + (1 - \lambda)d \in C$$

i.e. any convex combination of the two points is again in C.

Examples of convex sets are R^n the $n-$dimensional real space, I^n the $n-$dimensional solid square box and D^n the $n-$dimensional ball.
Similar to the convex set we have the *convex cone*.

Definition 2.9 *A set $C \subset R^n$ is a convex cone if for any $x, y \in C$ we also have*

$$\lambda x + \mu y \in C, \lambda, \mu \geq 0$$

i.e. any conical combination of the two points is again in C.

Note that the convex cone is also convex.
We define the *intersection* and *sum* of two sets X, Y.

Definition 2.10 *The intersection of two sets X, Y is given by*

$$X \cap Y = \{z : z \in X, z \in Y\}$$

Definition 2.11 *The sum of two sets X, Y is given by*

$$X + Y = \{x + y : x \in X, y \in Y\}$$

We can easily verify that convex sets and convex cones are closed under both intersection and sum.

A useful construction is the convex hull of a set X.

Definition 2.12 *The convex hull* conv.hull(X) *of a set X is the intersection of all sets containing X. Subsequently it is the minimal convex set (with regard to inclusion) containing X. If X is finite* conv.hull(X) *is a polytope.*

Although this definition is rather abstract, it can be shown that the definition is equivalent to a more useful characterization.

Theorem 2.2 *For a set $X \subset R^n$ we have*

$$\text{conv.hull}(X) = \{x : x = \lambda_1 x_1 + \cdots + \lambda_m x_m, x_i \in X, \sum_i \lambda_i = 1, \lambda_i \geq 0\}$$

If X is finite there exists a subset $X' \subset X$ with $|X'| = n + 1$ such that each x can be expressed uniquely as a convex combination of points in X', and X' are then the vertices of the polytope of X.

Again we have the similar definition of the cone of a set X.

Definition 2.13 *The cone* cone(X) *of a set X is the smallest convex cone containing X. If X is finite* cone(X) *is finitely generated.*

Again it can be shown that the definition is equivalent to a more useful characterization.

Theorem 2.3 *For a set $X \subset R^n$ we have*

$$\text{cone}(X) = \{x : x = \lambda_1 x_1 + \cdots + \lambda_m x_m, x_i \in X, \lambda_i \geq 0\}$$

If X is finite there exists a subset $X' \subset X$ with $|X'| = n$ such that each x can be expressed uniquely as a convex combination of points in X'.

2.2.2 Halfspaces and polyhedron

The polytopes and cones are closely related to another kind of convex set, the *polyhedron*, which is the intersection of a finite number of *halfspaces.*

Definition 2.14 *A halfspace $H \subset R^n$ is a subset of R^n such that there exist a vector $r \in R^n, r \neq 0$ and a real number $\delta \in R$ such that*

$$H = \{x : r^T x \leq \delta\}$$

Since scalar products commute with vector addition in R^n, we see immediately that all halfspaces are also closed convex sets.
Similar to the halfspace we have the *hyperplane*.

Definition 2.15 *A hyperplane $P \subset R^n$ is a subspace of R^n such that there exist a vector $r \in R^n$ and a real number $c \in R$ such that $P = \{x : r^T x = c\}$.*

A hyperplane is also called an *affine subspace*, and in the case when $c = 0$ a *linear subspace*.

Definition 2.16 *A polyhedron $P \subset R^n$ is an intersection of finitely many halfspaces. I.e.*

$$P = \{x : Ax \leq b\}$$

where A is a $m \times n$ matrix that determines the m halfspaces that P is an intersection of.

Since a halfspace is a closed, convex set, any intersection of halfspaces is also a closed, convex set. So all polyhedron are then closed and convex. And intuitively they look very much like polytopes and cones. The relation between the different kinds of sets are given by the following theorem.

Theorem 2.4 *Any polyhedron P is a sum of a polytope Q and a finitely generated cone C, i.e.*

$$P = Q + C$$

, where Q and P have the same vertex set. If P is bounded it equals the polytope Q (or C is empty).

For a proof refer to [1] This means that everything that is true for polytopes (or sums of polytopes and finitely generated cones) is also true for bounded (or general) polyhedron! Due to the explicit definition of the polytopes and finitely generated cones it is often easier to prove attributes of these, than it is for the polyhedron with their implicit definition.
The converse to this theorem is also true, but for us this theorem is of most interest, as we will see in a moment.

2.2.3 Linear Programming

Misleading as the name may be, Linear Programming (LP) has little to do with actual programming, but concerns rather the problem of finding the maximum of minimum value of a linear function over a convex domain. To this end several algorithms have been developed over the years, and among those that stand out are the Simplex algorithm and the interior point methods. Formally we have the *objective function* (i.e. the function to maximize or minimize, and we will hereby assume minimization) $f : P \to R$ where P is the domain polyhedron $P = \{x : Ax \leq b\}$. Our problem is thus to find

$$min\{c^T x : Ax \leq b\}$$

where A is the *constraint matrix* of the LP problem.

Intuitively the LP problem is very easy to solve, as the sets for which the objective function have a constant value are hyperplanes. Thus solving an LP problem is really just the same as moving this hyperplane along its normal vector, decreasing the value of the objective function, until it reaches the boundary of the convex domain. This intuition also tells us that the minimum value of the the objective function is attained in at least one of the vertices of the domain, if at all. The objective function might decrease along a direction in which the polyhedron is unbounded. In this case we say that the LP problem is *unbounded*. The other extreme case is when the polyhedron is empty, i.e. no solution exists at all. This is an *infeasible* LP problem.

Of course finding the solution to an LP problem is not as easy as intuition leads us to believe, but this is where the power of theorem (2.4) can be used:

Theorem 2.5 *For a feasible and bounded LP problem the optimal value is always attained in a vertex of the domain polyhedron.*

Proof. We start with the case of a bounded LP problem. For a non-empty polytope Q and a linear function $f : Q \to R$ the minimum of f is attained in a vertex of Q. Let Q have vertices $v_1, \ldots, v_n$. Then any point $x \in Q$ can be written

$$x = \lambda_1 v_1 + \cdots + \lambda_n v_n, \lambda_I \in [0, 1], \sum \lambda_i = 1$$

Now since f is linear, we get

$$f(x) = f(\lambda_1 v_1 + \cdots + \lambda_n v_n)$$

$$= \lambda_1 f(v_1) + \cdots + \lambda_n f(v_n) \geq \min\{f(v_1), \ldots, f(v_n)\}$$

Since any bounded polyhedron is also a polytope, we then have for bounded LP problems that the minimum is attained in a vertex. Now we might have an LP problem where the domain is unbounded, but the minimum value for the objective function might still exist and be finite. Proving that the optimal value here is attained in a vertex as well requires some extra details. Assume now that the polyhedron P is unbounded and has at least one vertex, and that f has a bounded minimum value that is attained in P. Now we know that $P = Q + C$ where C is nonempty. Assume that there exists a vector $z \in C$ such that $f(z) = f(0) + d, d > 0$. But then $f(nz) = f(0) + nd$, and since $nz \in C$ f is unbounded, which contradicts the assumption that the problem was bounded. Thus $f(z) \leq f(0)$ for all $z \in C$, and we may examine only the points in P of the form $x + y, x \in Q, y \in C$ where $y = 0$, that is we may consider only the points of the polytope Q. Now since we know that the minimum value is attained in the polytope Q, we also know that it is attained in a vertex of Q, which is again a vertex of P by (2.4). □

2.2.4 The dual problem

An important theorem in linear programming concerns a problem related to an original LP problem, called the *dual problem*. For an LP problem

$$min\{c^T x : Ax \leq b, x \geq 0\}$$

the dual problem is

$$max\{b^T y : A^T y \geq c, y \geq 0\}$$

Now the famous *duality theorem* states that if both the original and the dual problems are feasible, then their optimal solutions are the same:

Theorem 2.6 (LP duality theorem) *For a linear optimization problem and its dual we have* $min\{c^T x : Ax \leq b, x \geq 0\} = max\{b^T y : A^T y \geq c, y \geq 0\}$ *if both problems are feasible.*

For a proof refer to [5].

In the case that one of the problems is infeasible we can make use of Farkas' lemma to show that the other problem must be unbounded. Not so much a lemma as a theorem, it states the following:

Theorem 2.7 (Farkas' lemma) *The system* $Ax = b$ *has a nonnegative solution if and only if there is no vector* y *satisfying* $y^T A \geq 0, y^T b < 0$.

For a proof refer to [1]

2.2.5 Convex optimization

Not to be confused with a convex set is a *convex function*

Definition 2.17 *A function $f : D \to R$ is convex if for any $x_1, x_2 \in D$ and $0 < \lambda < 1$ we have*

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2)$$

The field of convex optimization concerns minimizing or maximizing *convex functions over convex domains*. Thus linear optimization is a subfield of convex optimization.
Several algorithms exist for solving convex optimization problems, and among the most significant is the *Frank-Wolfe algorithm*. Briefly this algorithm works as follows:
Start with an initial guess at the solution, x_0. Then approximate the objective function with a linear approximation around x_i, and solve the resulting LP-problem, obtaining the solution x_i'. Now use a convex combination of x_i, x_i' as x_{i+1}, and do a new approximation around x_{i+1}. Terminate when some criterion is met, e.g. improvements are less than some ϵ for each iteration.
The Frank-Wolfe algorithm does not give an exact solution, but rather an approximation to the optimal solution. Unfortunately the improvements made in each iteration of the algorithm decrease rapidly, and obtaining an accurate approximation might require a large amount of iterations, each consisting of constructing and solving a linear optimization problem. Nevertheless the algorithm is often used for convex optimization problems.

2.2.6 Complexity

The *complexity* of an algorithm refers to roughly how many operations need to be performed for the algorithm to terminate. The complexity of a problem equals the smallest complexity of a solution algorithm.
Complexity is used as a measure of how long time a problem will take to solve, based on some measure of problem size. For instance we can consider the problem of finding max(S) for some finite set S of integers. A fastest algorithm that solves this problem is

```
max = S(1)
for (i = 2; i <= |S|; i++)
   if (S(i) > max) max = S(i)
return max
```

If $|S| = n$ this algorithm performs roughly n loops, of which roughly $\log_2(n)$ contain four operations, and the rest contain three. In total this is $3n + \log_2(n)$ operations. However keeping exact track of the number of operations is not of very much interest for the average algorithm theoretician. When n grows large it is clear that $3n$ dominates $\log_2(n)$, so our algorithm has a running time which is roughly $3n$. Again constant terms are not really significant. When comparing $3n$ to n^2 it is clear that for large enough values of n the extra term n dominates the term 3, and so it is customary to also strip all constants. Thus we end up with a running time roughly proportional to n for our algorithm, or we say that it has complexity $O(n)$ - at the order of n. And the problem of finding the largest of n integers then also has complexity $O(n)$.
Note that if we made the assumption that the integers in S were sorted, a fastest algorithm would simply be

```
return S(|S|)
```

which clearly has complexity $O(1)$.

We usually consider the algorithms that have complexity $O(n^m)$ for some fixed m, the *polynomial algorithms*, to be good. And of course the lower the exponent m, the better. If the complexity of an algorithm is $O(e^n)$ we say that the time is *exponential*, which is bad. Many problems have exponential complexity, and these are ones we'd rather not solve precisely. In these cases approximation algorithms with polynomial complexity may often be used to find an approximate solution of the problem instead.

3 The traffic routing problem

In fact there is not *the* traffic routing problem, but this involves rather a large amount of related problems. In the introduction the following examples were given:

- Routing traffic through a city, with as little congestion as possible.
- Routing internet traffic, with as little delay as possible.
- Routing containers between harbors, while transporting empty ones as short distances as possible.

There was also a mention of the situation in which traffic is *not* directed in any way. and that this could lead to another traffic routing. These listed situations are examples of *system optimal routing* where we try to minimize some measure of badness. On the other hand this second situation would be a *user equilibrium* where the behavior of the users decide the solution. Both scenarios can be studied using static network flow formulations.
In addition one might add the dimension of time, leading to some slightly harder problems. An example of this could be to route a given amount of traffic through a network over time, and in such a way that the total travel time of all the traffic was minimized. This would typically involve avoiding congestion, and would be calculating the *dynamic system optimal routing*. On the other hand one could also study how this traffic would route itself if no interference was done, and thus obtain the *dynamic user equilibrium*. Much effort is also being put into researching the relation between the system and user equilibria, as finding a way to use e.g. proper taxing to obtain a user equilibrium that equals the system optimal solution would be rather splendid in a lot of real world scenarios.

3.1 Traffic model

In order to bring the real world problem of examining traffic in road networks over to the mathematical workbench, we need a model that represents the original problem. This subsection will draw some outlines for such models.
We always represent the network in question as a directed graph where traffic flows along the arcs of the graph. Now the two most important factors that characterize a stretch of road in the real world is, at least

for me, the speed at which traffic flows and the amount of traffic that there is room for along the road. Typically these two are inversely proportional, something we see if we assume that each car desires to have a certain amount of time to the car in front. Then this distance increases proportionally with the speed of the car, and the amount of cars there are room for per length then decrease. Let's say that a sensible amount of time to have between two cars is τ. In a sense this determines an absolute capacity on this road, where the inflow rate of traffic cannot be greater than $\frac{1}{\tau}$ cars per time unit for each lane of the road. But in fact there is another mechanism that causes slowdown *before* this limit is reached: When the concentration of cars is rather high, and one car breaks, the one behind it will also have to break immediately to retain the τ time distance to the car in front. But the second car does probably not react instantly to the car in front, and thus has to break more than the first car in order to stay far enough behind. Now the third car, behind the second one, will have to break even more, and so on.
This all leads us to think of the travel time along a stretch of road as a non-decreasing function of traffic concentration. And of course the actual length of the road also factors into the travel time as one would expect.

So we represent the network as a digraph where each arc has a latency function $l_a(x_a)$, $l_a : R^+ \to R^+$ dependent on the flow assigned to the arc. Note that this is the function we use as the length of each arc when considering the network only as a graph. This latency function is usually assumed to be *convex and nondecreasing*, and so we will assume here. In addition each arc may have a capacity constraint $c(a)$, $c : A \to R^+$, but when the cost function is increasing, this might also play the role of a capacity in limiting the amount of flow assigned to the arc.

In the dynamic case we cannot use the simple network flow model anymore, but must expand of change the traffic model to describe the added time dimension. In this case our latency functions are often much more complex, consisting of differential equations or the like, to accommodate for queues and variable latency situations. However this will not be the main focus of this text, and when examining the dynamic problems I will assume the latency functions to be of a rather simple kind that allows for only a small expansion of the network flow model.

Each traffic agent must have an origin and a destination, but since we are not treating the commuters individually we use sources and sinks of continuous *flow*. Since it is significant where the flows run from and to, we have to treat travelers originating from a vertex s and destined to a vertex t differently from other travelers, that is we need to distinguish

$s - t$-flows from all other flows for each pair (s, t). To this end we introduce one *commodity* for each origin-destination pair, and index these with $i \in I$. Thus we have $|I|$ flow functions $f_i(a)$, $f_i : A \to R^+$, and the total flow along arc a is

$$x_a = \sum_{i \in I} f_i(a)$$

Everyone knows how boring it is to be stuck in traffic jams. In fact most people would agree that spending time at home of at work is preferable to spending it in a car or on a bus altogether. So we assume that all traffic agents are interested in minimizing their time spent in traffic; their *travel time*. And of course the travel time experienced by one traffic agent is equal to the sum of the travel time along each road of the agent's route, from origin to destination, or in other words the length of the path (or route) the traveller takes in the network.
In the dynamic cases another assumption we make is that each of our commuters have got a certain time they wish to arrive at their destination (and possibly also a desired departure time). And when too many have the same desired arrival time the roads get crowded at some time intervals, causing congestion and delays. In this case each commodity has it's own *departure cost function* $g(t)$, $g : R \to R^+$ and *arrival cost function* $h(t)$, $h : R \to R^+$. Both of these are assumed to be convex, and to avoid that spending time in traffic is preferable to spending time at home or at work we also require that $h'(t) \geq -1$, $\forall t$ and $g'(t) \leq 1$, $\forall t$.

To avoid congested traffic situations we can consider routing traffic along alternative paths to relieve the most heavily used roads. Another possibility is to hurry or delay departures such that not everyone enters the network at the same time. But how do we route traffic? The general assumption is that each traffic agent does exactly what's best for themselves, i.e. totally selfish behavior. And as we shall see this may cause much more congestion than what is indeed necessary! Finding a way to make the selfish user equilibrium and the altruistic system optimal solution coincide will be the ultimate goal of this analysis.

3.2 Static system optimum

The first problem we will look at is finding the system optimal routing in a static setting. The motivation for this problem is that we wish to route a static flow - perhaps the peak traffic causing the usual jams in cities during the morning and afternoon hours - through a given traffic network. The data we are given is the network itself, represented

as a graph, where each arc has a *travel time function* (or *latency function*) dependent on the flow along it. The arcs may also have capacity constraints. In addition we have a set origin and destination pairs, each with a flow of a certain magnitude that needs to flow between them. Each of these pairs correspond to one commodity.
What we wish to do is minimize the *total travel time of all commuters*. The total travel time of all traffic is given by:

$$\sum_{a \in A} x_a l_a(x_a) \tag{4}$$

where x_a is the total flow along arc a and l is the latency function giving the travel time along the arc as a function of traffic flow. Now this flow is a composition of flows between several origin-destination pairs. Let these pairs be indexed by the set I, and let $f_i(a)$ denote the amount of flow of commodity i along arc a, such that

$$x_a = \sum_{i \in I} f_i(a)$$

We also require flow conservation of each of the i flows. If $b_i(v)$ is the supply of commodity i at vertex v we can express these constraints as

$$\sum_{a \in \delta_{out}(v)} f_i(a) = b_i(v) + \sum_{a \in \delta_{in}(v)} f_i(a) \ \ \forall v \in V, i \in I \tag{5}$$

In addition we have non-negativity constraints on the flows:

$$f_i(a) \geq 0 \ \ \forall a \in A, i \in I \tag{6}$$

and capacity constraints along each arc:

$$\sum_{i \in I} f_i(a) \leq c(a) \ \ \forall a \in A \tag{7}$$

This is quite a problem to solve, but if we study it more closely we see that all our constraints are linear equalities or inequalities. In other words our feasible domain is a polyhedron. Since the latency function is convex we see that our problem fulfills the criteria for being a convex optimization problem! Thus we know one way of solving it, although a rather general and possibly not very efficient way. In an attempt to gain some more insight into this problem, I will look at some special cases of it later, in section 5.

3.3 Static user equilibrium

The second problem I will consider is finding the user equilibrium routing in a static setting. The motivation for this problem is to calculate what flow we will actually get in a given network if we let the users determine the flow of traffic. Another aim is to find some characterizations of these user equilibria, which may then be used in making a system optimal solution become a user equilibrium by taxation. The data we are given is exactly the same as in the system optimal problem above.
How to solve this problem is not intuitively easy, but we can start with the famous principle of Wardrop:

Postulate 3.1 (Wardrop's first principle) *The journey times in all utilized routes are equal, and equal to or less than those which would be experienced by a single vehicle on any unused route.*

An equivalent formulation, viewing each traffic agent as a player, is that the user equilibrium is a *Nash equilibirum*:

Postulate 3.2 *In a user equilibrium no traffic agents can improve their travel times by unilaterally changing routes.*

This definition can not be used directly to calculate the user equilibrium, as the number of players is too great. We don't even treat them individually in our flow model. Luckily we can formulate an optimization problem that gives us the user equilibrium! Consider the function

$$\sum_{a \in A} \int_0^{x_a} l_a(x)dx \tag{8}$$

where

$$x_a = \sum_{i \in I} f_i(a)$$

We want to show that this function actually is minimal exactly when the postulates above hold. Let vertices s, t be the source and sink of f_i for some $i \in I$, let P, Q be two $s - t$-paths, and assume the cost of P is less than the cost of Q. Now let A_p be the arcs in p that are not also in q, and similarly for A_q. Then

$$\sum_{a \in A_p} l_a(x_a) < \sum_{a \in A_q} l_a(x_a)$$

and shifting a flow of magnitude δx from q to p changes the value of (8) by

$$\sum_{a \in A_p} \int_{x_a}^{x_a + \delta x} l_a(x)dx - \sum_{a \in A_q} \int_{x_a - \delta x}^{x_a} l_a(x)dx \tag{9}$$

which is negative for sufficiently small δx since l_a is non-decreasing. Thus all paths between each origin-destination pair have equal cost when (8) is minimal. And we can formulate the problem of finding the user equilibrium as minimizing (8) subject to the same constraints (5 - 7) as the system optimal problem.

Again the assumption that each latency function is convex (or just non-decreasing, in fact) makes this a convex optimization problem, solvable by known algorithms. I will also look at special cases of this problem later, as well as compare the user equilibrium flow and the system optimal flow in the same network.

3.4 Dynamic system optimum

The third and by far the hardest problem is that of finding a system optimal solution to a flow that changes over time. The motivation here is that we have a certain amount of traffic we want to route through a given network, as opposed to a flow of a certain magnitude. Now all the traffic can not be moved at the same time due to capacity constraints, so we must spread it out over a period of time. This period of time is assume to be $t \in [0, T]$ For the commuters this means the extra cost of departing and arriving at less than optimal times. We expect the system optimal solution to respect this, by finding a routing of the traffic that minimizes both time spent in traffic and deviations from the preferred departure and arrival times.
For the traffic planner it means we are no longer dealing with flows and capacities along the arcs as real numbers, but as real valued functions $f_i(a, t)$ of arcs and time. Thus we have a vastly larger domain to optimize over.
In addition we must be able to calculate the travel time along each arc at any given time τ, and we need a *latency model* for doing this. Now this calculation is dependent not only on the inflow at $t = \tau$, but also the inflow to the arc at all times $t < \tau$ We assume that inflow at $t > \tau$ does *not* influence the travel time at $t = \tau$, and this is called *causality* of our latency model. This ensures that we can in fact calculate the travel time at $\tau = t$ if we know the inflow and travel times up to this point of time. It should also be impossible to arrive at an earlier time by choosing the same route, but departing later. This encompasses the *FIFO*, or queue, principle; the First In are the First Out. Lastly we require that the total outflow from each arc must equal the total inflow to that arc, which is the *conservation of traffic*. We end up with *inflow functions* $f_i^{in}(a, t), x_a^{in}(t)$,

outflow functions $f_i^{out}(a,t), x_a^{out}(t)$ and latency functions $l_a(x_a^{in},t)$ for each arc a at time t, where again

$$x_a^{in}(t) = \sum_i f_i^{in}(a,t)$$

and similarly for x_a^{out}, f_a^{out}. The relation between x_a^{out} and x_a^{in} needed to satisfy the conservation of traffic is given by

$$x_a^{out}(t + l_a(x_a^{in},t)) = x_a^{in}(t)\frac{1}{1 + \frac{\delta}{\delta t} l_a(x_a^{in},t)} \tag{10}$$

and similarly for $f_i^{in}(a,t), f_i^{out}(a, r + l_a(x_a^{in},t))$. This is obtained by differentiating the integral of inflow up to time t and outflow up to time $t + l_a(x_a^{in},t)$, which must be equal.
The FIFO principle directly translates as

$$l_a(x_a^{in},t) + \Delta t \leq l_a(x_a^{in}, t + \Delta t)$$

which implies

$$\frac{\delta}{\delta t} l_a(x_a^{in},t) \geq -1$$

And in addition we assume that whenever $x_a^{in}(t) > 0$ we have

$$\frac{\delta}{\delta t} l_a(x_a^{in},t) > -1$$

These are all properties that need to be satisfied by our latency model, and the models that are usable in this sense range from very simply to very complex. The model I choose later in the text is quite simple.

We are ready to state the dynamic system optimal traffic routing problem:
The total travel time of all traffic agents is given by

$$\sum_{a \in A} \int_0^T x_a^{in}(t) l_a(x_a^{in},t) dt \tag{11}$$

Assuming each commodity has a common departure cost function $g_i(t)$ and arrival cost function $h_i(t)$, denoting the source and sink of commodity i by s_i, t_i, and letting

$$b_i(v,t) = \sum_{a \in \delta_{out}(v)} f_i^{in}(a,t) - \sum_{a \in \delta_{in}(v)} f_i^{out}(a,t)$$

denote the difference in outflow and inflow of commodity i at vertex v at time t, the total departure and arrival time deviation cost is

$$\sum_i \int_0^T b_i(s_i,t)g_i(t) - b_i(t_i,t)h_i(t)dt \tag{12}$$

Thus the minimization in the dynamic system optimality problem is

$$\min\{\sum_{a\in A}\int_0^T x_a^{in}(t)l_a(x_a^{in},t)dt + \sum_i \int_0^T b_i(s_i,t)g_i(t) - b_i(t_i,t)h_i(t)dt\} \tag{13}$$

such that the below constraints all hold.

The flow balance constraints, when allowing excess traffic to remain temporarily at each vertex, are for the non-source or -sink vertices of commodity i

$$\int_0^t b_i(v,\tau)d\tau \leq 0 \tag{14}$$

or written out

$$\int_0^t \sum_{a\in\delta_{out}(v)} f_i^{out}(a,\tau)d\tau \leq \int_0^t \sum_{a\in\delta_{in}(v)} f_i^{in}(a,\tau)d\tau \tag{15}$$

with the inequality replaced by an equality at time $t = T$. If we do not allow excess traffic to remain at internal vertices, we replace the inequality with an equality at all times. For the source vertex s_i the inequality is relaxed by adding b_i on the right hand side

$$\int_0^t b_i(s_i,\tau)d\tau \leq b_i$$

and for the sink t_i the equality at time $t = T$ is

$$\int_0^T b_i(t_i,\tau)d\tau = -b_i$$

Note that the supplies at sources and sinks are not given explicitly as functions of time, but are consequences of flow balance and total supply at the terminal time $t = T$.
In addition the capacity constraints

$$x_a^{in}(t) \leq c(a,t) \tag{16}$$

and the non-negativity constraints

$$f_i(a,t) \geq 0 \tag{17}$$

apply as usual.

In this case we are very far from having solved the problem, even though we have formulated it precisely. The unknowns are no longer points in R, but functions from R^+ into R^+. This is a problem since the optimization methods we have mentioned will no longer be applicable. In addition the latency functions $l_a(x_a^{in}, t)$ are functions of x_a^{in}, which are themselves functions of t, and the functions l_a may not at all be simple; maybe even inexpressible.

3.5 Dynamic user equilibrium

Finding the dynamic user equilibrium will not be treated as a separate problem here, but I will look at some characterizations of dynamic user equilibrium flows in the analysis concerning the system optimal case. The reason for this is twofold: The problem is rather hard, and I am not as interested in finding the user equilibrium as I am in finding the system optimal solution. What I am interested in is rather conditions that ensures a flow is a user equilibrium.

4 Existing work and solution algorithms

As expected both optimization and graph theory in general, and traffic planning specifically, has received lots of attention through the years, and the amount of articles on the latter is vast. This section contains the theory and algorithms I have found useful for solving to the traffic assignment problems, and most of it is general theory found everywhere in the literature.

4.1 Distance and shortest paths

In section 2 we defined the length of a walk as the sum of the length of each edge in the walk, counting multiplicity. We also defined the distance from one node s to another t as the length of a shortest walk from s to t, but we never said how to find this distance. This is clearly something we might be interested in. Let's try to solve this problem in a graph where all edges have length 1:
One way of doing this could be to try all possible walks from s to t, and find the minimum of the lengths of these. But there is one problem: if the graph contains any (directed) cycle reachable from s we will end up trying to go through this cycle one time, two times, three times and so on to each time form a different walk. This produces infinitely many different walks, and we will never terminate our search for the shortest walk!
Let us therefore try to focus our attention on just the simple walks, or paths, from s to t. Again we might try all different paths from s to t and use the length of one of the shortest ones as the distance. Since there are only finitely many nodes in our graph any path will be of finite length, and we thus have only a finite amount of possible paths to examine. This number might nevertheless be horrendously huge! Imagine a graph with n nodes, where there is an edge between any pair of nodes. This is called the *complete* graph of order n. Then the number of different $s-t$-paths is

$$\sum_{i=1}^{n-1}(i-1)!\begin{pmatrix} n-2 \\ i-1 \end{pmatrix}$$

Considering that a graph with 100 nodes is not at all large, this certainly is a problem.
A different approach is needed. We will pursue a simple but nice idea that actually inspires several more advanced algorithms later on:
Knowing all nodes reachable from s in k steps, find all nodes reachable

in $k+1$ steps. When node t is encountered in the l-th step, we have that the length of a shortest walk from s to t is exactly l. Let's describe an algorithm, called a breath first search (BFS), in more detail:
Let $V_i, i = 0 \dots |V|$ be the set of nodes reachable from s in minimum i steps. Let U be the set of unvisited nodes, let $d(s,t$ be the distance from s to t and let $\pi : V \to V$ be a mapping we will use to determine the actual shortest path from s to any other node.

Initialization	$V_0 \leftarrow \{s\}, V_i \leftarrow \emptyset \ \forall i > 0$
	$U \leftarrow V \setminus V_0$
	$d(s,s) \leftarrow 0,\ d(s,u) \leftarrow \infty \ \forall u \in U$
	$k \leftarrow 1$
Loop	while $t \notin V_k$ and $k < \|V\|$:
	for $v \in V_k$:
	for $u \in \delta_{out}(v) \cap U$:
	$d(s,u) = k$
	$\pi(u) \leftarrow v$
	$V_{k+1} \leftarrow V_{k+1} \cup \{u\}$
	$U \leftarrow U \setminus \{u\}$
	$k \leftarrow k+1$

When the algorithm terminates we will have calculated $d(s,t)$, and we can also find the reverse of the walk used to reach t by repeatedly applying π, beginning with $\pi(t)$.

Theorem 4.1 *The breadth first search algorithm finds a shortest path from s to t, if such a path exists.*

Proof. Assume that $t \in V_k$ for some k, and also that the nodes in V_i are precisely the nodes reachable from s in minimum i steps. Then $d(s,t) = k$ after the termination of the algorithm is in fact the distance from s to t. Since each application of π on a node in V_i yields a node in V_{i-1} we will reach a node in V_0 after k steps, beginning with t. This node must be s, and reversing the direction we find a path from s to t of length k.
We must show that the nodes in V_i are precisely what we claim they are, and will do so by induction on i:
For $i = 0$ the claim is obviously true. Now assume the claim holds for $i = 0 \dots l$. Then for u, v as in the loop section of the algorithm a path from s to u of length $l+1$ is easily obtained by combining a path from s to v with the edge (v,u). Assume that there exists a path from s to u with length $j < l+1$. Then $u \in V_j$, and has thus already been removed from U, contradicting the assumption that $u \in U$.

Now assume that $t \in U$ after the algorithm terminates. Thus $d(s,u) = \infty$. We must show $u \in U \iff$ no walk from s to u exists.
$\Rightarrow$:In a graph with n nodes (of *order* n) there are no paths of length $\geq n$. This is because in a path each node is visited only once, and at each step a new node is visited, making the maximum possible length of a path $n-1$. So if t is not reachable from s in n steps, we can conclude that no path of any length exists from s to u.
$\Leftarrow$: If there exists a walk from s to t, there must also exist a path from s to t, obtained by eliminating all cycles from the walk. So if no walk from s to t exists, neither does a path of any length, so t is unreachable from s and remains in U through the whole algorithm. □

Directly from the algorithm we can also see that the algorithm is quite fast:

Corollary 4.1 *The above algorithm finds a shortest path from s to t in at most $|E|$ steps, if such a path exists.*

Proof. Each edge is processed exactly one time. □

Thus we have not only obtained an algorithm for finding a shortest path (and the distance) between a pair of nodes (or vertices!), but we have also obtained a *fast* algorithm. And the idea we have used here, of examining the 'closest' nodes first, will be the basis of more advanced algorithms later, among which the Dijkstra-Prim algorithm is probably the most famous.

4.2 Weighted shortest path

In the section above we assumed all edges had length 1. We will now look at the case where the length of each edge may be any positive real number. Finding a shortest path between a pair of nodes is not quite as easy as when all edges have unit length, but thanks to Dijkstra and Prim there exists a not too difficult algorithm nonetheless. We will define and prove the algorithm for directed graphs:
We want to find the shortest path from s to t for all $t \in V$. We assume that G has no directed cycles of negative length.
Let U be the set of unvisited nodes, let $f : V \to R^+$ be the $s-v$-distances we wish to calculate and let $\pi : V \to V$ be used for keeping the reverses of the shortest paths from s to all visited nodes v.

Initialization	$U \leftarrow V$
	$f(s) \leftarrow 0, f(t) \leftarrow \infty \ \forall \ t \in U \setminus \{s\}$
Loop	while $U \neq \emptyset$:
	find $u \in U$ s.t. $f(u) = \min\{f(u) : u \in U\}$
	for $a = uv \in \delta_{out}(u)$ s.t. $f(v) > f(u) + l(a)$:
	$f(v) \leftarrow f(u) + l(a)$
	$\pi(v) \leftarrow u$
	$U \leftarrow U \setminus \{u\}$

Theorem 4.2 *The function f gives the distance (of a shortest path) from s to t for all $t \in V$. If no such path exists the distance is ∞.*

Proof. Let $d(s,t)$ be the distance from s to t. We will show that for each u chosen in the loop, we have $f(u) = d(s,u)$. Thus $f(u) = d(s,u) \ \forall \ u \in V \setminus U$ by induction. Clearly this holds initially, when $V = U$. Note that $f(u) \geq d(s,u) \ \forall \ u \in V$ always holds, since $f(u)$ is the length of *some* path from s to u. Now assume $f(u) > d(s,u)$. Then a shortest $s-u$-path $(s, a_1, v_1 \ldots, v_{n-1}, a_n, u)$ must pass through U. Let i be the smallest index for which $v_i \in U$. Now if we can show $f(v_i) \leq d(s, v_i) \leq d(s,u) < f(u)$, we have a contradiction to $f(u)$ being minimal. So we need to show $f(v_i) \leq d(s, v_i)$: If $i = 0$ then $f(v_i) = f(s) = 0 = d(s,s) = d(s,v_i)$. If $i > 0$ then we must have $f(v_i) \leq f(v_{i-1}) + l(v_{i-1}v_i) = d(s, v_{i-1}) + l(v_{i-1}v_i) = d(s, v_i)$. □

It can also be show that the running time of the algorithm, with the set U implemented as a *heap*, is rather good:

Theorem 4.3 *The Dijkstra-Prim shortest path algorithm (with heaps) has a running time of $O(|A| log_2(|V|))$.*

A typical use for the Dijkstra-Prim algorithm is to find the shortest path through a graph for someone who wishes to travel from one place to another in the graph, but since this is such an abstract notion the algorithm obviously has many uses. For instance it can be used for determining maximum flows through graphs when applied repeatedly to a series of *residual graphs*, as we shall see later in this section. Another smart use of it is to construct a graph in such a way that finding a shortest path through it solves another, maybe more confusing, problem.

In the case the graph has negative cost arcs we can no longer use Dijkstra's algorithm to find shortest paths. But in this case we we can still use the *Bellman-Ford* algorithm for the same purpose. This algorithm looks perhaps more like the breadth first search. Assume we want to find the shortest $s-v$-paths in the graph, for some s. Let f and π be as

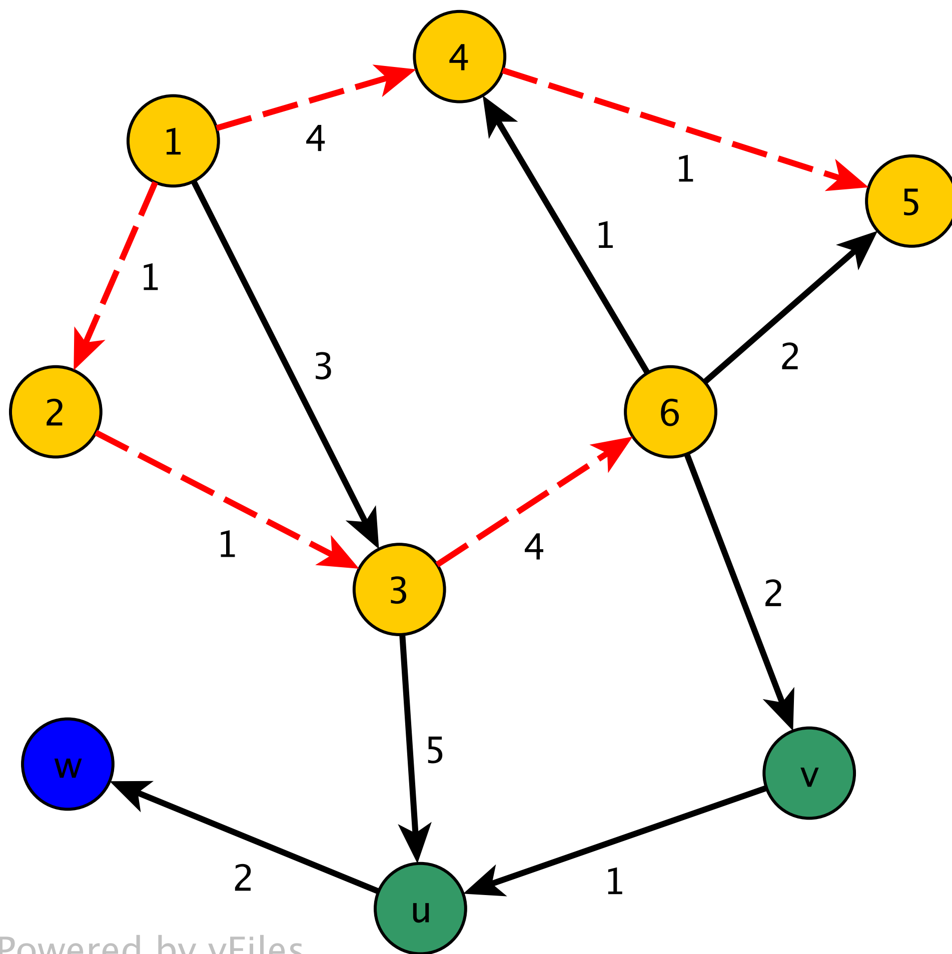

Figure 5: Snapshot of the Dijkstra-Prim algorithm. The vertices are labeled with the order in which the are chosen, and arcs used for the shortest paths are dashed. u is the next vertex to be picked. w is not examined at all yet.

above.

Initialization	$f(s) \leftarrow 0, f(v) \leftarrow \infty \ \forall v \in V \setminus \{s\}$
Loop	for $i = 1 \ldots \lvert V \rvert$:
	for $a = uv \in A$:
	if $f(u) + l(a) < f(v)$:
	$f(v) \leftarrow f(u) + l(a)$
	$pi(v) \leftarrow u$

Theorem 4.4 *The Bellman-Ford algorithm computes a shortest $s - t$ path in $O(|A||E|)$ time, if such a path exists.*

For a proof refer to [1].

Note that in the case when G contains a negative cost directed cycle this can be detected by examining π. If applying π several time results in returning to some vertex, then there is a negative cost directed cycle, and the distances computed by the algorithm may be wrong.

4.3 The Simplex algorithm

Having proved that an optimal value to an LP problem, if it exists, is always attained in a vertex, an algorithm that examines the vertices of the feasible polyhedron sounds like a good idea for solving the given LP problem. And in fact such an algorithm exists, thanks to Danzig. This algorithm is the famous Simplex algorithm, about which several books have been written. I will give a short description of the algorithm, and why it works, some discussion on complexity and average number of iterations, and then suggest some alternative methods for solving LP problems.

4.3.1 Basic idea

The basic idea idea of the Simplex algorithm is that since the optimal solution of the LP problem is attained in a vertex, we can look at only the vertices of our feasible region, if any. Now finding a vertex of the feasible region is not necessarily easy, but if we have found one vertex, finding an adjacent one is no problem. Remember that a point is a vertex of the polyhedron if and only if it satisfies to equality a number of the linear independent inequalities equal to the dimension n of the space. Then moving from one vertex to an adjacent one is done by exchanging one of

those equalities with another of the inequalities currently not satisfied to equality, by moving along the $n-1$ equalities until the boundary of another halfspace is encountered. Now choosing to always travel in a direction in which the objective function is nondecreasing (taking care not to go back to the same vertex twice) will eventually lead to a vertex with the optimal value of the objective function! Doing all this sounds like a lot of book-holding, but all this is beautifully kept track of by the Simplex algorithm, as we shall see.

4.3.2 Basic and non-basic variables

Let's consider the problem $min\{c^T x : Ax \le b, x \ge 0\}$, where A is an $m \times n$ matrix, i.e. $x \in R^n$, and there are $m+n$ inequalities (m from the matrix, and n from requiring non-negativity of x). Now obviously expressing a point x can be done with n basis vectors. However the simplex algorithm introduces m extra ones, so that there is one $x_i, i = 1, \dots, n$ for each dimension and one $x_i, i = n+1, \dots, n+m$ for each inequality. Now $x_i = 0$ means that inequality i is satisfied to equality, thus making it easy to see which ones are. For example $x_1 = 0$ means that the first coordinate of x is 0, and $x_{m+n} = 0$ means that the last inequality from the matrix A is satisfied to equality. Note that since the extra variables $x_{n+1}, \dots, x_{n+m}$ are a measure of how far from equality equation m is, these variables are also called *slack variables.* Now an example will certainly help clear things up. Consider the following LP:

$$min\{c^T x : Ax \le b, x \ge 0\}$$

$$c = \begin{bmatrix} 5 \\ -4 \\ 3 \end{bmatrix} A = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 1 & 2 \\ 3 & 4 & 2 \end{bmatrix} b = \begin{bmatrix} 5 \\ 11 \\ 8 \end{bmatrix}$$

Here we will use x_1, x_2, x_3 as the basis vectors of R^3, and the vectors x_4, x_5, x_6 for each of the three inequalities in A, by transforming the problem into the following, equivalent problem:

$$min\{c^T x : x' = b - Ax, x, x' \ge 0\} \text{where } x' = \begin{bmatrix} x_{n+1} \\ \dots \\ x_{n+m} \end{bmatrix}$$

Another equivalent problem which we will use when dealing with the matrix notation for LP problems is the following:

$$min\{c'^T x' : A'x' = b, x' \ge 0\} \text{where } x' = \begin{bmatrix} x_1 \\ \dots \\ x_{n+m} \end{bmatrix}, c' = \begin{bmatrix} c \\ 0 \end{bmatrix}, A' = [A\ I]$$

Written out the first formulation of the problem looks like:

$$\begin{array}{rrcrcrcrcr} \text{minimize:} & f & = & & & 5x_1 & - & 4x_2 & + & 3x_3 \\ \text{subject to:} & x_4 & = & 5 & - & 2x_1 & - & 3x_2 & - & x_3 \\ & x_5 & = & 11 & - & 4x_1 & - & x_2 & - & 2x_3 \\ & x_6 & = & 8 & - & 3x_1 & - & 4x_2 & - & 2x_3 \\ & & & & & & & & & x \geq 0 \end{array}$$

Now the book-holding of the Simplex algorithm is done by assuming that all the variables appearing on the first line, i.e. x_1, x_2, x_3 in this case, are all 0. This means we are in fact looking at the point $(0,0,0) \in R^3$, since x_1, x_2, x_3 correspond to the basis vectors of R^3. Now this makes it very easy to check the value of the objective function: It is 0. These variables are called the *non-basic variables*. Now checking the value of the variables x_4, x_5, x_6, we see that they are $5, 11, 8$ respectively, and so they are all greater than or equal to 0, and we see that our point is feasible. Now this might not always be the case in the starting set-up like here, but there are ways to deal with that. The variables appearing on the left hand side of the equations, in this case x_4, x_5, x_6, are called the *basic variables*.
Having seen that we are in fact at a feasible point of our LP we can begin looking for a new vertex that improves the value of the objective function. Looking at the expression $f = 5x_1 - 4x_2 + 3x_3$ we see that increasing x_1, x_3 would lead to an increase in f, whereas increasing x_2 would in fact lead to a decrease in f! Now let's try to do exactly this. How we will do this is to exchange x_2 with one of the basic variables, making x_2 basic and setting the other variable to 0. This is called a *pivot*. To do this we see that we already have each of the basic variables expressed as linear functions of the non-basic variables, and it is then easy to find an expression also for a non-basic variable in terms of the other non-basic variables and one basic one. Then after choosing a non-basic and a basic variable we can simply substitute all occurrences of the non-basic variable with its expression in terms of the chosen basic and the other non-basic variables. Now the point is to choose the right variable to exchange x_2 with. We see that increasing x_2 will lead to a decrease in all of the basic variables, and taking into consideration that each of them must still be non-negative after the pivot, we see that we can check which one of the basic variables first becomes 0 as we are increasing x_2. In this example we see that x_4 will be 0 when x_2 is $\frac{5}{3}$, which is the smallest value of x_2 that will make any of the basic variables equal to 0, so the variable we must pivot on is thus x_4. Looking at the expression $x_4 = 5 - 2x_1 - 3x_2 - x_3$ we see that we can express x_2 as

$x_2 = \frac{1}{3}(5 - 2x_1 - x_4 - x_3)$. We then substitute each of the occurrences of x_2 by this expression and obtain the following:

$$
\begin{array}{rrcrcrcrcr}
\text{minimize:} & f & = & -\frac{20}{3} & & \frac{23}{3}x_1 & + & \frac{4}{3}x_4 & + & \frac{13}{3}x_3 \\
\text{subject to:} & x_4 & = & \frac{5}{3} & - & \frac{2}{3}x_1 & - & \frac{1}{3}x_4 & - & \frac{1}{3}x_3 \\
 & x_5 & = & \frac{28}{3} & - & \frac{10}{3}x_1 & + & \frac{1}{3}x_4 & - & \frac{5}{3}x_3 \\
 & x_6 & = & \frac{4}{3} & - & \frac{1}{3}x_1 & + & \frac{4}{3}x_4 & - & \frac{2}{3}x_3 \\
 & & & & & & & & & x \geq 0
\end{array}
$$

Now all the non-basic variables appear with positive signs in front, meaning that increasing any of them above 0 will make the objective function greater. In other words we have already obtained an optimal solution, which is $f = -\frac{20}{3}$, and the point in which this value is attained is $x_1 = 0$, being a non-basic variable, $x_2 = \frac{5}{3}$, being a basic variable, and $x_3 = 0$, again being a non-basic variable. Or $(0, \frac{5}{3}, 0)$ in short.

Now in general we can expect to have several non-basic variables with negative sign in the objective function, and choosing which one should enter the basis can be done in several ways. One common method is simply to choose the variable with the greatest negative coefficient, and if there are ties, just choose one of them. This is known as the *greatest coefficient rule.* Now to determine the variable leaving the basis when there are ties, a common method is the *lexicographical pivot rule* in which each of the slack variables are increased by a arbitrarily small value at the start of the algorithm. We define

$$
0 < \epsilon_{n+m} << \epsilon_{n+m-1} << \cdots << \epsilon_{n+1} << \text{all other data}
$$

and for each slack variable x_i we add ϵ_i to the right hand side of the equation determining x_i. Using this perturbation our starting dictionary in the problem above would look like this:

$$
\begin{array}{rrcrcrcrcr}
\text{minimize:} & f & = & & & 5x_1 & - & 4x_2 & + & 3x_3 \\
\text{subject to:} & x_4 & = & 5 + \epsilon_1 & - & 2x_1 & - & 3x_2 & - & x_3 \\
 & x_5 & = & 11 + \epsilon_2 & - & 4x_1 & - & x_2 & - & 2x_3 \\
 & x_6 & = & 8 + \epsilon_3 & - & 3x_1 & - & 4x_2 & - & 2x_3 \\
 & & & & & & & & & x \geq 0
\end{array}
$$

Now the idea here is that this perturbation of the original problem is so small that it does not change the solution, and can thus be removed again when an optimal dictionary is found, but that it makes the choices of leaving variables during the algorithm unambiguous, thus preventing the algorithm from going in circles.

4.3.3 Correctness and complexity

Without going into details on this, it can be shown that the Simplex algorithm (with proper pivot rules) terminates for a given LP problem, and that it finds an optimal solution to the given problem if one exists. Otherwise it determines if the problem is either unbounded or infeasible. The complexity analysis will not be done properly here, but in short it is believed that there is no variant of the Simplex algorithm that has better worst-case time than exponential. However the average running time of the algorithm is rather good. Using n and m as a measure of the size of an LP problem we see that in general we must expect nm updates for each pivot, as we can expect all the equations to be affected by the pivot, and these contain nm variables in total. Now the number of pivots is the hard part to analyze thoroughly, but we can imagine a worst case scenario where all the vertices of the feasible domain are visited once. Since the number of vertices can be exponentially large in n, this could potentially be bad for the algorithm. In practice however an expected number of pivots is no more than $O(m)$, which is rather good. Although polynomial time algorithms for solving LP problems exists, they are often outperformed by the Simplex algorithm in practice.

4.4 Matrix notation, bases

As mentioned above it is also possible to give a formulation of both an LP problem and of the Simplex algorithm that looks like

$$min\{c^T x : Ax = b, x \geq 0\} \tag{18}$$

where the matrix A contains the identity matrix as a submatrix as in the example. This way of working with the problem will be used in the work with the Tree-Simplex algorithm.
The Simplex algorithm works by choosing which m of the $m + n$ variables are basic variables. Now let us split A into two parts: B containing the columns that correspond to the basic variables and N containing the columns corresponding to the non-basic variables. After possibly rearranging the columns of A and the rows of x, c, b we can then rewrite:

$$A = [B \; N] \tag{19}$$

$$x = \begin{bmatrix} x_B \\ x_N \end{bmatrix} \tag{20}$$

Our constraints are then

$$Ax = [B\ N] \begin{bmatrix} x_B \\ x_N \end{bmatrix} = Bx_B + Nx_n = b \tag{21}$$

We also partition the cost vector and the objective function:

$$c^T x = \begin{bmatrix} c_B \\ c_N \end{bmatrix}^T \begin{bmatrix} x_B \\ x_N \end{bmatrix} = c_B^T x_B + c_N^T x_n \tag{22}$$

Now the fact that the basic variables can be written as functions of the non-basic variables corresponds to the matrix B being invertible in (21), and we get:

$$x_B = B^{-1}b - B^{-1}Nx_n \tag{23}$$

Now the algorithm consist of keeping track of which variables are basic and which are non-basic, updating the values of x_B and the objective function, and pivoting on chosen variables. The computationally heavy part is solving the set of equations involving B, as B changes each time a pivot is performed. Note that mathematically this is exactly the same as the above approach to the Simplex algorithm.

4.5 Network flow

As promised we return to the problem of network flow, and will here look at how to solve such a problem. Letting A be the incidence matrix of the given digraph, the network flow problem is

$$min\{l^T x : 0 \leq x \leq c, Ax = -b\}$$

To solve this we will first look at the problem with the simplification that we are ignoring the capacities, i.e.

$$min\{l^T x : 0 \leq x, Ax = -b\}$$

What makes the network flow problem interesting is the special form of the matrix B which is used for the basic variables here. It can be shown that the matrix A has rank $m - 1$. We delete one row from A to obtain a new matrix A' and the corresponding entry from b to get b'. We call the node corresponding to the deleted row the *root node*. The following theorem contains the main idea for the network Simplex algorithm:

Theorem 4.5 *A square submatrix of A' is a basis if and only if its columns correspond to arcs forming a spanning tree in the network.*

For a proof refer to [5]

Now solving the set of equations $Bx_B = -b'$ actually is very simple. It corresponds to fulfilling the flow balance equations at each node of the graph, assuming all non-tree arcs have 0 flow. The following is an efficient method of calculating the flow along the spanning (basis) tree arcs:

Pick a leaf node. The flow along all arcs entering and leaving this node are known, except one. The supply of the node is also known. Calculate the flow along the last arc, and remove the node and this arc from the spanning tree, producing a smaller tree. Repeat the process until the tree is empty.

Of course the matrix B must have properties that allow us to solve the set of equations in the same way, and we can verify this by examining it closer. In fact we never have to do neither multiplications nor divisions, which speeds things up a bit in a computer.

Now doing a pivot in the tree simplex algorithm corresponds to choosing a non-basic arc to enter the basis, as usual. Adding this arc to the tree results in exactly one (undirected) cycle in the tree, and we then update the flows along only the arcs of this cycle as we increase the flow along the chosen non-basic arc. Which arc to leave the basis is determined by which arc has the least potential for change, as usual.

To add the capacity constraints along the arcs to our problem we use the trick of introducing some extra nodes and arcs to our graph. Assume we have vertices v_i, v_j with the arc a_{ij} having a capacity c_{ij}, cost l_{ij}. To enforce the capacity constraint on the flow along a_{ij} we can introduce an extra node v_k and replace a_{ij} by a_{ik} and a_{jk}. Here we let a_{ik} have cost $l_{ik} = d_{ij}$ and a_{jk} has cost $l_{jk} = 0$. In addition we increase the supply of v_j by c_{ij} and give the new node v_k a supply of $-c_{ij}$. Now we are again in the situation of a network without capacities, but one that corresponds to the original one with capacities. To recover the solution of the original network flow problem, simply take the $f(a_{ik})$ to be $f(a_{ij})$ of the original problem, ignoring the $f(a_{jk})$ arc. Since the cost $c_{jk} = 0$ the two problems will also have the same cost. The operation on the matrix A is less complicated:

Add a new column for the new arc a_{jk} (and just keep the a_{ij} as a_{ik}).

Add a new row expressing

$$f(a_{ik}) + f(a_{jk}) = c_{ij}$$

Subtract this row from the row

$$\cdots + f(a_{ik}) + \cdots = -b_j$$

to obtain

$$\cdots - f(a_{jk}) + \cdots = -b_j - c_{ij}$$

4.5.1 Multi-commodity flows

The situation we have looked at so far with network flows has had only one kind of flow commodity. In practice we often encounter problems where there is not *one* kind of commodity, but several. These problems are called multi-commodity flow problems, and are a rather straightforward generalization of the single-commodity flow problem, as we shall see now.

Instead of having a single commodity with sources and sinks, we now have several commodities, each with its own supply in each vertex of the network. The difference now is that it is not irrelevant which commodity ends up where, but the flow balance property must hold *individually for each commodity*. Let the n commodities be defined by the index set I, such that the functions $b_i(v) : V \to R$ defines the magnitude of commodity i at vertex v, and $f_i(a) : A \to R^+$ denotes the flow of commodity i along arc a. Then we require that

$$\sum_{a \in \delta_{out}(v)} f_i(a) = b_i(v) + \sum_{a \in \delta_{in}(v)} f_i(a) \;\; \forall i \in I, \;\; \forall v \in V$$

And to satisfy the capacity constraints, we must also require that

$$\sum_{i \in I} f_i(a) \leq c(a) \;\; \forall a \in A$$

as well as assuming non-negativity

$$f_i(a) \geq 0 \;\; \forall i \in I, a \in A$$

If we wish to minimize the total cost of our flow, which is given by

$$\sum_{a \in A} \left(l(a) \sum_{i \in I} f_i(a) \right)$$

we se that this problem is in fact an LP-problem, and we can thus solve it by the tools we have for these. The Tree-Simplex algorithm no longer works, but the general one does, and can thus be used for solving these problems rather efficiently, even for quite large networks with many commodities.

4.5.2 Maximum flows

Another approach to network flow is not finding the minimum cost flow satisfying some supplies/demands, but finding the maximum flow between a pair of vertices s, t. That is the flow with the greatest value. Let $G = (V, A)$ be a graph, with a length function $l : A \to R$ and a capacity function $c : A \to R^+$. We assume G has no negative cost directed cycles. Let $f : A \to R^+$ be a flow in G. We then construct the *residual graph* G_f *of* f as follows:
For each arc $a = uv$ with $f(a) > 0$ add a *residual arc* $a^{-1} = vu$ with length $l(a^{-1}) = -l(a)$ and capacity $c(a^{-1}) = f(a)$. Then reduce the capacity of a to $c(a) - f(a)$, and if the new capacity equals 0 remove a from G_f.

We are ready to formulate the *flow augmenting algorithm* of Ford and Fulkerson (or the *Successive Shortest Path algorithm* since we are choosing shortest paths in the algorithm). The algorithm is based on calculating flows g in the residual graph G_f, and then augment the existing flow f with the new flow, resulting in a flow $f + g$ with a greater value. This process is repeated until the residual graph no longer contains any $s - t$-paths, at which point we have a maximum value flow. We choose to augment the existing flow along the shortest path in the residual graph.

Initialization $f = 0$
Loop while true:
- $P \leftarrow$ shortest $s - t$-path in G_f
- if $l(P) = \infty$ STOP
- $\mu \leftarrow \min_{a \in P}\{c(a)\}$
- $g \leftarrow g(a) = \{\mu : a \in P \cap A,\ -\mu : a \in P \cap A^{-1},\ 0 : a \notin P\}$
- $f \leftarrow f + g$

It can be shown [1] that the algorithm terminates when the capacities $c(a)$ are rational, and that the resulting flow is maximal. But since we chose to augment the existing flow along a shortest path in each iteration of the algorithm, we can show even more: The algorithm computes a maximum flow with a minimal cost, and in fact each flow f during the course of the algorithm is minimum cost among all flows with the same value! This deserves a theorem:

Theorem 4.6 *The Ford-Fulkerson algorithm with shortest augmenting paths (or the Successive Shortest Path algorithm) computes for each step a flow which is minimum cost among all flows with the same value, and termi-*

nates with a maximum value flow if the capacities $c(a)$ are rational. If c is integer and bounded by M the algorithm has running time $O(M|E||A|)$.

For a proof refer to [1].

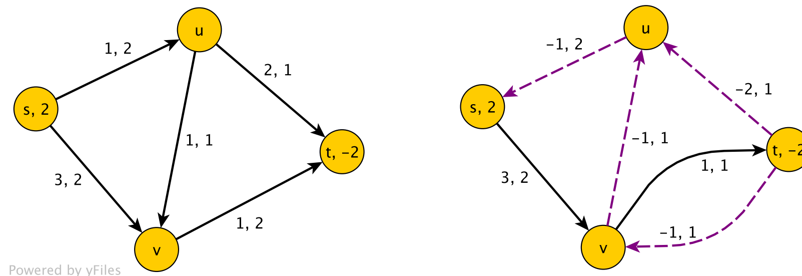

Figure 6: The same graph as in figure (2.1.4), and the residual graph corresponding to the flow in that example. Negative, or residual, arcs are dashed.

5 Analysis

In this section I will examine the problems posed in section 3. I will look at simplifications and special cases in an attempt to gain some insight into the problems, and to see which problems are solvable and how good solution methods we have for each of them.
This is done first for the static problems, and then I will go on and study the connection between the system optimum and the user equilibrium in the static setting, and actually present an algorithm that makes the system optimum a user equilibrium!
In the last part of this section I take a closer look at the dynamic system optimum problem. First by the rather intuitive approach of discretizing the time dimension of the dynamic network. And then by the less intuitive, but computationally faster and more compact, approach of chain decomposition of flow; flows that exist during certain time intervals.

5.1 Special cases and simplifications

When faced with a large problem in mathematics one often looks at special cases of the problem at hand to see if some useful results might be found for the special case, and then perhaps generalized back to the original problem. In this subsection I look at special cases of static networks in the system optimum and user equilibrium problems.

5.1.1 Number of commodities

In all original problems we had several commodities making up the total flow along each arc. These commodities represented different origin-destination pairs, and were indexed with the set I. It is clear that reducing the number of commodities greatly reduces the size of our problem, as the number of unknown variables is proportional to $|I|$. As we shall see having just a single commodity, together with some other simplifications, could allow for some more specialized algorithms to be used.

The first thing I will do here is to reduce the number of needed commodities to represent our problem. As stated we had one commodity for each origin-destination pair. But we can do with less! Treating all flow from the same source s but to different sinks t_i as one flow is possible. To do this we have to add a universal sink node T_s to our graph. For each sink t_i of the OD-pairs that have s as a source, we then add an arc t_iT_s with capacity equal to $b(t_i)$ and cost 0. This ensures that the correct amount of flow goes from s to each sink t_i without altering the flow in

any other way. Using this we can expect to reduce the number of commodities by a great amount. In a graph where all vertices are sources of flow to all other vertices, this trick would reduce the number of commodities to the square root of the original amount. And in a graph with only one source the number of commodities would be reduced to just one!

Alternatively we could treat all flow going to the same sink t as one flow

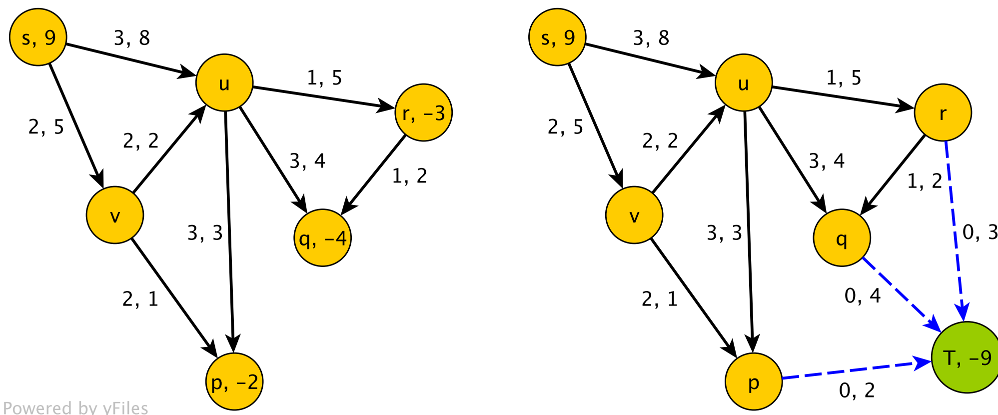

Figure 7: A network with several sinks for the same commodity is shown to the left. To the right is shown a network with the same routing problems, but with only one sink and thus only one commodity.

by adding a universal source in the same way as above. And in a specific problem we might chose which of the two approaches to use based on the number of sinks vs. sources. Let us summarize this:

Observation 5.1 *To reduce the number of commodities it is possible to treat all flow with a common source (or sink) as one flow.*

5.1.2 Special latency functions

Remember that the total travel time of all commuters was given as

$$\sum_{a \in A} x_a l_a(x_a)$$

where

$$x_a = \sum_i f_i(a)$$

This was also the objective function of the static system optimality problem, whereas the user equilibrium objective function was

$$\sum_{a \in A} \int_0^{x_a} l_a(x)dx$$

One of the simplifications we can do is restricting the latency functions on each arc of our graph to certain kinds. This can lead to problems that are solvable by more specialized and faster algorithms than what are needed for the general convex optimization case. The simplest choice for a latency function is the constant function

$$l_a(x) = l_a, l_a \in R^+$$

Inserting this expression in the static system optimality objective we obtain the following objective function

$$\sum_{a \in A} x_a l_a \tag{24}$$

which is linear. As we already stated the feasible domain of this problem is a polyhedron, due to the linear inequalities (5 - 7), and we recognize the current simplification to be an LP problem since the objective function is now also linear. We can then use the Simplex method to solve this efficiently, and this is a huge improvement compared to the rather slow approximation methods used for general convex optimization.

Note that assuming constant latency functions as exactly the same effect on the user equilibrium object (8), and the objective function for the two problems coincide in this case.

If we also assume that we have just a single commodity our problem becomes exactly the minimum cost network flow problem described in section 2, with capacities. We can then use the even faster Network Simplex algorithm to solve our problem, or alternatively the successive shortest path algorithm.

It is important that we do not drop the capacity constraints on the arcs in this case, as doing so would simplify the whole problem to finding the shortest path from the source to the sink, and then routing all flow along this path. Although very simple to solve, this problem would probably not reflect the real world problem very well. This also applies to the multi commodity case with constant latency functions. This problem, without capacities, would decompose to finding a shortest path for each OD-pair like in the single commodity case.

Observation 5.2 *In the case when our latency functions are constant the objective function of both the system optimum and the user equilibrium problems become linear, allowing the use of the Simplex algorithm. If we also have only a single commodity we can use the Network Simplex algorithm or the successive shortest path algorithm to find the solutions.*

A note on discontinuous latency functions might be needed. We defined the user equilibrium as a situation in which all utilized paths had equal cost, or latency. With latency functions of the kind $l_a(x) = l_a$ this might clearly be impossible to satisfy. If the network has only two parallel arcs a, b with $l_a < l_b$, and $c(a), c(b)$ finite, then we cannot really find a user equilibrium flow. What we could do in this situation is to think of the latency functions as increasing very rapidly at exactly the capacity of the edge, so adding just an infinitesimal flow over the capacity causes an infinitely expensive flow. This justifies the existence of the user equilibrium also in this graph.
Note also that the argument that minimizing the user equilibrium objective (8) corresponds to satisfying Wardrop's first principle is still valid, as it only required the latency functions to be non-decreasing.

5.1.3 Simplified networks

Another type of special cases we can look at is when the network graph itself has special properties. I will also look for simplifications that can be done without altering the solution, like the commodity number reduction above.
It is clear that any vertex v with no supply $b(v) = 0$ and with a single entering arc a and a single leaving arc b can be removed, joining the two arcs a, b to a new arc c. The latency is then summed together

$$l_c(x) = l_a(x) + l_b(x)$$

and the capacity is the minimum of the two

$$c(c) = \min\{c(a), c(b\}$$

We could also try joining two parallel arcs a, b to form one arc c with the same source and target as a and b. It is then clear that the capacity of the new arc would be the sum of the capacities of the two original, $c(c) = c(a) + c(b)$. Unfortunately finding the latency function of the new arc is a bit harder, and is actually influenced by whether we want to find the user equilibrium or the system optimum.

If we want the user equilibrium we should make the assumption that flow x_c is spread between the two arcs a and b in such a way that

$$l_a(x_a) = l_b(x_b)$$

For a given cost L this means

$$x_a = l_a^{-1}(L), x_b = l_b^{-1}(L)$$

Since

$$x_c = x_a + x_b$$

this implies

$$l_c^{-1}(L) = l_a^{-1}(L) + l_b^{-1}(L)$$

which finally gives the expression

$$l_c(x) = \left(l_a^{-1}(x) + l_b^{-1}(x)\right)^{-1}$$

The problem with this expression for l_c is that in many cases l_a^{-1} or l_b^{-1} might not be defined. Take for instance $l_a(x) = l_a$, a constant latency function. Then l_a^{-1} is not defined. But even though we found a way of working around this problem, which I'm sure we could, *finding* inverses can be a problem in itself. And on top of that many functions which could result from adding l_a^{-1} and l_b^{-1} don't even have expressible inverses! Assume for instance $l_a(x) = x, l_b(x) = x^2$, then $l_c(x) = \left(x + x^{\frac{1}{2}}\right)^{-1}$ which to the best of my knowledge is not expressible.

If we want the system optimum we should assume the flow x_c spread between a and b in a way that minimizes the total cost of using those two arcs. As shown below this is equivalent to

$$(x_a l_a(x_a))' = (x_b l_b(x_b))'$$

Pursuing the same idea as above we get from this that

$$\left[(x l_c(x))'\right]^{-1} = \left[(x l_a(x))'\right]^{-1} + \left[(x l_b(x))'\right]^{-1}$$

With the identification

$$f'^{-1} = \frac{1}{f'}$$

this is then

$$\frac{1}{(x l_c(x))'} = \frac{1}{(x l_a(x))'} + \frac{1}{(x l_b(x))'}$$

Since $xl_c(x) = 0$ when $x = 0$ we then finally get

$$l_c(x) = \frac{1}{x}\int_0^x \frac{1}{\frac{1}{(yl_a(y))'} + \frac{1}{(yl_b(y))'}} dy$$

or without the identification above

$$l_c(x) = \frac{1}{x}\int_0^x \left(\left[(yl_a(y))' \right]^{-1} + \left[(yl_b(y))' \right]^{-1} \right)^{-1} dy$$

Unfortunately this is even more impractical than in the user equilibrium case.

5.1.4 Alternative optimality criteria

The way we defined the two problems of user equilibrium and system optimality in the static network were very different. The user equilibrium problem was initially stated as finding a flow such that for origin-destination pair s, t all paths in use from s to t were of equal cost. This in turn led to a minimization problem with the objective function

$$\sum_{a \in A} \int_0^{x_a} l_a(y) dy$$

which then turned out to be a convex optimization problem.

For the system optimality problem the initial formulation was that of minimizing the total travel time of all traffic, given by

$$\sum_{a \in A} x_a l_a(x_a) \tag{25}$$

This problem also has a formulation similar to that of the user equilibrium, expressed locally on each $s-t$-path for each origin-destination pair.
Let P be an $s-t$-path and let x_P be the flow along this path. Then I claim the following:

Theorem 5.1 (System optimality condition) *Minimizing the system optimality objective function (25) is equivalent to requiring that*

$$\mathcal{L}_P = \frac{d}{dx_P}\left(\sum_{a \in P} x_a l_a(x_a) \right)$$

is equal for all used paths P and equal or less than for any unused path.

Proof. $\rightarrow$: Let P, Q be two $s - t$-paths with $\mathcal{L}_P < \mathcal{L}_Q$, and let A_P be the arcs in P not in Q and similarly for A_Q. Then shifting a flow of value δx from Q to P changes the value of (25) by

$$\sum_{a \in A_P} ((x_a + \delta x) l_a(x_a + \delta x) - x_a l_a(x_a)) - \sum_{a \in A_Q} (x_a l_a(x_a) - (x_a - \delta x) l_a(x_a - \delta_x))$$

But this expression is just the same as

$$\sum_{a \in A_P} \int_{x_a}^{x_a + \delta x} \frac{d}{dx}(x l_a(x)) dx - \sum_{a \in A_Q} \int_{x_a - \delta x}^{x_a} \frac{d}{dx}(x l_a(x)) dx$$

which is negative for small enough δx, since we have assume that $\mathcal{L}_P < \mathcal{L}_Q$.
$\leftarrow$: We have assume that each latency function $l_a(x)$ is convex and non-decreasing in the interval $0 \le x \le c(a)$. Then

$$(x l_a(x))' = l_a(x) + x l_a'(x)$$

is non-decreasing in the same interval, for each arc a. Thus each $\mathcal{L}_P$ is also non-decreasing as a function of x_P. Assume $\mathcal{L}_P$ is equal for each used path P and less than or equal for any unused path, for some flow x. Say $\mathcal{L}_P = L_x$. Then any other flow y for which the same holds, with $L_y < L_x$, must have $y_P < x_P$ for all paths P. But since the total value of the flow is $\sum_P y_P$ this means that the value of y is less than the value of x, so y cannot be feasible. Likewise we cannot have a flow y for which the incremental equality condition hold, but for which $L_y > L_x$. So let y be another flow with the same value as x satisfying the incremental equality condition, and for which $L_y = L_x$. It is clear that we can transform x to y by a series of flow shifts of values Δx from one path P used more by x to another path Q used more by y, such that $\mathcal{L}_P = \mathcal{L}_Q$. And then this does *not* change the value of the objective function. So x and y have exactly the same cost, and then so do all flows that satisfy the incremental equality condition. Since minimizing the objective function produces such a flow, we finally get that all flows satisfying the incremental equality condition also minimize the objective function! □

5.2 System optimal vs. user equilibrium

As an example of the relation between the system optimal and the user equilibrium solutions to the traffic assigment problem in the static case, consider the following example.

We have a network as shown in figure 8, where $l_a(x) = 1$, a constant latency function, and $l_b(x) = x$, a linear one. This means that the first arc has the same travel time regardless of how much flow is assigned to it, while the other arc has a travel time directly proportional to the flow. Now assume we are to route a flow of value 1 from s to t. The objective

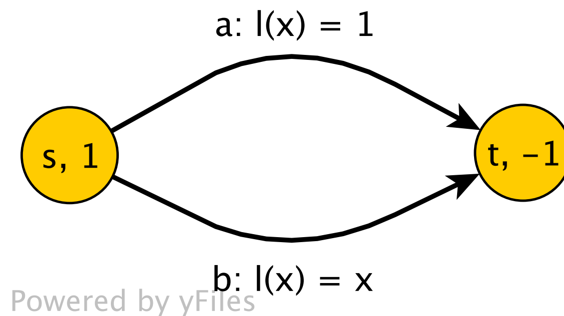

Figure 8: A small network with two arcs. Both arcs have unlimited capacity, and latencies are shown in the figure.

function for the system optimal solution is

$$\tau_s = x_a + x_b^2$$

Differentiating this and using the relation $x_a = 1 - x_b$ we get

$$\tau_s' = -1 + 2x_b$$

And we see that τ_s has a minimum at $x_b = \frac{1}{2}$, which gives $\tau_s = \frac{3}{4}$ While the total travel time is given by the same function here, the objective function for the user equilibrium is

$$\tau_u = x_a + \frac{1}{2}x_b^2$$

Differentiated this is

$$\tau_u' = -1 + x_b$$

And we see that τ_u has a minimum at $x_b = 1$, which gives $\tau_s = 1$. So the user equilibrium has in this case $\frac{4}{3}$ of the total travel time of the system optimal solution. None of the users accept that any other can travel faster than themselves, and thus everyone end up on the same road b,

worsening the travel time for all the other commuters, and ultimately themselves.

How bad great can the difference between the system optimal and user equilibrium solutions be? It has been shown [3] that for linear latency functions the relation is never greater than $\frac{4}{3}$. But with arbitrary latency functions, even convex ones, we can get arbitrarily large difference. Replacing $l_b(x) = x$ with $l_b(x) = x^n$ we can do the same analysis again.

$$\tau_s = x_a + x_b^{n+1}$$
$$\tau_s' = -1 + (n+1)x_b^n$$
$$x_{b_0} = \left(\frac{1}{n+1}\right)^{\frac{1}{n}}$$

And the optimal solution is $1 - x_{b_0} + x_{b_0}^{n+1} = 1 - x_{b_0}\left(1 - \frac{1}{n+1}\right)$ which gets arbitrarily small with a sufficiently large choice of n.
For the user equilibrium we get

$$\tau_u = x_a + \frac{1}{n+1}x_b^{n+1}$$
$$\tau_u' = -1 + x_b^n$$
$$x_{b_0} = 1$$

And the total travel time is again 1. Comparing with the system optimal we see that the quotient becomes arbitrarily large with sufficiently large n!

Theorem 5.2 *The quotient of the total travel time of the user equilibrium and the system optimal solution may be arbitrarily large for arbitrary latency functions.*

The other extreme case to consider is when all latency functions are constant. Then we see that the terms in the objective functions for the system optimal and user equilibrium problems coincide:

$$xl(x) = xl_0 = \int_0^x l_0 dx' = \int_0^x l(x')dx'$$

And we therefore get that the system optimal and the user equilibrium problems are exactly the same!

Theorem 5.3 *The user equilibrium and the system optimal solutions coincide if all latency functions are constant functions.*

Usually the latency functions are somewhere in between the two extremes, and it is meaningful to work with both cases.

How is it possible to force the user equilibrium and the system optimum to coincide? In the first example above we could guess at adding a *toll* of value $\frac{1}{2}$ to the road with the linear latency function, thus replacing $l_b(x) = x$ with $l_b(x) = x + \frac{1}{2}$. Here the constant factor is not actually time delay, but rather a tax you would have to pay to drive along that road. This should work because the two arcs would then have the same cost with the system optimal flow, and this should then also be a user equilibrium flow. Repeating the analysis we get

$$\tau_u = x_a + \frac{1}{2}x_b^2 + \frac{1}{2}x_b$$

$$\tau_u' = -1 + x_b + \frac{1}{2} = -\frac{1}{2} + x_b$$

And we see that the minimum is now at $x_b = \frac{1}{2}$, which is indeed the system optimal solution! What about the general case?

Remember Wardrop's characterization of a user equilibrium; that all paths from s to t that are in use have the same cost, and cost equal to or less than that of any unused path. In a system optimum this is not necessarily the case. I want to introduce taxes to some of the arcs in a given network such that the cost incurred by the travelers along arc a is $l(a) + T(a)$ where $T : A \to R^+$ is the *tax function*. Then there is a system optimal solution that minimizes the total travel cost f_s, and there is also a user equilibrium F_u corresponding to the new tax modified cost functions. Let it be absolutely clear that these taxes do not directly affect the latency along the arcs, but are only perceived by the travelers some generalized cost which they want to minimize together with travel time. *Then I claim that if these taxes are chosen appropriately we can force the system optimum f_s and the user equilibrium f_u to coincide!* To solve the problem of choosing an appropriate tax function I came up with the following:

Theorem 5.4 *Given an acyclic graph G, a length function $l : A \to R$ and vertices s with in-degree 0 and t with out-degree0 it is possible to find a function $T : A \to R^+$ such that all $s - t$-paths have equal length when considering the length function $l + T$, and such that the length of all these equal the length of the longest $s - t$-path when considering only l. This can be done in time $O(|A|)$.*

Proof. Since G is acyclic and s, t have in- and out-degrees 0 respectively we can find a topological ordering $v_0, v_1, \ldots, v_n$ of G where $v_0 = s$ and

$v_n = t$, and where all arcs are of the form $v_i v_j, i < j$. This can be done in time $O(|A|)$.
Let $\mathcal{P}_{v_i}$ denote all $v_i - t$-paths and let $L_{v_i} = \max\{l(P) : P \in \mathcal{P}_{v_i}$. I will prove by induction that for any vertex v_i we can find $T(a)$ such that

$$(l - T)(P) = L_{v_i} \ \forall P \in \mathcal{P}_{v_i}$$

for all outgoing arcs a from v_i. For $v_i = t$ the claim is trivial.
Now assume that the hypothesis holds for all $v_j, j > i$. Consider an outgoing arc $v_i v_j$ from v_i. Then the claim holds for v_j, and thus all $v_i - t$-paths passing through v_j are of equal $l - T$-length $L_{v_j} + l(a) + T(a)$ with $L_{v_j} + l(a) \leq L_{v_i}$. Now setting $T(a) = L_{v_i} - L_{v_j} - l(a)$ for all outgoing arcs a makes the hypothesis true for v_i.
Each arc is examined exactly once, so the time usage follows. □

Corollary 5.1 *The tax function T above also makes all $v_i - v_j$-paths equally long when considering the length function $l + T$.*

Proof. This follows immediately from the theorem, as any concatenation of a $v_i - v_j$-path and a $v_j - t$-path results in a $v_i - t$-path. Then since all $v_i - t$-paths and all $v_j - t$-paths have equal $l + T$-length all $v_i - v_j$-paths must also have equal $l + T$-length. □

Let us return to the problem of making the system optimum f_s of latency function l and the user equilibrium f_u of generalized cost function $l + T$ coincide in the graph G. We could try to accomplish this by considering the subgraph $G^{f_s} \subset G$ consisting of only those arcs used by f_s, with length function $l^{f_s}(a) = l_a(f_s(a))$. Since f_s is system optimal and $l_a(f(a)) \geq 0$ we can assume that G^{f_s} is acylic. Then we can find the tax function T such that all $s - t$-paths *in* G^{f_s} are of $l + T$-length equal to L_s. And adding this same tax function to the arcs in the original G we see that all $s - t$-paths in use by f_s have equal $l + T$-length under f_s! However some path not used by f_s might be shorter than L_s, and f_s then fails to be a user equilibrium for the whole graph G. Now let $\mathcal{P}^{L_s}$ denote all $s - t$-paths in G with l-length less than or equal to L_s. Now including all arcs in these paths in G^{f_s} might cause G^{f_s} to no longer be acyclic. Thus we just have to take extra care when calculating T for our graph G and system optimal flow f_s, to ensure T really causes f_s to be a user equilibrium under when considering $l + T$.

The following algorithm solves our problem of finding a tax function T for a graph G with a system optimal $s - t$-flow f_s considering latencies l such that f_s also becomes a user equilibrium when considering $l + T$. Let L_s be the length of the longest path in use by f_s.

Algorithm for finding optimal tolls first in G^{f_s} and then in all of G.

```
L_v ← −1 ∀v ∈ V
L_t ← 0
call findMaxDistance(s)
call unusedTolls()


function findMaxDistance(v)
  if L_v ≥ 0 return L_v
  L_v ← max{call findMaxDistance(u)+l_a(f_s(a)) : a = vu ∈ δ_out(v), f_s(a) > 0}
  T(a) ← L_v − L_u − l_a(f_s(a)), ∀a = vu ∈ δ_out(v), f_s(a) > 0}
  return L_v

function unusedTolls()
  d(v) ← L_s − L_v ∀v
  U ← V

  while U ≠ ∅
    v ← u ∈ U s.t. d(u) is minimal
    if d(v) > L_s BREAK
    U ← U \ {v}

    for a = vw ∈ δ_out(v)
      if L_w = −1 d(w) ← min{d(v) + l_a(f_s(a)), d(w)}
      else T(a) ← max{d(w) − d(v) − l_a(f_s(a)), 0}
```

Here the *findMaxDistance* function works within the acyclic subgraph G^{f_s} consisting of the arcs used by f_s and finds tolls such that all $s-t$-paths used by f_s become equally long. This is the part of the algorithm that corresponds to Theorem equalizer. The vertices v covered by f_s get $L_v > -1$. Then the *unusedTolls* function searches for shortest paths (Dijkstra-Prim based) with length less than or equal to L_s in the whole graph G, and assigns a correction toll to such paths each time it finds one, such that all these paths get cost at least L_s. Note that when a correction toll is assigned to an arc already in use by f_s, and thus with a tax already assigned to it, the new assignment equals the old, so no changes are done. So at the end all $s-t$-paths in use by f_s have $l+T$-length L_s,

and all other paths have $l + T$-length greater than or equal to L_s.

Note that we could actually have chosen *any* acyclic flow f in the algorithms above, not just the system optimal f_s, as the theorem (5.4) only requires the graph to be acyclic. So we see that we can make any flow a user equilibrium by proper taxation! But for me, of course, the system optimum makes the most sense.

What about the case when there is no longer just one commodity? We *can* also solve the more general multi commodity problem by calculating a set of tolls for each commodity. Then we would require knowledge of the origin and destination of each traveler in the network. Although this is currently impractical for real world traffic uses, it is an interesting theoretical result. And with the ever increasing importance of computer networks in our daily life, it might be possible to implement successfully in the future.
It could also be possible to find a way of calculating just one tax function for several commodities, such that the system optimum becomes a user equilibrium when including taxes. Alas I have not been able to solve this problem. I end this subsection with an example calculation of a static

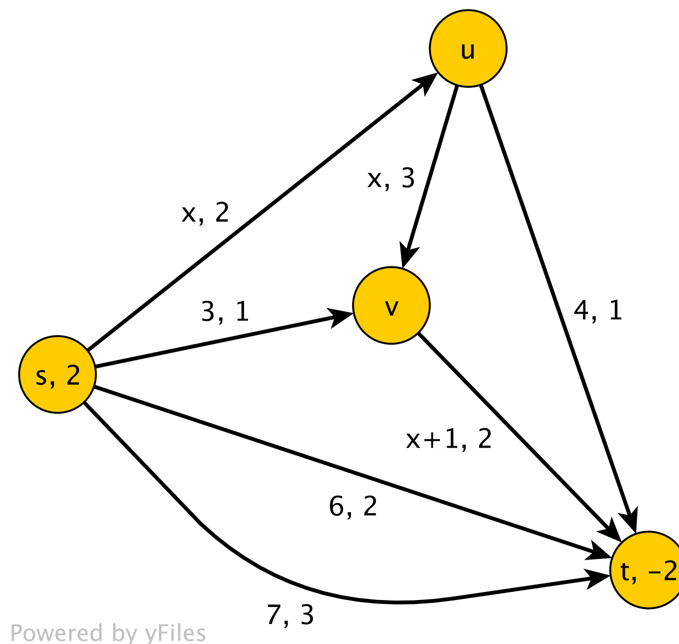

Figure 9: An example graph with one source/sink pair and with latencies and capacities as given on the arcs.

system optimal solution and corresponding taxes. Consider the graph shown in figure (5.2). We are to route a flow of value 2 from s to t. To find the system optimal solution I will use the alternative optimality criterion derived in the previous subsection. Let P_1 be the path $P_1 = s - u - t$, $P_2 = s - u - v - t$, $P_3 = s - v - t$, $P_4 = s - t$ along the arc with cost 6 and $P_5 = s - t$ along the arc with cost 7. Then $x_1 + x_2 + x_3 + x_4 + x_5 = 2$. And

$$\mathcal{L}_1 = 2(x_1 + x_2) + 4$$

$$\mathcal{L}_2 = 2(x_1 + 3x_2 + x_3) + 1$$

$$\mathcal{L}_3 = 2(x_2 + x_3) + 4$$

$$\mathcal{L}_4 = 6$$

$$\mathcal{L}_5 = 7$$

Let us assume that $x_5 = 0$. We will see later that this is a correct assumption. Then the equalities above constitute a linear set of equations. Solving this we get

$$x_1 = x_2 = x_3 = x_4 = \frac{1}{2}$$

The corresponding flow is shown in figure (5.2), with the latencies in parenthesis.

Using the latencies, or costs, in this graph, we run the taxing algorithm on the graph shown in figure (5.2), where the resulting taxes are shown as $+T$ on each arc. Adding these taxes to the original graph we finally get the graph in figure (5.2), where the latencies and taxes add up to form the new costs along each edge. We easily check that the flow $x_1 = x_2 = x_3 = x_4 = \frac{1}{2}$ makes all $s - t$-paths in use equally expensive when considering both latencies and taxes. And we also know that this flow reduces the total travel time, since it was the solution to the system optimality problem we started with.

Out of curiosity we can also find the user equilibrium in the original problem in the same way as we found the system optimum. This turns out to be

$$x_1 = x_3 = \frac{1}{3}, x_2 = \frac{4}{3}$$

The total cost of the user equilibrium flow is then $\frac{34}{3} = \frac{136}{12}$, whereas the system optimal flow has cost $\frac{39}{4} = \frac{117}{12}$.

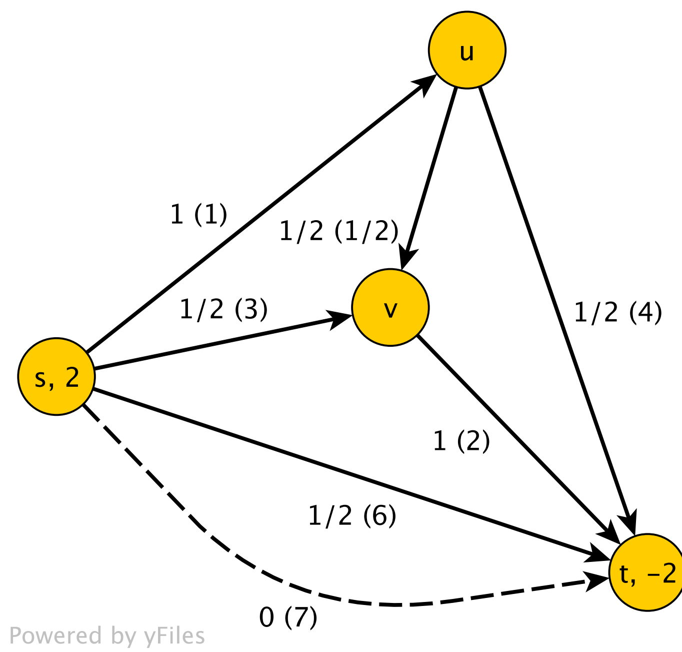

Figure 10: The system optimal flow of this graph, along with latencies corresponding to this particular flow f_s.

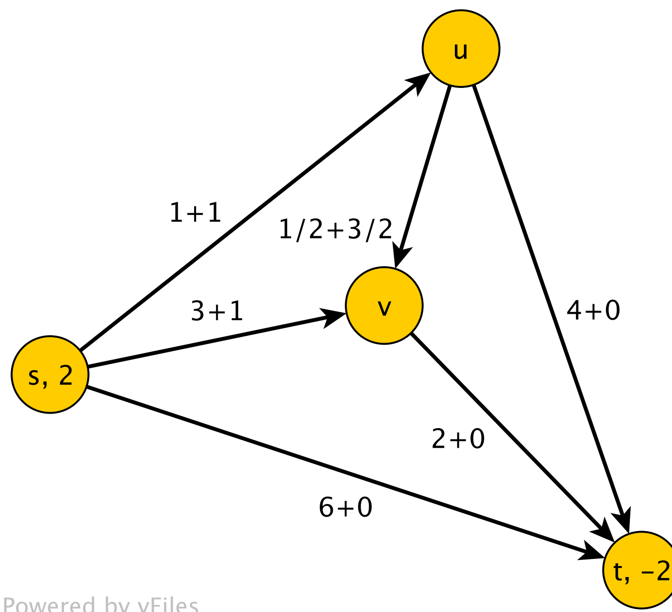

Figure 11: The graph G^{f_s} after computing taxes. Total arc costs $l + T$ are latencies + taxes on each arc.

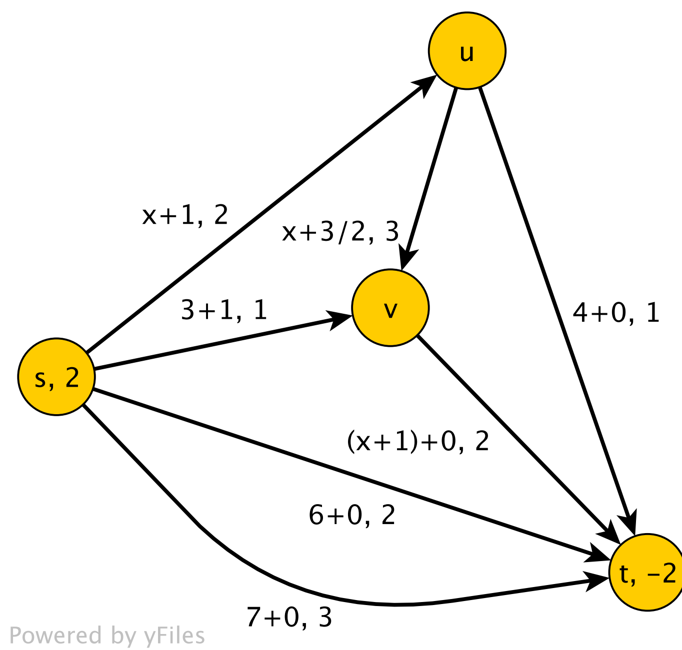

Figure 12: The original graph, with necessary taxes added along the arcs. The user equilibrium when considering latencies + taxes is now the same as the system optimum when considering only the latencies.

5.3 The dynamic case, simplified latency model

One interesting special case of network models was the one where there are absolute capacities on the arcs, and where the travel time (or cost) along each arc remains constant with regard to loading. For a vehicle traffic situation this seems a bit unrealistic, but for e.g. information flow in networks these assumptions might very well be acceptable. In fact, the *deterministic queue model* [2] exhibits exactly this behavior, under the assumption that there are no queues! In this case very much is known about our graph, and computing shortest paths, minimum cost flows, maximum flows etc. is all possible with well known polynomial time algorithms. Perhaps we might derive some useful results from this already well established area?

5.3.1 System optimal planning

A *dynamic system optimal solution* to a route and departure time planning problem is one which minimizes the *total cost* of all users, defined in section 3. Assume we bottleneck with an absolute capacity and desired through-flow greater than this capacity for a certain time interval. This could be a graph with just two nodes s, t and one arc $a = st$. We assume the deterministic queue model for this arc, that is

$$\frac{d}{dt} l_a(x_a, t) = \frac{x_a(t)}{c(a)} - 1 \tag{26}$$

if there is already a queue, or one is forming, or

$$l_a(x_a, t) = l_a \tag{27}$$

if there is no queue, and the inflow $x_a(t)$ not is great enough for one to form.
Let us also assume the departure deviation cost function $g(t) = 0$ to be zero, and the arrival deviation cost function to be $h(t) = \frac{1}{2}|t|$.
In this case a system optimal solution will consist of a constant inflow equal to the capacity of the bottleneck, such that the bottleneck is maximally utilized, but also such that no queues arise. The arrival time interval will, for a dynamic flow of value V, be $[-\frac{V}{2c(a)}, \frac{V}{2c(a)}]$, which causes the departure time interval to be $[-\frac{V}{2c(a)} - l_a, \frac{V}{2c(a)} - l_a]$.

The dynamic user equilibrium is again such that the total cost experienced by each traveller s equal.So the dynamic system optimum above certainly differs from the user equilibrium in that several of the travel

agents could have done better by choosing a departure time that would bring their arrival time closer to their desired arrival time $t = 0$. But in doing so they would have caused all later entrants to be delayed by the time it would have taken themselves to pass the bottleneck, and thus a queue would have arisen, increasing the total cost of all later travel agents. Here the dynamic flow that causes all travel agents to experience the same cost has a queue that starts to form at time $t = \frac{V}{2c(a)} - l_a$ and grows at a rate such that

$$l'_a(x_a, t) = -h'(l_a(x_a, t) + t)$$

until the time when $l_a(x_a, t) = 0$, at which the queue starts to shrink again at a rate such that

$$l'_a(x_a, t) = -h'(l_a(x_a, t) + t)$$

The dynamic inflow that satisfies this is

$$x_a(t) = \begin{cases} 2c(a) & : \quad -\frac{V}{2c(a)} - l_a \le t < -\frac{V}{4c(a)} - l_a \\ \frac{2}{3}c(a) & : \quad -\frac{V}{4c(a)} - l_a \le t < \frac{V}{2c(a)} - l_a \end{cases}$$

Comparing the two solutions we see that the inflow happens during exactly the same time interval, but in the user equilibrium case the inflow is great enough to cause a queue, so that none of the later travelers get a lower total cost than the very first ones.

In the rest of the treatment on dynamic flows I will, however, assume there are no queues. We see that adding a time dependent toll

$$\xi(t) = \frac{V}{2c(a)} - \frac{1}{2}|t + l_a|$$

at the entrance to the bottleneck would cause the dynamic system optimal flow to be a user equilibrium! And I expect it to be easy to find some tolls that can be used to make the system optimal flow a user equilibrium, also in the more general case. The crudest approach could just be to add tolls at the sink of a flow, equal to some constant minus the arrival deviation cost function for the flow arriving at that sink, and possibly also negate the departure deviation cost in the same way. *Thus the rest of this section will focus on constant latency function networks. No queues!*

5.3.2 Existence and uniqueness of the system optimal solution

The system optimal solution being the one that minimizes the total cost incurred by all users of the network, its existence is not hard to

prove. Since the total cost function is a continuous function into R with a bounded minimum this minimum is attained by some dynamic flow. In general it might be hard to say whether the system optimal solution is unique or not, but in the case of the constant latency simplification we can see that it is not necessarily unique, but convex: Let f_1 and f_2 be two system optimal solutions with no queues, and with total costs $C_1 = C_2 = C$. Both these are governed by (27), since there are no queues. Consider a convex combination $f_3 = \lambda f_1 + (1-\lambda) f_2, \lambda \in [0,1]$. Then along each arc in f_1 the flow is always less than or equal to the capacity of that arc, and the same for f_2, so a convex combination of flows along each arc will again never exceed the capacity of that arc. Thus there will not be any congestion in f_3 either. Now since the latency of each arc is constant with regard to flow, if $C_{a,i}$ is the total cost associated with using arc a in solution f_i, then $C_{a,3} = \lambda C_{a,1} + (1-\lambda) C_{a,2}$, and thus the total cost of f_3 is just $C_3 = \lambda C + (1-\lambda)C) = C$, so S_3 is also system optimal.

5.3.3 The time discrete graph

Finding flows with various properties is a well established area in graph theory. Optimal peak routing is one example of an application of this to real life problems, where we solve the problem of routing a given flow of traffic through a network of available roads. This is exactly the static system optimality problem we have studied already. But again this is just a snapshot of the traffic situation throughout the whole day, or throughout the time of the day where congestion is a problem. If we wish to route traffic not only through different routes, but also at different times, the problem becomes a bit harder.

One way of transforming this problem into an already well known and solvable problem is to discretize the time dimension of the problem and make a graph where the vertices of the new graph are the vertices of the original graph *at different times*. The arcs then go from a vertex to vertices that are reachable from that vertex in the original graph, but with a later time coordinate. Let the original graph G have vertices V and arcs A, and let the time discretization be $\mathcal{T} = \{t_n\}$. We assume that the latency functions l_a are all positive and integer, i.e. $l_a : A \to N$. Then the vertices of the augmented graph $\mathcal{G} = (V \times \mathcal{T}, A \times \mathcal{T})$ are (v, t_n) and there is an arc from (v, t_n) to (w, t_m) if there is an arc a from v to w and the travel time $l(a)$ is equal to $t_m - t_n$, and the cost of this arc is equal to the original travel time plus any fixed cost of that arc (tolls etc.), while the capacity is the same as that of the original arc a multiplied by the

time discretization unit Δt. There are also arcs from (v, t_n) to (v, t_{n+1}) with cost equal to $t_{n+1} - t_n$, representing waiting one time unit at node v.
In addition there are two nodes for each origin destination pair; one representing the origin at all time steps, the universal origin s_u, and one representing the destination at all time steps, the universal destination t_u. Let (s, t_n) be the original origin node in each time step, then there are arcs from s_u to (s, t_n) with cost equal to the departure deviation cost at time t_n, and similarly for the destinations. The universal origins and destination are sources and sinks respectively with supply equal to the total amount of traffic that must pass from the origin to the destination in the original graph. An example of this construction is shown in figures (5.3.7) - the original graph - and (5.3.7) - the augmented graph.

My claim is that finding a system optimal travel plan is equivalent to finding a minimum cost flow through the augmented graph, that satisfies the source and sink constraints at the origins and destinations. This can be seen by letting the time discretization steps go towards 0. Then the total cost of the multi-commodity flow in the augmented graph approaches the dynamic system optimality objective function, as the sums over all time steps approach the integrals. And since we minimize the cost of the multi-commodity flow in the augmented graph, we also minimize the dynamic system optimality objective function.

Proposition 5.1 *A minimum cost (multi-commodity) flow in the augmented graph is an approximate solution to the minimum cost (multi-commodity) dynamic flow in the dynamic network.*

We then see immediately that this model has good flexibility in several aspects: varying capacity with time, using any kinds of departure and arrival specific cost, time varying toll functions.

5.3.4 System optimal planning by use of the augmented graph

The system optimal solution is one which minimizes the total cost incurred by all travel agents. Since the cost of following a path from s_u via (s, t_n) and (t, t_m) to t_u in the augmented graph is the same as the cost that a travel agent departing at time t_n and arriving at time t_m incurs, then a minimum cost flow through the augmented graph must be the same as a system optimal solution in the continuous case. This is with the reservation that all time dependent cost functions are piecewise constant in the time discrete graph, but may be continuous in the original

problem. But by choosing a fine enough discretization we can get arbitrarily close to the original continuous cost functions. This will, however, lead to a graph with an arbitrarily huge number of nodes, something we do not want, and thus choosing a fine enough but also not too fine discretization will be important.
Thus the problem of finding a system optimal flow in a network where we assume constant travel time and absolute capacities on the arcs can be approximated by finding a minimum cost flow in the augmented graph. This results in a dynamic flow that is really a different static flow for different time steps. But since it is a feasible flow for each of these, the concatenation of each of these static flows result in a feasible dynamic flow. So by finding a minimum cost static flow that satisfies the constraints of the augmented graph, we have really found a dynamic flow that satisfies the dynamic constraints (14 - 17).
It is also possible to solve the multi-commodity minimum cost problem by use of linear programming, and thus we can even approximate system optimal solutions to networks with several origin destination pairs!

5.3.5 Variable preferred arrival time

Something the first outline of our time discrete graph did not allow for was the option of having different preferred arrival times (or departure times). I will give a solution to this for the arrival time case. The departure one is treated similarly.
By introducing some extra nodes and altering the arcs that enter the universal sink we get a graph where different choices of paths the last two arcs correspond to different arrival time preferences. Let t be the original destination node and (t, t_n) its time discretization as usual. Then instead of having arcs from each of the (t, t_n) to t_u we introduce an extra set of nodes, δ_{h_k}, between these, such that each of these extra nodes correspond to a different arrival deviation cost function h_k. From (t, t_n) to δ_{h_k} there are arcs with unlimited capacity and cost equal to $h_k(t_n)$, i.e. the cost of arriving at time t_n with cost function h_k. And from each of the δ_{h_k} to t_u there are arcs with capacity equal to the amount of travel agents having h_k as their arrival time cost function.
We see immediately that this allows for different preferences in arrival time, as this is just the same as translating the cost function along the time axis. But it also allows for any different kinds of cost functions!

5.3.6 Properties of the augmented graph

As the augmented graph is a special construction, we expect it to have some properties that might be of use when solving the minimum cost flow problem in it.

Proposition 5.2 *The augmented graph has no directed cycles.*

Proof. All nodes of the graph are one of the following types:

- Universal origin/destination. These only have arcs exiting/entering, and can thus not be part of any cycle.
- Departure/arrival cost nodes. These only have arcs entering from/exiting to the universal origin/destination, and can therefore not be part of any cycles either, as such a cycle then would have to include the universal origin/destination.
- Time discretization nodes. Since all arcs representing time discretizations of original arcs have a positive travel time, it is impossible to depart from any time discretization node and return to this node within the same time step.

□

In fact the graph is not only acyclic, but in the case when the latency and capacity functions are constant over time, which is the case we are the most interested in, it contains a plethora of equal paths from say (s, t_n) to (t, t_m), and then also from (s, t_{n+k}) to (t, t_{m+k}) with $k \in Z$. We might expect that if one of these is used, then so will many of the other equal paths be. Unfortunately this information is not utilized by the Simplex algorithms. In the algorithm in the last part of this section I do utilize this information, to find a much faster solution method than this augmented graph method. This comes at the cost of the loss of flexibility in time varying latencies and capacities.

5.3.7 Examples of the augmented graph

I will demonstrate how to create and use the augmented graph by presenting a small example. The network we will use is rather simple, consisting only of an origin vertex and a destination vertex, and two different arcs between these. One of the arcs has a shorter length than the other, and also greater capacity, but as the desired flow through the

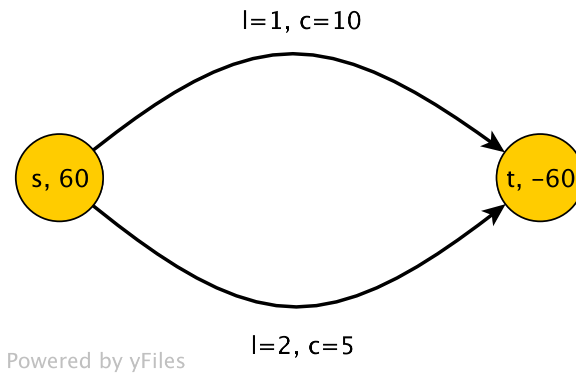

Figure 13: A sample network we discretize and do some calculations with.

graph exceeds the time unit capacity of the shorter arc we expect that both arcs will be used. The original graph is drawn in figure 5.3.7.

In the first example I will show the first approach to our time discrete graph, that is allowing only one arrival cost function. The time discretization is done here by choosing $\Delta t = 1$, and by using 6 time steps $t_n, i = -1, \ldots, 4$. The arrival cost function $h(t)$ used here is piecewise linear, with $h(t) = -0.5t, t < 3$ and $h(t) = 2t, t > 3$, and the departure cost is $g(t) = 0$. Thus the costs along the arcs from the universal origin s_u to the time discretizations (s, t_n) of the origin node are all zero, and the capacities are infinite, and the arcs from the time discretizations of the destination node (t, t_n) to the universal destination t_u have costs $h(t_n)$ and infinite capacities. And also, the arcs from (s, t_n) to (t, t_{n+1}) have cost 1 and capacity 10, and the arcs from (s, t_n) to (t, t_{n+2}) have costs 2 and capacity 5. The resulting graph is shown in figure 5.3.7.

Solving the problem of a minimum cost flow of value 60 we end up with fully utilizing all the arcs marked in blue and cyan (dashed) colors, and we also see that we could have used the green arcs (dotted) at the expense of the cyan ones, something which would not make any difference to the total cost. Interpreting this we see that the faster arc will be used over a greater time interval than the slower one, but that both of them will indeed be used. Adding together the cost of the different components of the flow, we get a total cost of 122.5. Now the way we have defined the cost along the arcs that correspond to arrival costs, we have assumed everyone arrives at exactly the same time, i.e. we have chosen $l((t, t_n)t_u) = h(t_n)$. This is of course slightly wrong. We could

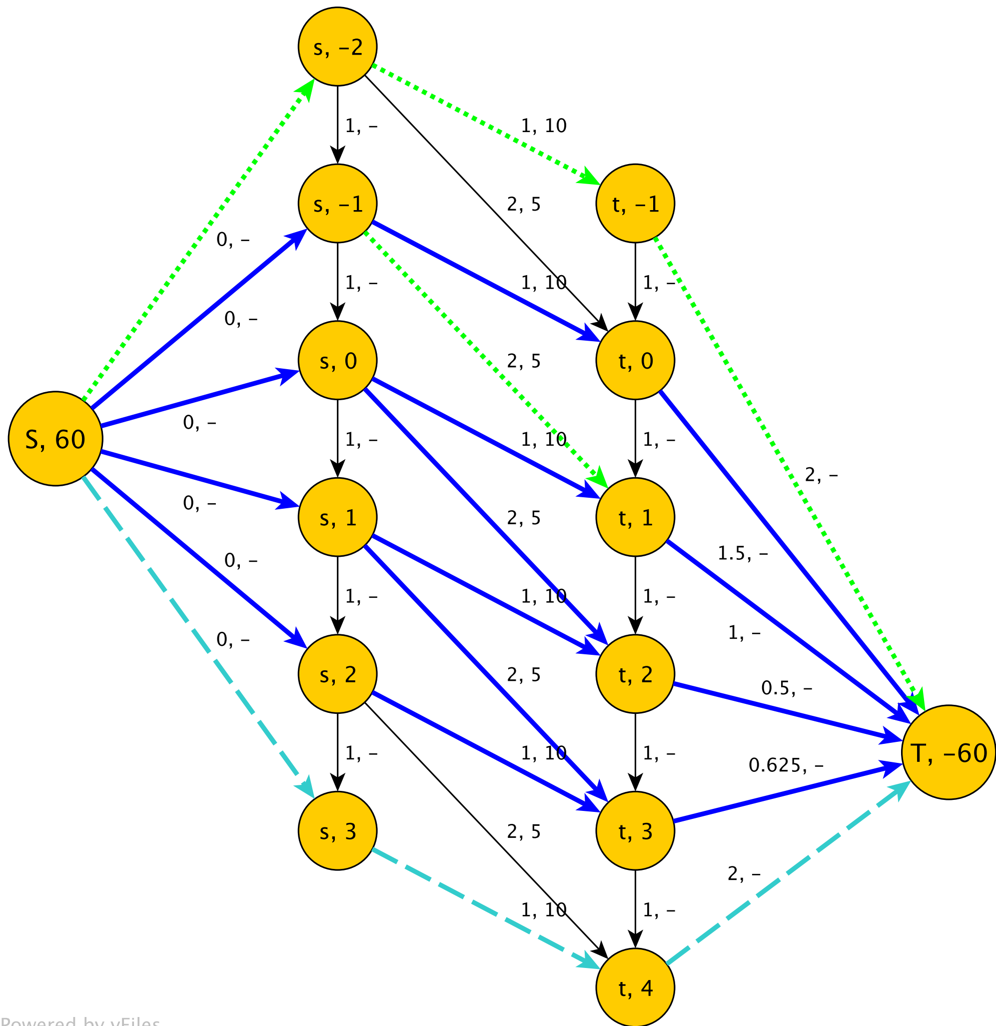

Figure 14: An augmentation of the sample graph, with time discretization unit 1. A minimum cost flow is shown in thick blue and cyan (dashed) lines. The green (dotted) arcs have the same cost as the cyan, and could have been used as well.

instead have used the average cost of the flow with average arrival time t_n, and this will henceforth be used. Now since h is piecewise linear this actually gives the same cost everywhere except where h has a break point. In this example h only has one break point, namely at $t = 3$, and the cost $l((t,3)t_u)$ here becomes $\frac{5}{8}$ instead. Recalculating the total cost then gives $131\frac{7}{8}$, which is indeed the cost of this flow when viewed continuously as well.

Solving analytically we get that the faster arc will be in full use in the time interval that causes arrivals in the interval $(-\frac{26}{30}, \frac{119}{30})$ and the slower arc in the time interval that causes arrivals in the interval $(\frac{34}{30}, \frac{104}{30})$. Integrating the flow cost terms here we get a total cost of $123\frac{5}{6}$. We see that we here have a better solution that the one we got by using the augmented graph, which gave a solution that was roughly 6.5% more expensive. But remember that in the solution of the minimum cost flow problem we had several equally expensive choices for routing the most expensive part of the flow. If we had chosen do divert some flow to each of them, we should expect to get a better result than we did. This is again because the cost along the arrival cost arcs are based on arrival at the mean time of the time interval represented by that arc, and if we had only used a bit of that interval, we could have chosen the cheaper part of it, thus obtaining a lower cost than our graph model shows.

Pursuing this idea I tried moving the desired arrival time from $t = 3$ to $t = 2.5$, which then changes only the costs along the last arcs in the graph. Solving the minimum cost flow problem here gives a flow with value 60 and cost only 125, or less than 1% more than the optimal cost! This solution is shown in figure 5.3.7. We see that here we have no alternative choices of arcs that would give the same total cost. Now the optimal solution would still use some of the unused arcs to a very small extent, at the expense of the most expensive choices here, but I think we are pretty close, and with a rather simple discretization. We see, however, that we could also have bad luck when choosing our discretization. An upper bound on the error of the cost obtained could be nice.

Now for the case with different arrival cost functions. Assume three different arrival cost functions h_1, h_2, h_3 with h_2 as the cost function h used above, with desired arrival time $t = 3$, and with $h_1(t) = h_2(t+1)$ and $h_3(t) = h_2(t-1)$. The amount of traffic that has each cost function is 20, 30 and 10 respectively. The augmented graph is set up above, except for the last arcs. These now go to the three different vertices for the different arrival cost functions, and then there are arcs from each of these to the universal destination vertex. The graph is shown in

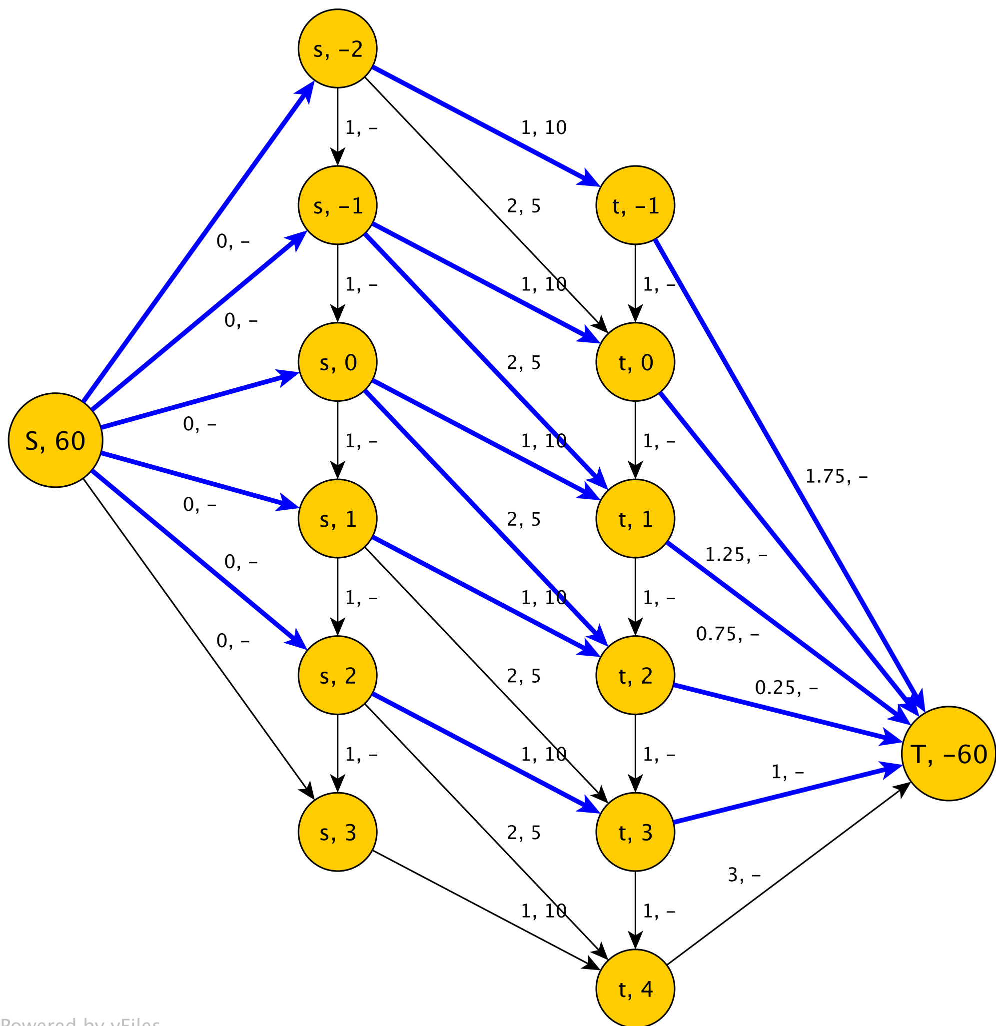

Figure 15: The same graph augmented, but with the time discretization translated $\frac{1}{2}$ time unit. This results in a much cheaper minimum cost flow, in thick blue.

figure 5.3.7. Again the solution to the minimum cost flow problem with a flow of value 60 is shown with the arcs in blue being fully utilized. The solution has a total cost of 105.

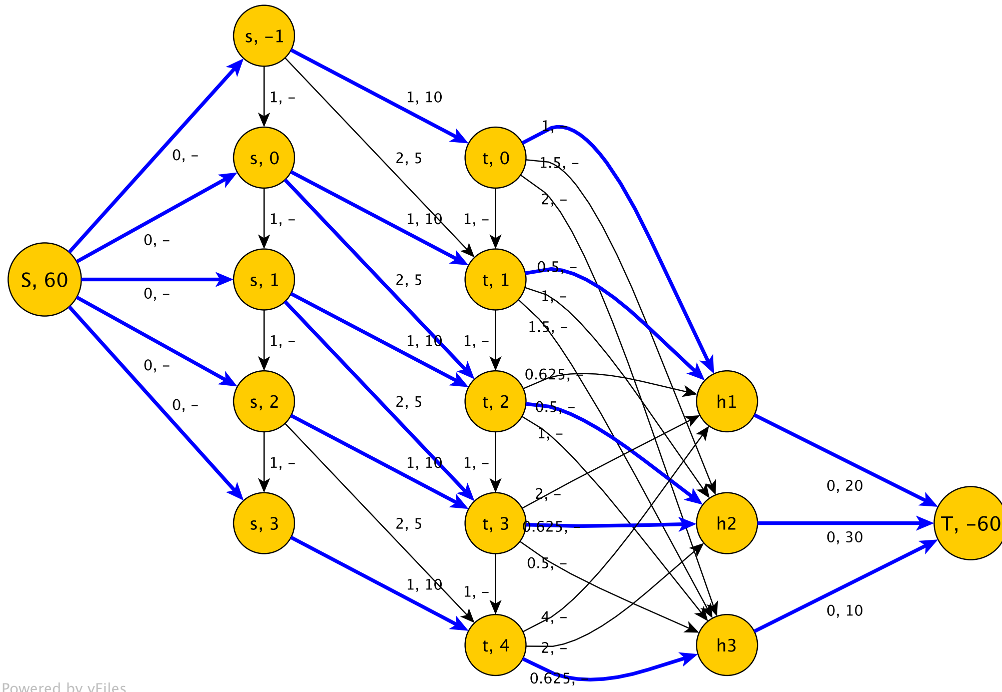

Figure 16: Again the same graph, but with three different arrival cost functions. The thick blue lines indicate again the minimum cost flow.

5.3.8 Multi-commodity planning

Having dealt with the case of routing a flow from an origin to a destination in a network across different times, we see that there's a fairly obvious generalization we should look at, and which we have already mentioned: Dynamic networks with several origin/destination pairs.
How to construct the augmented graph in this case is rather straightforward: Just add universal origins and destinations (or sinks and sources), possibly with corresponding departure or arrival cost nodes, for each of the sources and sinks in the original network. Thus we get several universal sources and sinks that will be used as the sources and sinks of our new multi-commodity flow problem in the time discrete graph.

To solve the multi-commodity flow problem we need only solve the LP problem we get from formulating the graph's flow constraints and costs, like in the static multi commodity case.
Thus we see that practically any dynamic system optimality problem can be solved this way, but possibly resulting in a large graph for which the multi-commodity flow problem takes long time to solve. Both the number of unknowns and constraints is proportional to both the number of commodities and vertices. In the brief time analysis of the Simplex algorithm we used the dimensions n, m of the constraint matrix as a measure of problem size. Here $n, m \in O(|\mathcal{T}||I|)$. We get slightly better results for the single commodity case, but the time discretization can still be a problem if the network already is large.

5.4 Chain decomposable flows

As we saw in the previous section we could use a time augmented graph to solve the dynamic system optimal problem, assuming all latency functions were constant with regard to flow. Although problems are solvable by this method, the size of the augmented graph can be a problem, as the number of arcs and nodes are proportional to the number of time discretization steps. Recent work on dynamic flows [4] has been a breakthrough in this area, and Hoppe has come up with an algorithm to solve several dynamic flow problems in true polynomial time, regardless of the time discretization chosen! The problems solved in this fashion are

- The dynamic flow problem: Finding a dynamic flow from one source to one sink satisfying a given supply within a certain time interval.
- Quickest dynamic flow problem: Finding a dynamic flow in as short a time as possible. Uses the above algorithm with binary search for smallest possible time interval.
- Lexicographical maximum dynamic flow problem: Maximizing the flow between source/sink pairs in a prescribed order of importance during a given time interval.
- The dynamic transshipment problem: Finding a dynamic flow satisfying given supplies for several source/sink pairs within a certain time interval. Modifies the network so the problem is solvable by the above algorithm.

- Quickest dynamic transshipment problem: Finding a dynamic transshipment flow in as short a time as possible. Uses the above algorithm with binary search for smallest possible time interval.

The algorithms devised by Hoppe build on the rather simple successive shortest path algorithm for finding a maximum $s-t$-flow in a graph with capacities, but include considerations regarding the time discretization. Instead of only assigning flow along the computed shortest paths of the augmented graph, the flows are assigned during a maximal time interval; starting from the source at time step 0 and ending so that the last part of the flow reaches the sink in the terminal time step T. The work in [4] assumes an integer time discretization of the original graph, and thus integer latency functions, as this makes some theoretical results easier. I try to work with arbitrary positive latency functions.

5.4.1 System optimal solution with chain flows

Let us now reconsider the dynamic system optimal planning in the constant latency setting. To summarize we have a network consisting of a graph G with several origin-destination vertex pairs (s_i, t_i), each with a supply/demand. Each arc a has a constant latency function $l_a(x) = l_a$ and a capacity $c(a), c : A \to Q^+$. Each commodity also has a departure cost function $g_i(t)$ and and arrival cost function $h_i(t)$ applying when flow leaves the source or enters the sink.
Looking back at the general form of the system optimal planning problem (11 - 17) we see that the constant latency functions simplify the objective function slightly and the constraints massively. This is because we no longer have the complicated relation between inflow and outflow to each arc. We denote the inflow to arc a at time t by $x_a(t)$, and the outflow at time $t + l_a$ is then equal to $x_a(t)$.
We still have the objective function split in two parts. This really is just like in the augmented graph above, but from a slightly different viewpoint, as we no longer make a time discretization. The total travel time is now:

$$\sum_{a \in A} \int_0^T x_a(t) l_a dt \tag{28}$$

And the total departure and arrival time deviation cost is:

$$\sum_i \int_0^T b_i(s_i, t) g_i(t) - b_i(t_i, t) h_i(t) dt \tag{29}$$

The constraints apply just as before, but with the much simpler relation between inflows and outflows.

If we choose to optimize with regards to the travel time only our problem really simplifies. Since the total travel time is also the integral over all traffic of the travel time experienced by each travel agent, it is clear that minimizing the travel time of each infinitesimal piece of flow minimizes the total travel time. But this is just the same as calculating a shortest path through the graph, and routing traffic only along this path until all the supply is satisfied! Of course this might lead to a really long time during which traffic flows in the graph, and commuters might be almost arbitrarily late (or early) for work. Thus minimizing the total travel time alone makes little sense in the dynamic setting. On the other hand optimizing only with regard to the departure and arrival costs we try to find a cheapest (and thus shortest) possible set of time intervals in which flow leaves the sources and enters the sinks. This is close to the quickest dynamic flow problem for one origin/destination pair, or the quickest transshipment problem with several pairs [4]. And as we remember these problems are now solvable in polynomial time with chain decomposition! But since travel time is not regarded at all, disproportionally slow routes may be utilized, leading to a solution that is not system optimal in the full sense. It is again not good enough to consider only one part of the objective function. However using the ideas pursued by Hoppe we might hope to devise a faster way of solving the full dynamic system optimality problem by eliminating the use of the time-augmented graph!

We still assume that all latency functions are constant, and that we have capacity constraints on the arcs of our graph. We also assume that all traffic shares the same convex departure and arrival cost function $g(t), h(t)$. I'll start with the case of a single commodity, or dynamic $s-t$-flow. What I propose is that the following Dynamic System Optimality algorithm solves the system optimality problem with these assumptions. This algorithm is based on the Successive Shortest Path algorithm in section 4.5.2.
Let us first define a *chain* (flow):

Definition 5.1 *A chain F_i in network G with source s and sink t is a quadruple $F_i = (P_i, c_i, t_i^s, t_i^e)$. Here P_i is an $s-t$-path in the undirected underlying graph of G. c_i is a real number denoting the constant amount of flow along P_i, such that F_i sends flow of value c_i along arcs included in P_i with their positive direction, and canceling flow of value $-c_i$ along arcs included in P_i with their negative direction. And finally there are two*

reals denoting the time interval in which flow runs along P_i; the starting time t_i^s at source s and the ending time t_i^e at sink t.

Let $F = \{F_i\}$ be a set of chains, and let $l_i = l(P_i)$ be the length of P_i for compactness. Then the total dynamic flow $\mathcal{F} : A \times R \to R$ induced by F is the sum of all chains in F, and the total amount of flow

$$f_{tot} = \sum_i c_i(t_i^e - t_i^s - l_i)$$

We also denote the static flow induced by the k first elements of F by f_k. Let f_{demand} be the total amount of flow needed from s to t. Let C_{max} be the maximum individual cost associated with the current flow, and let $C_{max}^{old}, C_{max}^{new}$ be lower and upper bounds on C_{max}. Let G_{f_i} be the residual graph associated with G and the static flow f_i. Let also

$$H_i(t) = l_i + g(t - l_i) + h(t)$$

$H_i(\tau)$ is then the total cost incurred by a traveler using path P_i at a time such that she arrives at t at time τ. We assume that each $H_i(t)$ has a minimum for some t.

Dynamic System Optimum algorithm

$C^{old}_{max}, C^{new}_{max} \leftarrow 0$
$F \leftarrow \emptyset$
$G'_0 \leftarrow G$
$i \leftarrow 0$

while $f_{tot} < f_{demand}$
  $i \leftarrow i + 1$
  $P_i \leftarrow$ shortest path in G'_{i-1}
  if $!\exists P_i$
    $C^{new}_{max} \leftarrow \infty$
    BREAK
  else
    $c_i \leftarrow$ capacity of P_i
    $C^{new}_{max}\min\{H_i(t)\}$
    $t^s_i \leftarrow \min\{t : H_i(t + l_i) = C^{new}_{max}\}$
    $t^e_i \leftarrow \max\{t : H_i(t) = C^{new}_{max}\}$

  for $j = 1, \ldots, i - 1$
    $t^s_j \leftarrow \min\{t : H_j(t + l_j) = C^{new}_{max}\}$
    $t^e_j \leftarrow \max\{t : H_j(t) = C^{new}_{max}\}$
  if $f_{tot} \geq f_{demand}$
    BREAK
  else
    $F \leftarrow F \cup \{(P_i, c_i, t^s_i, t^e_i)\}$
    $G_{f_i} \leftarrow G_{f_{i-1}}$ updated with flow c_i along P_i
    $C^{old}_{max} \leftarrow C^{new}_{max}$

find $C_{max} \in [C^{old}_{max}, C^{new}_{max}]$ s.t. $f_{tot} = f_{demand}$
  when t^s_i, t^e_i is updated accordingly to C_{max}
F induces a system optimal dynamic flow $\mathcal{F}$

The idea behind this algorithm is simple enough: Find a shortest path in the residual graph and determine the minimum cost for anyone using it, and the time at which this minimum is attained. Use this path initially at only the minimum cost time. Then increase the time interval for which this path is in use, until some other path (possible with negative arcs, meaning a modification to already existing flow) becomes equally expensive at its minimum cost time. Then increase the time interval for

which both these paths are in use, until a third path becomes equally expensive. And so on.
The chains are chosen and updated such that no infinitesimal chain in use is more expensive than any not in use, and such that no infinitesimal chain in use has cost greater than C_{max}^{new}. In addition all infinitesimal chains with cost less than or equal to C_{max}^{old} is in use at the start of each loop iteration, so the C_{max} that gives a feasible total flow is always in the interval $[C_{max}^{old}, C_{max}^{new}$.
The total cost of the dynamic flow is the same as the sum of the costs of each of the chains. So a chain with flow along negative arcs takes into account the modification done to already existing static flow, and these chains are then chosen in such a way that the total cost is always the lowest. Note also that the total cost of dynamic flow $\mathcal{F}$ induced by F can be much more compactly given with the chain representation. The total cost of chain F_i is

$$\int_{t_i^s+l_i}^{t_i^e} H_i(t)dt$$

$$= \int_{t_i^s}^{t_i^e-l_i} g(t)dt + \int_{t_i^s+l_i}^{t_i^e} h(t)dt + (t_i^e - t_i^s - l_i)l_i$$

Notice that which arcs the chains use are of no direct importance, as the total cost is fully determined by outflow from the source and inflow to the sink.

Before proving correctness of the algorithm, let's look at an example application. The graph we consider is shown in figure (17). Here the departure and arrival deviation cost functions are

$$g(t) = -\frac{1}{2}t$$

$$h(t) = \frac{1}{2}t + |t|$$

The first chain found by the algorithm follows path $P_1 = (s - u - v - t)$ which has total latency $l_1 = 3$ and capacity $c_1 = 3$. We see that $H_1(t) = \frac{9}{2} + |t|$ which is minimal at $t = 0$. Thus $F_1 = ((s - u - v - t), 3, -3, 0)$ is added to F at the end of the first loop iteration, with $C_{max}^{old} = \frac{9}{2}$.
The residual graph G_1' resulting from f_1 is shown in figure (18). The shortest path in G_1' is $P_2 = (s - v - u - t)$ with total latency $l_2 = 5$ and capacity $c_2 = 1$. This gives $H_2(t) = \frac{15}{2} + |t|$ which is minimal at $t = 0$. Thus C_{max}^{new} is raised to $\frac{15}{2}$, which increases the time interval of use of F_1 to $t_1^s = -6, t_1^e = 3$. This gives a total flow of $3(3 - (-6) - 3) = 18 < 26$,

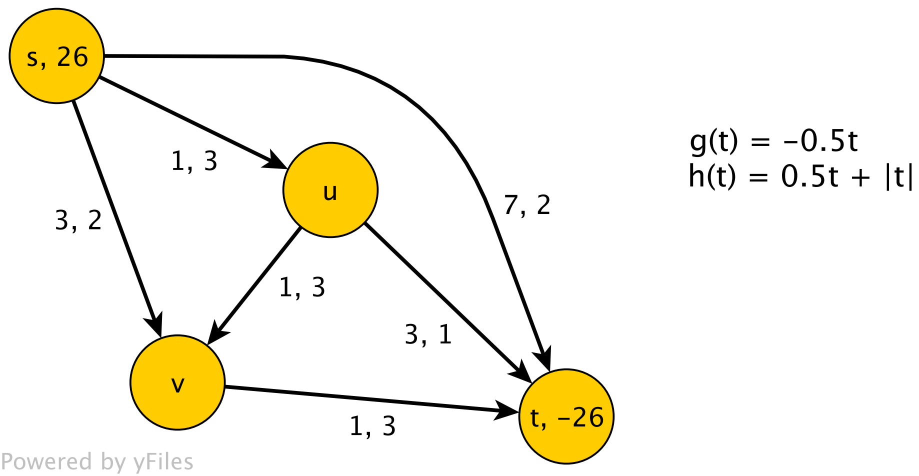

Figure 17: An example constant latency graph, with latencies and capacities shown on each edge, and the total supply shown in the terminal vertices.

so $F_2 = ((s - v - u - t), 1, -5, 0)$ is added to F at the end of the second iteration, with $C^{old}_{max} = \frac{15}{2}$

In the third iteration the only $s - t$-path in G'_2 is the slow arc st. We get $P_3 = (s - t)$ with latency $l_3 = 7$ and capacity $c_3 = 2$. This gives $H_3(t) = \frac{21}{2} + |t|$ which is minimal at $t = 0$ with minimum value $\frac{21}{2}$. Raising C^{new}_{max} to $\frac{21}{2}$ we increase the intervals of use of F_1, F_2 to $t^s_1 = -9, t^e_1 = 6$ and $t^s_2 = -8, t^e_2 = 3$ which gives a total flow of $3(6-(-9)-3)+1(3-(-8)-5) = 42 > 26$. Thus we break the loop and search for a $C_{max} \in [\frac{15}{2}, \frac{21}{2}]$ that gives the correct amount of total flow. Since we have piecewise linear cost functions in this example we quickly find that $C_{max} = \frac{17}{2}$ gives the intervals $t^s_1 = -7, t^e_1 = 4$ and $t^s_2 = -6, t^e_2 = 1$ resulting in exactly 26 total flow.

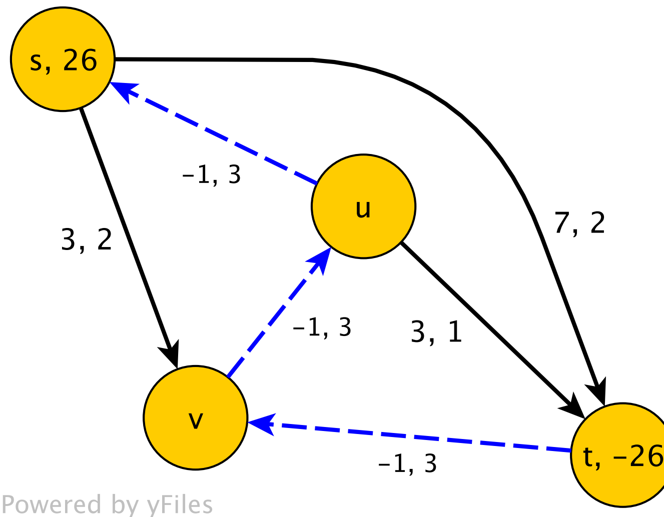

Figure 18: The residual graph G'_1. We see that the shortest $s - t$-path is now $(svut)$.

The resulting dynamic flow is shown in 12 time snaps in figure (19). Here F_1 is shown in light green (full-drawn) and F_2 is shown in red (dotted). Note that F_2 starts flowing from u *before* it reaches v, which is before u in P_2. This is of course because it travels along uv in the negative direction, with negative travel time. This results in a flow of value 2 along uv when both chains use the arc, and 3 when only F_1 uses it. The arcs unique to F_1 are not affected by F_2.

The rest of this section will be an attempt to prove the following theorem:

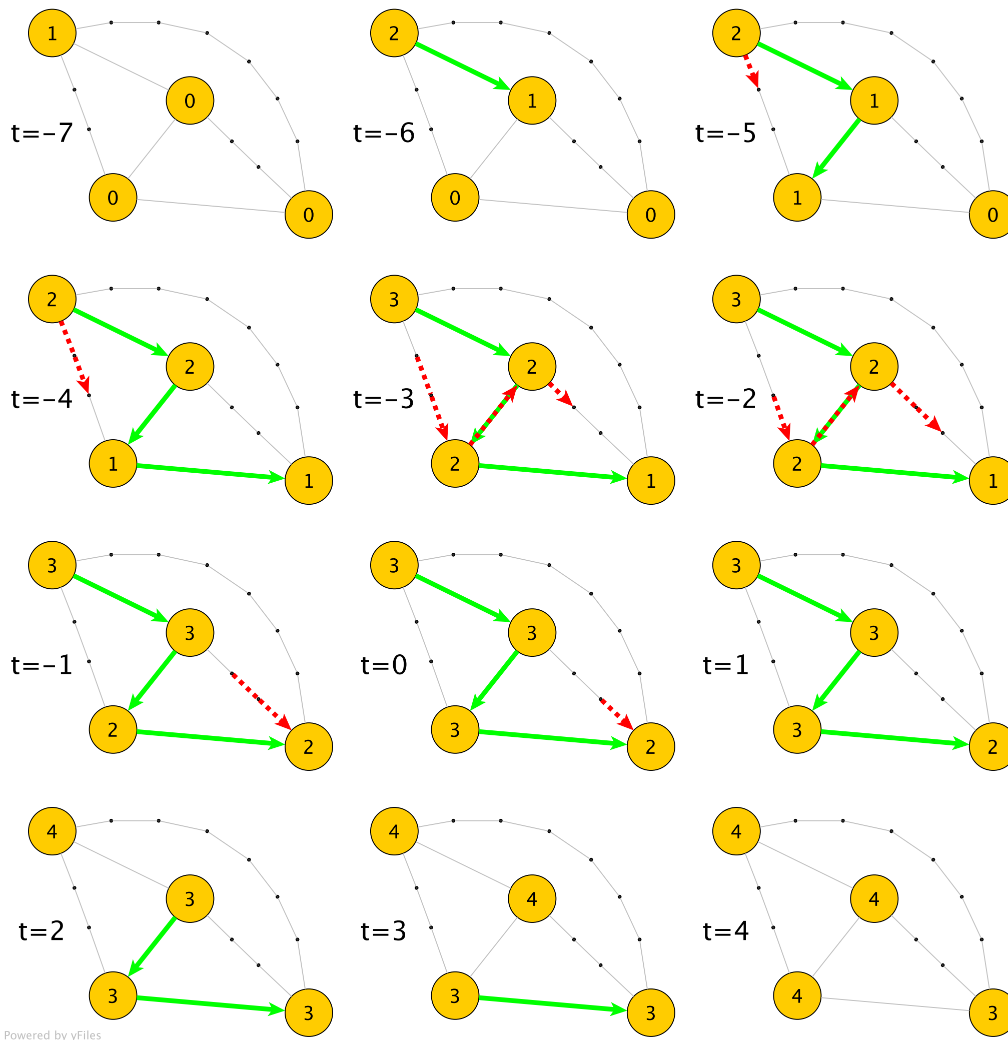

Figure 19: The system optimal dynamic flow shown at 12 different time steps. Green (full drawn) represents the chain F_1, while red (dotted) represents F_2.
The numbers inside the vertices is the number of the *slice* they belong to (Definition 5.3 below).

Theorem 5.5 *The Dynamic System Optimum algorithm above find a dynamic $s-t$-flow that is feasible and minimum cost.*

Proving that this theorem is correct will be some work. First let's show that it produces a dynamic flow that is feasible at all times. For this we need the following lemma:

Lemma 5.1 *Suppose f is a minimum cost static flow in G, and let static flow g augment f along a shortest $s-t$-path in residual graph G_f. Then (1) $f+g$ is a minimum cost static flow in G and (2) for any vertex v the distance from s to v in G_f is less than or equal to the distance in G_{f+g}, or $d_f(s,v) \leq d_{f+g}(s,v)$, and $d_f(v,t) \leq d_{f+g}(v,t)$.*

For a proof refer to [4].
If we denote the static flows induced by $\cup_{i=1}^{k} F_i$, the first k elements of F, by f_k for each k it is clear that all these are feasible minimum cost static flows or their respective values, as these are calculated exactly as in the Successive Shortest Path algorithm [1].
If we can show that for each vertex v the time intervals $\tau_{F_{i_m}}(v)$ in which each F_{i_m} cover v is ordered with regard to inclusion we must therefore have that all constraints are satisfied at all vertices (and arcs) at all times.

Lemma 5.2 *For any vertex v and any chains $F_i, F_j, j < i$ covering v at time intervals $\tau_{F_i}(v), \tau_{F_j}(v)$ we have*

$$\tau_{F_i}(v) \subset \tau_{F_j}(v)$$

Proof. For the source and sink vertices this is easy to show, as for each F_i we have

$$H_i(t_i^s + l_i) = C_{max} = H(t_i^e)$$

So for the starting times we have

$$g(t_i^s) + h(t_i^s + l_i) + l_i = g(t_{i-1}^s) + h(t_{i-1}^s + l_{i-1}) + l_{i-1}$$

By assumption on h we have $h'(t) \geq -1$ which gives

$$g(t_{i-1}^s) + h(t_{-1}i^s + l_{i-1}) + l_{i-1} \leq g(t_{i-1}^s) + h(t_{i-1}^s + l_i) + l_i$$

Now since t_i^s is such that

$$g(t_i^s) + h(t_i^s + l_i) + l_i = C_{max}$$

where the left hand side is non-increasing, and since

$$g(t_i^s) + h(t_i^s + l_i) \leq g(t_{i-1}^s) + h(t_{i-1}^s + l_i)$$

we can conclude that

$$t_i^s \geq t_{i-1}^s$$

Similarly, with the assumption that $g'(t) \leq 1$, we get that

$$g(t_i^e - l_i) + h(t_i^e) \leq g(t_{i-1}^e - l_i) + h(t_{i-1}^e)$$

and again t_i^e is such that

$$g(t_i^e - l_i) + h(t_i^e) = C_{max}$$

where the left hand side is non-decreasing. So we also get

$$t_i^e \leq t_{i-1}^e$$

So the inclusion holds at the terminal nodes.

Now assume P_i is a shortest path in G'_{i-1}, and consider any vertex v covered by F_i at some time. Assume that F_i reaches v before some $F_j, j < i$, and let Q_i, Q_j be the $s-v$-path components of P_i, P_j respectively. Since $t_{j,s} \leq t_{i,s}$ this means that $l(Q_i) < l(Q_j)$. Now Q_i is a shortest $s-v$-path in G'_{i-1}. If it wasn't then a shorter $s-v$-path could be combined with the $v-t$-component of P_i to make a shorter $s-t$-path than P_i, which contradicts P_i being a shortest $s-t$-path in G'_{i-1}. Similarly Q_j is a shortest $s-v$-path in G'_{j-1}. Then we have

$$l(Q_i) = d_{f_i}(s,v) < d_{f_j}(s,v) = l(Q_j)$$

This contradicts lemma (5.1), and thus F_i can not reach any vertex v before any $F_j, j < i$.

Similarly we prove that each chain F_i leaves each vertex v before each $F_j, j < i$. So the interval in which each chain covers each vertex is ordered by inclusion, and thus all constraints are satisfied at all vertices (and arcs) at all times, and the dynamic flow is at all times feasible. □

What remains is to show that the dynamic flow $\mathcal{F}$ is minimum cost of all dynamic flows with the same value, when considering both travel time and arrival deviation costs. This is of course the tricky part.

Let the dynamic flow found by the algorithm be $\mathcal{F}$ and let $\mathcal{Z}$ be a minimum cost dynamic flow with the same value. What I want to show is that the cost of $\mathcal{Z}$ if equal to the cost of $\mathcal{F}$. My strategy for proving this is to first split the difference $\mathcal{Z} - \mathcal{F}$ between the two dynamic flows up into smaller, independent parts, and then show that for each of these parts the total cost is non-negative. Then since the difference $\mathcal{Z} - \mathcal{F}$ is the sum of all the smaller parts, we also get that the cost of $\mathcal{Z} - \mathcal{F}$ is non-negative, which implies that $\mathcal{F}$ is also minimum cost.

I will need some lemmas for this second part of the proof. First a definition.

Definition 5.2 *A static $s-t$-flow f is extreme if it is minimum cost among all static $s-t$-flows with the same value.*

So all flows in the Successive Shortest Path algorithm, and also the static flows in this algorithm, are extreme. There is a useful characterization of an extreme flow:

Lemma 5.3 *A flow f is extreme if and only if the residual graph G_f contains no negative cost cycles (or circulations).*

A proof can be found in [1].
This is only valid for static flows and networks, whereas my network is dynamic. I want to prove the similar claim that the *dynamic* residual graph $G_{\mathcal{F}}$ contains no negative length cycle. To do this I introduce the concept of a *slice*. In the proof of feasibility above we saw that for a vertex v the time intervals $\tau_{F_{i_m}}(v)$ for which chains F_{i_m} covers v is ordered by inclusion. Then we can look at the time intervals in which each vertex v sees the residual graph G_{f_j}, that is the time intervals in which $F_{i_m}, i_m \le j$ cover v, but no $F_{i_m}, i_m > j$ cover v. (Note that the residual graphs G_{f_j} seen by vertex v will be the same for $i = i_m, \dots i_{m+1} - 1$, but we still consider them separately.) Then the (possibly empty) time intervals in which vertex v sees residual graph G_{f_j} is $\mathcal{T}_j^1(v) = [t_j^s + d_{f_{j-1}}(s,v), t_{j+1}^s + d_{f_j}(s,v))$ and $\mathcal{T}_j^2(v) = (t_{j+1}^e - l_{j+1} + d_{f_j}(s,v), t_j^e - l_j + d_{f_{j-1}}(s,v)]$ for $j < K$, and for $j = K$ the time interval will be $\mathcal{T}_K^1(v) = \mathcal{T}_K^2(v) = [t_j^s + d_{f_{j-1}}(s,v), t_j^e - l_j + d_{f_{j-1}}(s,v)]$. Also define $t_0^s = -\infty, t_0^e = \infty$. Then we define the *slice*:

Definition 5.3 *The j-th slice is the union*

$$\cup_{v \in V} v \times \mathcal{T}_j^1(v)$$

of all vertices v at their respective time intervals $\mathcal{T}_j^1(v)$, the $(2K-j)$-th slice is the union

$$\cup_{v \in V} v \times \mathcal{T}_j^2(v)$$

of all vertices v at their respective time intervals $\mathcal{T}_j^2(v)$.

And with this we can prove the important lemma:

Lemma 5.4 *It is impossible to go from an i-th slice to a j-th slice with $j < i$ in the dynamic residual graph $G_{\mathcal{F}}$.*

Proof. Look at (v,t) in the i-th slice. Then either (1) $t \in [t_i^s + d_{f_{i-1}}(s,v), t_{i+1}^s + d_{f_i}(s,v))$ or (2) $t \in (t_{i+1}^e - l_{i+1} + d_{f_i}(s,v), t_i^e - l_i + d_{f_{i-1}}(s,v)]$ or (3) if $i = K$ then $t \in [t_i^s + d_{f_{i-1}}(s,v), t_i^e - l_i + d_{f_{i-1}}(s,v)]$.
(2) Look at vertex u. Then by traveling within the i-th slice the earliest we can get to u is

$$t + d_{f_i}(v,u) \geq t_{i+1}^e - l_{i+1} + d_{f_i}(s,v) + d_{f_i}(v,u)$$

But we know that

$$d_{f_i}(s,v) + d_{f_i}(v,u) \geq d_{f_i}(s,u)$$

so this leads to

$$t + d_{f_i}(v,u) \geq t_{i+1}^e - l_{i+1} + d_{f_i}(s,u)$$

which is also in the i-th slice.
(1,3) Look at vertex u. Then by traveling within the i-th slice the earliest we can get to u is

$$t + d_{f_i}(v,u) \geq t_i^s + d_{f_{i-1}}(s,v) + d_{f_i}(v,u)$$

Now assume

$$d_{f_{i-1}}(s,v) + d_{f_i}(v,u) < d_{f_{i-1}}(s,u)$$

Let P_{sv} be a shortest $s - v$-path in $G_{f_{i-1}}$ and P_{vu} a shortest $v - u$-path in G_{f_i}. Then clearly $P_{vu} \cap P_i^{-1} \neq \emptyset$, since F_i must have made a shorter path from v to u possible. With $P_{vu} = a_1 a_2 \ldots a_m$ let w be such that i is maximal when $a_i = yw \in P_{vu} \cap P_i^{-1}$ for some vertex y, and let x be such that i is minimal when $a_i = zx \in P_{vu} \cap P_i^{-1}$ for some vertex z. Let P_{wx} be the $w - x$-component of P_i and P_{xw} the $x - w$-component of P_{vu}. Then we must have

$$l(P_{xw}) + l(P_{wx}) = 0 \tag{30}$$

. Clearly it cannot be greater, as P_{xw} can be no longer than P_{wx}^{-1}, and if it was less then a flow of value ϵ along P_{wx} and back along P_{xw} would be a negative cost circulation, contradicting the fact that f_{i-1} is extreme.
We can rewrite the assumption above as

$$l(P_{sv}) + l(P_{vu}) < l(P_{su}) \tag{31}$$

Now since F_i only affects P_{vu} along P_{xw} we can split P_{vu} in P_{vx}, P_{xw}, P_{wu} where the first and last ones are paths in $G_{f_{i-1}}$. It is clear that

$$l(P_{sw}) + l(P_{wu}) \geq l(P_{su}) \tag{32}$$

But then (32) and (31) together with the splitting of P_{vu} become

$$l(P_{sw}) + l(P_{wu}) > l(P_{sv}) + l(P_{vx}) + l(P_{xw}) + l(P_{wu}) \tag{33}$$

Together with (30) we finally get

$$l(P_{sw}) + l(P_{wx}) > l(P_{sv}) + l(P_{vx}) \tag{34}$$

which contradicts P_i being a shortest $s-t$-path in $G_{f_{i-1}}$!
Thus

$$t_i^s + d_{f_{i-1}}(s, v) + d_{f_i}(v, u) \geq t_i^s + d_{f_{i-1}}(s, u)$$

which is again in the i-th slice.
So it is impossible to get to an earlier slice by traveling within one slice (or one static residual graph). Now it could be possible to get to some j-th slice with $j < i$ from a k-th slice, with $k > i$. But since there is a *last* slice, the $(2K)$-th slice, we see by induction that this is also impossible.

□

From this it is clear that a negative length cycle in the dynamic residual graph must stay within one slice. But a negative cost cycle within one slice would imply a negative cost cycle in the static residual graph corresponding to that slice, which contradicts the fact that all the static flows are extreme. Thus we have:

Corollary 5.2 *The dynamic residual graph $G_{\mathcal{F}}$ contains no negative length cycles.*

Another result we get is

Corollary 5.3 *All shortest (1) $s-t$-paths and (2) $t-s$-paths in the dynamic residual graph stay within one slice.*

Proof. (1) A path ending at t in the i-th slice can leave s in the i-th slice, but not later. (2) Similarly a path starting from t in the j-th slice must follow a flow from s to t, all of which follows paths of length $0 \leq l \leq d_{f_{j-1}}(s, t)$. Thus this flow cannot have started from s before the j-th slice. □

Note that the second statement is equivalent to saying that the slowest flows in use by $\mathcal{F}$ are not slow enough to fall through to later time slices.
I am finally ready to prove Theorem 5.5.

Proof. [Theorem 5.5] Let $\mathcal{F}$ be the dynamic flow found by the algorithm and let $\mathcal{Z}$ be a minimum cost dynamic flow with the same value. Then the difference between the two dynamic flows $\mathcal{Z} - \mathcal{F}$ can be written as a sum

of non-canceling dynamic circulations, possibly with non-constant flow values, and including waiting at vertices. By non-canceling I mean that if a dynamic circulation y has positive flow along arc a at time t then no other dynamic circulation x can have negative flow along a (or positive flow along a^{-1}) at time t that cancels the flow of y. This independence is important because we then know that each of these dynamic circulations have to be feasible when added together with $\mathcal{F}$, which again means that any positive flow in such a circulation y has to be feasible together with flow in $\mathcal{F}$, and any negative flow in y can only cancel flow in $\mathcal{F}$.
Now remember that the total cost of $\mathcal{F}$ was determined fully by outflow from s and inflow to t. Consider the cost of an infinitesimal piece of one of the circulations y, with constant travel time l_y and unit value. If this circulation contains neither s nor t the cost is 0. If it contains only s, and leaves s at time τ the cost is

$$g(\tau) - g(\tau + l_y) + l_y$$

If it contains only t, and leaves t at time τ the cost is

$$-h(\tau) + h(\tau + l_y) + l_y$$

In both these cases we see that for this cost to be negative we must have $l_y < 0$, because of the assumptions $g'(\tau) \leq 1, h'(\tau) \geq -1$. But we have already proven in Corollary 5.2 that this is impossible.
If it contains both s and t, leaves s at τ_s, travels to t along a path of length l_y^+, leaves t at τ_t and travels to s again along a path of length l_y^- the cost is

$$g(\tau_s) + h(\tau_s + l_y^+) + l_y^+ - h(\tau_t) - g(\tau_t + l_y^-) + l_y^-$$

In this case we can consider the infinitesimal dynamic circulation as two separate chains y^+, y^-, where y^+ comes in addition to $\mathcal{F}$, and y^- cancels some part of $\mathcal{F}$. Let the costs of these two chains be C_{y^+}, C_{y^-}. Now we already know that $-C_{y^-} \leq C_{max}$. So for the total cost of y to be negative we need $C_{y^+} < C_{max}$. y^+ has minimum cost if it follows a shortest path in the dynamic residual graph. Then look at a shortest path P_j in the i-th slice, with cost function $H_j(t)$ as in the algorithm. Then for $t \in R \setminus \tau_{F_j}(t)$ we have $H_j(t) \geq C_{max}$ since H_j is convex with a minimum in $\tau_{F_j}(t)$ and because of the way t_j^s, t_j^e are selected in the algorithm. So $C_{y^+} \geq C_{max}$, and we finally have that $C_{y^-} + C_{y^+} \geq 0$, and again that $C_{z-f} \geq 0$. And this means that $\mathcal{F}$ is also minimum cost! □

6 Summary

In this paper I have studied system optimal and user equilibrium flows in static and dynamic networks, and how to find these in both general and more specific situations. Of most interest are the alternative characterization of static system optimal flows, the algorithm for turning a static system optimal flow into a user equilibrium by taxes along the arcs, and the chain flow algorithm for solving the dynamic system optimality problem with the assumption of constant latency functions on the arcs. I will summarize these here.

- *Alternative static system optimality criterion*
  The definition of the user equilibrium was defined locally on each $s-t$-path. Each $s-t$-path in use has equal cost and cost less than or equal to that of $s-t$-path. We saw that it was possibly to formulate this also as a minimization problem that turned out to be a convex optimization problem.
  The system optimum was defined globally as a minimization problem, also a convex optimization problem. What I showed in section 5.1.4 was that in this case there was also an equivalent path-local definition, similar to that of the user equilibrium. Each $s-t$-path in use has incremental cost

$$\mathcal{L}_P = \frac{d}{dx_P}\left(\sum_{a\in P} x_a l_a(x_a)\right)$$

  that is equal, and equal to or less than that of any unused $s-t$-path.

- *Optimal taxing in a static network*
  I then went on in section 5.2 to find a way of turning a system optimal flow into a user equilibrium by adding taxes along the arcs of the network. This came from the idea that any static flow f_s gives rise to certain latencies along each arc in the graph G. Using the fixed graph G^{f_s} consisting of the arcs used by this flow and with latencies caused by this flow, we can find constant taxes for each arc in G^{f_s} such that each $s-t$-path in G^{f_s} has equal length. Then adding these taxes to the original graph G, with non-constant latencies, causes exactly the flow f_s to be a user equilibrium in G, as each $s-t$-path in G then has equal length exactly with f_s.
  I also pointed out that any flow f_s can be turned into a user equilibrium by this process, but choosing the system optimal flow makes the most sense in my case.

I have also programmed this algorithm, and tested it for some simple graphs. The code is found in the appendix.

- *Dynamic system optimum algorithms*
  Although I showed in section 5.3 that it is possible to compute the dynamic system optimal solution by making a time discretization of the problem, this will often be impractical due to the sheer size of the discretized optimization problem, even though the problem is such that the Simplex algorithm, or even the Network Simplex algorithm, can be used to solve it.
  For the single commodity case I found an algorithm in section 5.4 that works by gradually increasing the number of paths used and the time intervals of use of each of these. This is done such that we always add flow at the lowest possible cost. The result of this is a dynamic flow with the desired value, and with minimum total cost, represented as a set of chains. Each chain is a path, possibly with negative arcs, a value of constant flow along this path, and a time interval during which the flow is present along this path. Adding all the chains together gives the total dynamic flow. The algorithm runs in a time polynomial to the number of vertices and arcs, and proportional to the total supply, assuming the capacities are integer. This is a huge improvement compared to the time discretization approach!
  Although I only had time to develop the algorithm for the single commodity case, I also think it would be possible to develop a similar algorithm for the multi-commodity case, based on algorithm presented here and the quickest transshipment problem in [4].
  It should also be mentioned that although the time discretization approach might be slow, it is however capable of solving more general problems. It handles with ease time varying capacities, arbitrary departure/arrival deviation cost functions and multiple commodities. I also programmed code for turning a directed graph with supplies and departure/arrival cost functions into a time discretized graph, removing unused vertices and arcs. I also wrote some code for converting the graph to a format readable by a graph drawing program, and more importantly for formulating the LP problem of finding the dynamic system optimal flow through the graph in the AMPL language. This code is also found in the appendix.

7 Appendix

Below follow the .java files containing the code I have written for handling these graphs on the computer.

Arc represents an arc used in the graph implementation.

```
package trafficGraphs;

public class Arc {
	private double length, capacity, toll, flow;
	private Vertex source, target;
	private String name;

	public Arc(String name, double length, double capacity, double toll,
			Vertex source, Vertex target) {
		// this.name = name;
		this.name = source.getName() + target.getName() + (length + toll);
		this.length = length;
		this.capacity = capacity;
		this.toll = toll;
		this.source = source;
		this.target = target;
		source.getOutArcs().add(this);
		target.getInArcs().add(this);
	}

	public Arc(String name, double length, double capacity, Vertex source,
			Vertex target) {
		this(name, length, capacity, 0, source, target);
	}

	public Arc(String name, double length, Vertex source, Vertex target) {
		this(name, length, Double.POSITIVE_INFINITY, source, target);
	}

	public double getCapacity() {
		return capacity;
	}

	public double getCost() {
		return length + toll;
	}

	public double getLength() {
		return length;
	}

	public String getName() {
		return name;
	}

	public String getQuotedName() {
		return "\"" + name + "\"";
	}

	public Vertex getSource() {
		return source;
	}

	public Vertex getTarget() {
		return target;
	}

	public double getToll() {
		return toll;
	}

	public double getFlow() {
		return flow;
	}

	public void setToll(double toll) {
		this.toll = toll;
	}

	public void setFlow(double flow) {
		this.flow = flow;
	}
```

```
	public String toGraphString() {
		return source.getQuotedName() + " -> " + target.getQuotedName()
				+ " [label = \""
				+ (capacity == Double.POSITIVE_INFINITY ? "-" : capacity)
				+ ", " + getCost() + "\"]";
	}

	public String toTextFileString() {
		return "A " + name + " " + length + " " + capacity + " " + toll + " "
				+ source.getName() + " " + target.getName();
	}
}
```

AugementedGraph extends Graph, and represents the time discretization of the original graph.

```
package trafficGraphs;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.*;

public class AugmentedGraph extends Graph {
	protected HashMap<String, Vertex[]> augVertices;
	protected double step;
	protected int max;

	public AugmentedGraph(int max, double step) {
		super();
		this.max = max;
		this.step = step;
		augVertices = new HashMap<String, Vertex[]>();
	}

	public void toGraphFile(String filename) {
		PrintWriter writer;
		try {
			writer = new PrintWriter(filename + ".dot");
		} catch (FileNotFoundException e) {
			System.err.println("The file was not found!");
			e.printStackTrace();
			return;
		}
		writer.println("digraph {\nrankdir = LR\nsplines = false\n");
		for (Vertex[] vertexs : augVertices.values()) {
			writer.println("{\nrank = same;\n");
			for (Vertex vertex : vertexs)
				if (vertex != null)
					writer.println(vertex.getQuotedName());
			writer.println("}\n");
		}
		for (Vertex vertex : vertices.values())
			writer.println(vertex.getQuotedName());
		writer.println();
		for (Arc arc : arcs.values())
			writer.println(arc.toGraphString());

		writer.println("}");
		writer.close();
	}

	public void removeVertex(Vertex vertex) {
		super.removeVertex(vertex);
		String[] vinfo = vertex.getName().split(",");
		augVertices.get(vinfo[0])[(int) (Double.parseDouble(vinfo[1]) / step)] = null;
	}

	public AugmentedGraph augment() {
		throw new GraphAlreadyAugmentedException();
	}

	public class GraphAlreadyAugmentedException extends RuntimeException {
		private static final long serialVersionUID = -1;
	}
}
```

CostFunction is used to represent an arrival or departure cost function.

```
package trafficGraphs;
```

```
public abstract class CostFunction {
	public abstract double getCost(double time, double step);

	public static final CostFunction nullFunction = new CostFunction() {
		public double getCost(double time, double step) {
			return 0;
		}
	};

	public static CostFunction makeCostFunction(final String format) {
		if (format.equals("null"))
			return nullFunction;
		if (format.split(" ").length == 3)
			return new CostFunction() {
				private double a, b, t;
				{
					String[] s = format.split(" ");
					a = Double.parseDouble(s[0]);
					b = Double.parseDouble(s[1]);
					t = Double.parseDouble(s[2]);
				}

				public double getCost(double time, double step) {
					double lower = time - 0.5 * step;
					double upper = time + 0.5 * step;
					lower = (t - lower > 0 ? Math.min((t + 0.5 * step - lower)
							/ step, 1) : 1)
							* costAt(lower);
					upper = (t - upper < 0 ? Math.min((-t + 0.5 * step + upper)
							/ step, 1) : 1)
							* costAt(upper);
					return 0.5 * (lower + upper);
				}

				private double costAt(double time) {
					return time < t ? -a * (time - t) : b * (time - t);
				}
			};
		return makeCostFunction("0.5 2 5");
	}
}
```

CostVertex extends Vertex, and is used for vertices with supplies, and therefore cost functions.

```
package trafficGraphs;

public class CostVertex extends Vertex {
	private CostFunction function;
	private double magnitude;

	public CostVertex(String name, CostFunction function) {
		super(name);
		this.function = function;
	}

	public CostVertex(String name, double magnitude, CostFunction function) {
		super(name);
		setMagnitude(magnitude);
		this.function = function;
	}

	public CostFunction getCostFunction() {
		return function;
	}

	public double getCost(double time, double step) {
		return function == null ? 0 : function.getCost(time, step);
	}

	public String toGraphString() {
		return getQuotedName()
				+ (magnitude != 0 ? " [style = doublecircle, label = \""
						+ getName() + " : " + magnitude + "\"]" : "") + ":";
	}

	public String toTextFileString() {
		return "C" + super.toTextFileString() + " " + magnitude + " ";
	}

	public double getMagnitude() {
		return magnitude;
```

```
	}

	public void setMagnitude(double magnitude) {
		this.magnitude = magnitude;
	}

	public boolean isRemovable() {
		return super.isRemovable() && magnitude == 0;
	}
}
```

Graph is the main class for representing a graph. It uses most of the other classes here, and also contains the code that performs the discretization.

```
package trafficGraphs;

import java.util.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class Graph {
	protected HashMap<String, Vertex> vertices;
	protected HashMap<String, Arc> arcs;

	public Graph() {
		vertices = new HashMap<String, Vertex>();
		arcs = new HashMap<String, Arc>();
	}

	public Graph(String filename) {
		this();
		fromTextFile(filename);
	}

	/**
	 * Reads a graph from a text file.
	 *
	 * Assumes all vertices first, format: 'V' name 'CV' name magnitude [cost
	 * function]
	 *
	 * Then arcs, format: 'A' name length [capacity [toll]] source target
	 *
	 *
	 * @param filename
	 */
	protected void fromTextFile(String filename) {
		Scanner scanner;
		try {
			scanner = new Scanner(new File(filename + ".txt"));
		} catch (FileNotFoundException e) {
			System.err.println("The file was not found!");
			e.printStackTrace();
			return;
		}
		String type;
		int linen = 0;
		while (scanner.hasNextLine()) {
			try {
				type = scanner.next();
				if (type.equals("V")) {
					addVertex(new Vertex(scanner.next()));
				} else if (type.equals("CV")) {// TODO: Fixit cost function.
					addVertex(new CostVertex(scanner.next(), scanner.nextDouble(),
							CostFunction.makeCostFunction(scanner.nextLine().trim())));
				} else if (type.equals("A")) {
					String[] line = scanner.nextLine().trim().split(" ");
					switch (line.length) {
					case 4:
						addArc(new Arc(line[0], Double.parseDouble(line[1]),
								vertices.get(line[2]), vertices.get(line[3])));
						break;
					case 5:
						addArc(new Arc(line[0], Double.parseDouble(line[1]),
								Double.parseDouble(line[2]),
								vertices.get(line[3]), vertices.get(line[4])));
						break;
					case 6:
						addArc(new Arc(line[0], Double.parseDouble(line[1]),
								Double.parseDouble(line[2]),
```

```
                                        Double.parseDouble(line[3]),
                                        vertices.get(line[4]), vertices.get(line[5])));
                                break;
                        default:
                                System.err.println("Wrong number of arguments for arc "
                                                + line[0]);
                        }

                }
            } catch (Exception e) {
                System.err.println("Error at line " + linen);
            }
            linen++;
        }
        scanner.close();
}

public void toAMPLFile(String filename) {
        PrintWriter writer;
        try {
                writer = new PrintWriter(filename + ".dat");
        } catch (FileNotFoundException e) {
                System.err.println("The file was not found!");
                e.printStackTrace();
                return;
        }

        writer.print("set Arcs := ");
        for (Arc arc : arcs.values())
                writer.print(arc.getQuotedName() + " ");
        writer.println(";\n");

        writer.print("set Vertices := ");
        for (Vertex vertex : vertices.values())
                writer.print(vertex.getQuotedName() + " ");
        writer.println(";\n");

        writer.println("param:\td\tc\t:=");
        for (Arc arc : arcs.values())
                writer.println(arc.getQuotedName() + " " + arc.getCost() + " "
                                + (arc.getCapacity() == Double.POSITIVE_INFINITY ?
                                                "Infinity" : arc.getCapacity()));
        writer.println(";\n");

        writer.println("param:\tm\t:=");
        for (Vertex vertex : vertices.values())
                writer.println(vertex.getQuotedName() + " "
                                + vertex.getMagnitude());
        writer.println(";\n");

        writer.println("set Entering :=");
        for (Vertex vertex : vertices.values()) {
                // writer.printf("(%s, *)", vertex.getQuotedName());
                for (Arc arc : vertex.getInArcs())
                        writer.print(" " + vertex.getQuotedName() + " "
                                        + arc.getQuotedName());
                writer.println();
        }
        writer.println(";\n");
        writer.println("set Leaving :=");
        for (Vertex vertex : vertices.values()) {
                // writer.printf("(%s, *)", vertex.getQuotedName());
                for (Arc arc : vertex.getOutArcs())
                        writer.print(" " + vertex.getQuotedName() + " "
                                        + arc.getQuotedName());
                writer.println();
        }
        writer.println(";\n");
        /*
         * for (Vertex vertex : vertices.values()) { writer.printf("set
         * Entering%s := ", vertex.getName()); for (Arc arc :
         * vertex.getInArcs()) writer.print(arc.getName() + "\t");
         * writer.println(";\n");
         *
         * writer.printf("set Leaving%s := ", vertex.getName()); for (Arc arc :
         * vertex.getOutArcs()) writer.print(arc.getName() + "\t");
         * writer.println(";\n"); }
         */

        writer.close();

        try {
                writer = new PrintWriter(filename + ".mod");
```

```
        } catch (FileNotFoundException e) {
                System.err.println("The file was not found!");
                e.printStackTrace();
                return;
        }

        writer.println("set Vertices;");
        writer.println("set Arcs;");

        writer.println("set Entering within {Vertices, Arcs};");
        writer.println("set Leaving within {Vertices, Arcs};");
        /*
         * for (Vertex vertex : vertices.values()) { writer.printf("set
         * Entering%s;%n", vertex.getName()); writer.printf("set Leaving%s;%n",
         * vertex.getName()); }
         */

        writer.println("\nparam d {Arcs} >= 0;");
        writer.println("param c {Arcs} > 0;");
        writer.println("param m {Vertices};");

        writer.println("\nvar f {a in Arcs} >= 0, <= c[a];");
        writer.println("\nminimize Cost: sum {a in Arcs} f[a] * d[a];");
        writer.println("\nsubject to Flow {v in Vertices}: m[v] + "
                        + "sum {(v, a) in Entering} f[a] = sum {(v, a) in Leaving} f[a];");
        /*
         * for (Vertex vertex : vertices.values()) { writer.printf("subject to
         * Flow%s: sum {a in Entering%s} = m[%s] + sum {a in Leaving%s};\n",
         * vertex.getQuotedName(), vertex.getQuotedName(),
         * vertex.getQuotedName(), vertex.getQuotedName()); }
         */

        writer.close();
}

public void toTextFile(String filename) {
        PrintWriter writer;
        try {
                writer = new PrintWriter(filename + ".txt");
        } catch (FileNotFoundException e) {
                System.err.println("The file was not found!");
                e.printStackTrace();
                return;
        }
        for (Vertex v : vertices.values())
                writer.println(v.toTextFileString());
        for (Arc a : arcs.values())
                writer.println(a.toTextFileString());
        // TODO: To file.

        writer.close();
}

public void toGraphFile(String filename) {
        PrintWriter writer;
        try {
                writer = new PrintWriter(filename + ".dot");
        } catch (FileNotFoundException e) {
                System.err.println("The file was not found!");
                e.printStackTrace();
                return;
        }
        writer.println("digraph {\nrankdir = LR\nsplines = false\n");
        for (Vertex vertex : vertices.values())
                writer.println(vertex.getQuotedName());
        writer.println();
        for (Arc arc : arcs.values())
                writer.println(arc.toGraphString());

        writer.println("}");
        writer.close();
}

public void addVertex(Vertex vertex) {
        vertices.put(vertex.getName(), vertex);
}

public void addArc(Arc arc) {
        arcs.put(arc.getName(), arc);
}

public void removeArc(Arc arc) {
        arc.getSource().getOutArcs().remove(arc);
```

```
		arc.getTarget().getInArcs().remove(arc);
		arcs.remove(arc.getName());
	}

	public void removeVertex(Vertex vertex) {
		for (Arc arc : vertex.getOutArcs()) {
			arc.getTarget().getInArcs().remove(arc);
			arcs.remove(arc.getName());
		}
		for (Arc arc : vertex.getInArcs()) {
			arc.getSource().getOutArcs().remove(arc);
			arcs.remove(arc.getName());
		}
		vertices.remove(vertex.getName());
	}

	public void removeArc(String arcName) {
		removeArc(arcs.get(arcName));
	}

	public void removeVertex(String vertexName) {
		removeVertex(vertices.get(vertexName));
	}

	public AugmentedGraph augment(int max, double step) {
		AugmentedGraph aug = new AugmentedGraph(max, step);

		// Create new vertices and arcs.
		for (Vertex v : vertices.values()) {
			Vertex[] vs = new Vertex[max];
			String vn = v.getName();
			for (int i = 0; i < max; i++) {
				vs[i] = new Vertex(vn + "," + (i * step), i * step);
				aug.addVertex(vs[i]);
				if (i > 0)
					aug.addArc(new WaitArc(vs[i - 1].getName()
							+ vs[i].getName(), step, vs[i - 1], vs[i]));
			}
			aug.augVertices.put(vn, vs);

			// Vertex is source or sink.
			if (v instanceof CostVertex) {
				CostVertex cv = (CostVertex) v;
				CostVertex uv = new CostVertex(vn, null);
				aug.addVertex(uv);
				uv.setMagnitude(cv.getMagnitude());
				int it = cv.getMagnitude() > 0 ? 0 : 1;
				Vertex[] sts = new Vertex[] { uv, null };
				for (Vertex av : vs) {
					sts[1] = av;
					aug.addArc(new Arc(vn + av.getName(), 0,
							Double.POSITIVE_INFINITY, cv.getCost(av.getTime(),
									step), sts[it], sts[1 - it]));
				}
			}
		}

		// Add arcs from the old graph.
		for (Vertex v : vertices.values()) {
			Vertex[] vs = aug.augVertices.get(v.getName());
			for (Arc a : v.getOutArcs()) {
				Vertex u = a.getTarget();
				int l = (int) (a.getLength() / step);
				double cap = a.getCapacity() * step
						* (l * step + 1 - a.getLength());
				double carry = a.getCapacity() * step
						* (a.getLength() - l * step);
				Vertex[] us = aug.augVertices.get(u.getName());
				for (int i = 0; i + l < max; i++)
					// TODO: Kanskje fjern carry igjen?
					if (Math.abs(cap) > 1e-6)
						aug.addArc(new Arc(vs[i].getName()
								+ us[i + l].getName(), a.getLength(), cap,
								a.getToll(), vs[i], us[i + l]));
				for (int i = 0; i + l + 1 < max; i++)
					if (Math.abs(carry) > 1e-6)
						aug.addArc(new Arc(vs[i].getName()
								+ us[i + l + 1].getName(), a.getLength(),
								carry, a.getToll(), vs[i], us[i + l + 1]));
			}
		}

		// Remove superfluous vertices and arc.
```

```
		LinkedList<Vertex> removables = new LinkedList<Vertex>();
		for (Vertex v : aug.vertices.values())
			if (v.isRemovable())
				removables.add(v);
		while (!removables.isEmpty()) {
			Vertex v = removables.poll();
			aug.removeVertex(v);
			for (Arc arc : v.getInArcs())
				if (arc.getSource().isRemovable())
					removables.add(arc.getSource());
			for (Arc arc : v.getOutArcs())
				if (arc.getTarget().isRemovable())
					removables.add(arc.getTarget());
		}
		return aug;
	}

	public static void test(String filename, int max, double step) {
		Graph g = new Graph(filename);
		g.toGraphFile(filename);
		AugmentedGraph aug = g.augment(max, step);
		aug.toGraphFile(filename + "aug");
		aug.toAMPLFile(filename + "aug");
	}

	public static void main(String[] argh) {
		test("grafenminja", 30, 0.5);
	}
}
```

TollFinder has the code for finding tolls to make all paths from a given vertex s to a given vertex t equally expensive. It assumes an acyclic graph. *This code is no longer correct, as I discovered an error in the theory around this algorithm, but didn't have time to rewrite the code.*

```
package trafficGraphs;

import java.util.Collection;
import java.util.HashMap;

public class TollFinder {
	private Collection<Arc> used;
	private HashMap<Vertex, VertexWrapper> wraps;

	private TollFinder(Collection<Arc> used) {
		this.used = used;
		wraps = new HashMap<Vertex, VertexWrapper>();
	}

	public static void findTolls(Vertex s, Vertex t, Collection<Arc> used) {
		TollFinder tf = new TollFinder(used);
		tf.makeSubgraphOfUsedArcs(s, t);
		tf.calculateTolls(tf.wraps.get(s));
	}

	private void makeSubgraphOfUsedArcs(Vertex s, Vertex t) {
		putInSubgraph(s);
		wraps.get(t).distance = 0;
	}

	private void putInSubgraph(Vertex v) {
		if (wraps.containsKey(v))
			return;
		VertexWrapper vw = new VertexWrapper(v);
		wraps.put(v, vw);
		for (Arc a : vw.v.getOutArcs())
			if (used.contains(a))
				putInSubgraph(a.getTarget());
	}

	private double calculateTolls(VertexWrapper vw) {
		if (vw.distance >= 0)
			return vw.distance;
		if (vw.distance == -2)
			throw new RuntimeException("The graph was not acyclic! " + vw.v
						+ " encountered while active.");
		vw.distance = -2;
		double maxd = 0;
		for (Arc a : vw.v.getOutArcs())
			if (used.contains(a))
```

```
				maxd = Math.max(maxd, calculateTolls(wraps.get(a.getTarget()))
							+ a.getLength());
		for (Arc a : vw.v.getOutArcs())
			if (used.contains(a))
				a.setToll(maxd - wraps.get(a.getTarget()).distance
							- a.getLength());
		return vw.distance = maxd;
	}

	private class VertexWrapper {
		private Vertex v;

		private double distance;

		private VertexWrapper(Vertex v) {
			this.v = v;
			distance = -1;
		}
	}

	public static void main(String[] args) {
		Vertex s = new Vertex("s");
		Vertex t = new Vertex("t");
		Vertex u = new Vertex("u");
		Vertex v = new Vertex("v");
		Vertex w = new Vertex("w");
		Arc su = new Arc("su", 1, s, u);
		Arc uv = new Arc("uv", 3, u, v);
		Arc vt1 = new Arc("vt1", 1, v, t);
		Arc vt2 = new Arc("vt2", 2, v, t);
		Arc uw = new Arc("uw", 1, u, w);
		Arc wt = new Arc("wt", 1, w, t);
		Collection<Arc> used = new java.util.HashSet<Arc>();
		used.add(su);
		used.add(uv);
		used.add(vt1);
		used.add(vt2);
		used.add(uw);
		used.add(wt);

		findTolls(s, t, used);
		for (Arc a : used)
			System.out.format(
					"%s: Total length: %.0f, of which %.0f is toll.\n",
					a.getName(), a.getCost(), a.getToll());
	}
}
```

Vertex represents vertices in the graph.

```
package trafficGraphs;

import java.util.HashSet;
import java.util.Collection;

public class Vertex {
	private HashSet<Arc> inArcs, outArcs;
	private double time;
	private String name;

	public Vertex(String name, double time) {
		this.name = name;
		this.time = time;
		inArcs = new HashSet<Arc>();
		outArcs = new HashSet<Arc>();
	}

	public Vertex(String name) {
		this(name, Double.NaN);
	}

	public String getName() {
		return name;
	}

	public double getTime() {
		return time;
	}

	public double getMagnitude() {
		return 0;
	}
```

```
	public Collection<Arc> getInArcs() {
		return inArcs;
	}

	public Collection<Arc> getOutArcs() {
		return outArcs;
	}

	public String getQuotedName() {
		return "\"" + name + "\"";
	}

	public boolean isRemovable() {
		return inArcs.size() * outArcs.size() == 0;
	}

	public String toTextFileString() {
		return "V " + name;
	}
}
```

WaitArc extends Arc, and represents the arcs that are actually just waiting at the same vertex some one time step.

```
package trafficGraphs;

public class WaitArc extends Arc {
	public WaitArc(String name, double length, Vertex source, Vertex target) {
		super(name, length, source, target);
	}

	public String toGraphString() {
		return getSource().getQuotedName() + " -> "
				+ getTarget().getQuotedName()
				+ " [label = \"\", constraint = false]";
		// " [label = \"-, " + getCost() + "\", constraint = false]";
	}
}
```

References

[1] Alexander Schrijver *A course in combinatorial optimization* 2007

[2] B. G. Heydecker, J. D. Addison *Analysis of dynamic traffic equilibrium with departure time choice* TRANSPORTATION SCIENCE Vol. 39, No. 1, February 2005, pp. 39-57

[3] Jose R. Correa, Andreas S. Schulz, Nicolas E. Stier-Moses *Selfish routing in capacitated networks* MATHEMATICS OF OPERATIONS RESEARCH Vol. 29, No. 4, November 2004, pp. 961-976

[4] Bruce Edward Hoppe *Efficient dynamic network flow algorithms* Ph. D. Cornell University 1995

[5] Robert J. Vanderbei *LINEAR PROGRAMMING: Foundations and Extensions* Kluwer Academic Publishers, 2. edition, 2001