

Maximale Flüsse in fast-planaren Graphen

Maximum Flows in Nearly-planar Graphs

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt von
Dipl.-Math.
Jan Martin Hochstein
aus Göttingen

Referenten der Arbeit: Prof. Dr. Karsten Weihe
Prof. Dr. Michael Kaufmann

Einreichungsdatum : 18. Dezember 2006
Prüfungsdatum : 21. Mai 2007

Darmstadt 2007
D 17

Zusammenfassung

Das Maximalfluss-Problem ist eines der besterforschten Probleme in der kombinatorischen Optimierung. Gefragt ist hierbei nach dem maximalen statischen Durchsatz durch ein Netzwerk zwischen dem Startknoten s und dem Zielknoten t . Beispiele hierfür sind Netzwerke aus Rohren oder Kabeln oder Strassen- und Schienennetze.

Das Problem hat aber auch viele Verbindungen zu anderen Themen der kombinatorischen Optimierung und Anwendungen in so weit gestreuten Gebieten wie CAD, Bildverarbeitung und Projektplanung.

Es ist wohlbekannt, dass das Maximalfluss-Problem in planaren Graphen wesentlich schneller gelöst werden kann als in allgemeinen Graphen. Allerdings gibt es bisher keine Ergebnisse für *fast-planare* Graphen. Mit fast-planar bezeichnen wir Graphen, die wenige nicht-planare Stellen haben, wie zum Beispiel ein Straßennetz mit wenigen Brücken und Tunneln. Dabei ist es a priori nicht klar, wieviel “wenig” ist.

Unser Ziel ist es daher, Algorithmen zu finden, die das Maximalfluss-Problem in fast-planaren Graphen asymptotisch schneller lösen als die besten bekannten Algorithmen. Dazu betrachten wir die bekannten Algorithmen für planare Graphen und erweitern sie so, dass sie fast-planare Instanzen optimal lösen. Desweiteren untersuchen wir alle bekannten Lösungsansätze für allgemeine Graphen daraufhin, ob eine Veränderung der Algorithmen oder ihrer Laufzeitbeweise zu Verbesserungen der Laufzeitabschätzung führen.

Alle diese Untersuchungen führen wir sowohl empirisch als auch theoretisch durch. Denn dort wo theoretische Betrachtungen nicht weiter führen, liefert die Empirie möglicherweise neue Ideen für Algorithmen oder Beweise. Andererseits ist es auch interessant zu sehen, ob ein theoretisch guter Algorithmus den praktischen Anforderungen standhält.

Im Rahmen dieser Untersuchungen finden wir den ersten Algorithmus für ein Fluss-Problem speziell für fast-planare Graphen [HW07]. Er ist asymptotisch schneller als alle bekannten Algorithmen auf diesen Graphen. Die

Anzahl der nicht-planaren Stellen im Graphen geht dabei als Parameter in die Laufzeit ein, sodass sich für Graphen, die näher an der Planarität liegen, eine bessere Laufzeitabschätzung ergibt.

Desweiteren finden wir eine einfache Erweiterung eines bekannten Algorithmus', die auf unseren Testinstanzen ein sehr gutes Laufzeitverhalten aufweist. Nebenbei beweisen wir einige neue Beobachtungen für die theoretische Betrachtung des Maximalfluss-Problems. Andererseits können wir auch viele scheinbar erfolgversprechende Überlegungen endgültig widerlegen.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Methodological Approach	3
1.3	Background: Planar Graphs	4
1.4	Maximum Flow History and Background	7
1.5	Notation	8
1.6	The Model	10
1.7	Basic Facts about Maximum Flows	10
1.7.1	Limitations of the Ford-Fulkerson Algorithm	12
1.8	Problem Variants	13
2	Computational Investigation	17
2.1	Benchmark Algorithms	17
2.2	Real-world Data	18
2.3	Instance Classes	18
2.3.1	Properties of the Instances	19
2.4	Results	22
3	The Uppermost Path Algorithm and Some Variants	27
3.1	A Generic Augmenting Path Algorithm	28

3.2	Dynamic Tree Implementations	29
3.3	The Uppermost Path Algorithm	30
3.4	Completing a Sub-Optimal Solution	32
3.4.1	Repeated Application	33
3.4.2	Complete the Solution with Benchmark Algorithms	35
3.5	Planarization by Force	37
3.5.1	Implementation Difficulties	39
4	Theory for Augmenting Path Algorithms	41
4.1	The Generic Augmenting Path Algorithm	41
4.1.1	Direction Changes	43
4.2	Alternative Algorithm Description	44
4.3	Iteration Labels	44
4.4	Stubborn Algorithms	45
4.5	Augmenting Path Algorithms with Arc Memory	50
4.6	Stubborn and Balanced Algorithms with Arc Memory	52
4.7	List Scan Algorithms	53
4.8	Left-first Search Algorithms	54
4.8.1	StrictlyLeftmost	55
4.8.2	DeleteIncoming and EnableFastForward	56
4.8.3	FinalLabels	57
4.8.4	The Optimal Parameter Settings	57
4.8.5	A Theory for Left-first Search Algorithms	61
4.8.6	Regressions	65
4.8.7	Direction Changes	66
5	Blocking Flows	71

5.1	Dinic' Algorithm	71
5.2	Generalization	72
5.2.1	No Crossing Paths	73
5.2.2	No Reverse Augmentations	75
5.3	Karzanov's Algorithm	75
5.3.1	Application to Nearly-planar Instances	76
6	Generalized Preflow Push Algorithms	79
6.1	The Goldberg-Tarjan Algorithm	79
6.2	The Generic Algorithm	80
6.2.1	Correctness	81
6.2.2	Complexity	82
6.3	Nearly-planar Maxflow	84
6.4	Computational Results	86
6.5	Future Improvements	87
7	Dual Approaches	91
7.1	Shortest Paths in Planar Dual Graphs	91
7.2	Hassin's Idea	92
7.3	Generalization to the Non-planar Case	93
7.4	Trajectory Dependent Shortest Paths	93
7.5	Inhibit Flow Turn-off	96
7.6	Hyperfaces	97
7.6.1	Existence of a Solution	99
7.6.2	Finding Hyperfaces	100
8	Conclusion	103

8.1	Results in Detail	103
8.2	Outlook	105
A	The Implementation of the Left-first Search Algorithm	107

Chapter 1

Introduction

1.1 Motivation

Our main interest in this thesis is to find an algorithm for the maximum s - t -flow problem that runs asymptotically faster on nearly-planar instances than the best currently known algorithms. We begin by explaining and motivating this goal.

The maximum flow problem is one of the best investigated combinatorial problems. Informally it describes the task to optimize the throughput in a network. The term “throughput in a network” can be understood in many different ways. And there are as many variations to the maximum flow problem such as the balanced flow problem or the multicommodity flow problem.

In this thesis we will focus exclusively on the version that is commonly known in the literature as the maximum s - t -flow problem. That is, we are given a start node s and a target node t in the network and shall optimize the throughput from s to t subject to capacity restrictions in the network.

Intuitive examples for networks include road and railway networks. Here the nodes are crossroads, junctions or stations, and the edges are road or railway sections. The capacity of an edge would usually be calculated from the maximal throughput of vehicles or freight through such a section per time unit.

Pipe networks for fluids or gases are another example. These can be small like the hydraulic system in a machine or large like an intercontinental pipeline system. Nodes in these networks are pumps, pumping stations or reservoirs. The edges represent the connecting pipes, tubes or pipelines.

Again the edges' capacities are given by their maximum throughput per time unit.

Similar examples are power networks or electrical conductor networks on printed circuit boards or integrated circuits. Nodes, edges and capacities are derived in an analogous manner.

Telecommunication networks present a very inhomogenous class of examples. Nodes can be telephones, computers, transceiver stations or switches. Edges may be electrical or optical cables, radio links or laser beams — but these lists are far from complete. The edges' capacities depends on the kind of flow we are considering. For phone calls or data connections the capacity is the number of maximum concurrent calls or connections. However, we might also consider discrete voice or data messages or packets. In this case the capacity is given by the maximum number of such messages or packets per time unit.

However, the maximum flow problem has also applications in surface modelling [MMHW95], image segmentation [IG98], matrix rounding [CE82] and scheduling. There are many more applications and many more results that could be cited for each application. We can only give a brief overview here. For a great exposition of maximum flow theory and applications see [AMO93].

The maximum s - t -flow problem is not only interesting in its own right but also because of its many connections to other combinatorial problems. For example, the maximum edge-disjoint s - t -paths problem or the maximum matching problem in bipartite graphs are just special cases of the maximum s - t -flow problem.

On the other hand, the maximum s - t -flow problem is used as subroutine in a number of other problems such as the multicommodity flow problem or the minimum cost flow problem.

The currently best known algorithms for this problem do not use any geometric or topological properties of the underlying network. However, there are specific algorithms that solve the problem much faster if the network is planar. That is, if the network can be drawn without crossing edges in a plane. There is a rather sharp distinction between planar and non-planar. A network with only one crossing is already non-planar and thus the algorithms for the planar case fail to solve it.

This is unsatisfactory since we feel there should be a way to solve *nearly-planar* instances – instances that are near to planarity in some still undefined way – faster than general instances. Even more so as in many applications the purely planar case is rarely encountered. For example, most road net-

works have some tunnels or bridges and are thus non-planar. But they are not too far removed from planarity because the number of tunnels and bridges is small compared to the network size.

So our goal is to find nearly-planar networks and to define what makes them nearly-planar. And we address the question whether and how it is possible to solve these instances faster than general instances.

1.2 Methodological Approach

To achieve our aim we reconsider all known approaches to solving the maximum s - t -flow problem. We dedicate one chapter to each such approach. In particular we

- describe each approach,
- try to modify it and propose a new algorithm adapted to the nearly-planar case,
- give a formal analysis of the new algorithm and
- perform a computational investigation.

Existing algorithms for the planar case must be modified to find optimal solutions to nearly-planar instances. And for general maxflow algorithms we must find a way to describe their running time that results in a better worst case time bound for nearly-planar instances. In this latter case it is of great benefit to first derive instance parameters that the running time bounds depend on and then describe instances for which these parameters are small.

In cases where we can find formal proofs of correctness and complexity we use the computational investigation to analyse an algorithm's practical usefulness. And if we cannot find formal proofs then we try to derive strong evidence using the computational results.

In the rest of this chapter we give a more formal introduction to planar graphs and to the maximum s - t -flow problem and introduce notation that we will use throughout the thesis. In Chapter 2 we present our methods of computational investigation. In particular, we name a set of standard algorithms we use for comparison and describe our sets of test instances.

In Chapter 3 we consider Berge's uppermost path algorithm and try to extend it suitably to nearly-planar instances. Chapter 4 is dedicated to the

study of general augmenting path algorithms and to how they can perform better on nearly-planar instances. The same we do in Chapter 5 for blocking flow algorithms and in Chapter 6 for preflow push algorithms. In Chapter 7, finally, we consider Hassin's dual approach to the planar maximum flow problem and try to derive a fast algorithm for nearly-planar instances from this.

1.3 Background: Planar Graphs

The study of planar graphs has a long history beginning in the late 19th century. In 1869 Jordan stated the following theorem.

Theorem 1.3.1 (Jordan Curve Theorem [Jor69]). *A continuous simple closed curve separates the plane into two disjoint regions, the inside and the outside.*

In particular this means that any graph drawn in the plane such that no two edges cross, separates the plane into a number of disjoint regions. If the graph is thus drawn, we say that the graph is *embedded into the plane* and we call the regions *faces*. A graph that can be embedded into the plane is called *planar*. Note that not every possible drawing of a planar graph is a planar embedding.

This first definition of planarity is geometric. In 1930 Kuratowski gave a rather algebraic characterization using the special graphs shown in Figure 1.3.1

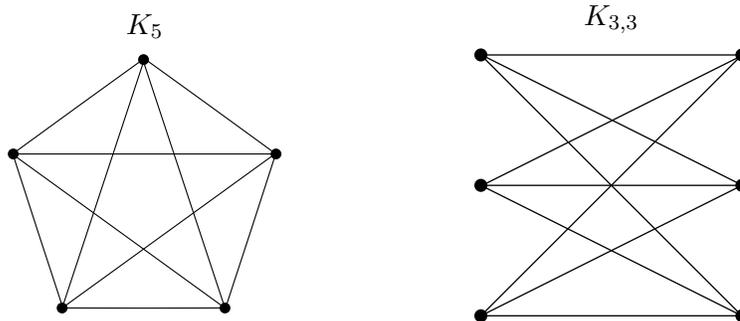


Figure 1.3.1.

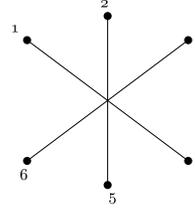
Theorem 1.3.2 (Kuratowski's Theorem [Kur30]). *A graph is planar if, and only if, it contains neither K_5 nor $K_{3,3}$ as a minor.*

As we have said before, we are interested in *nearly-planar* graphs here because of their greater applicability. However, there is no commonly accepted definition of near planarity.

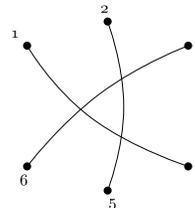
There are a number of possible ways to measure the degree of nonplanarity in a graph.

- One is the *crossing number*. That is the minimum number of crossings which is necessary when drawing the graph in the plane. In fact there are various different crossing numbers, for example the book crossing number or the rectilinear crossing number. These versions only differ in the type of graph drawing they allow. A common property of crossing numbers is that they are zero if and only if the graph is planar.
- This property also holds for the *genus*. This characteristic gives the minimum number of handles that have to be attached to the plane such that the given graph can be embedded in the resulting manifold.
- Another way to measure nonplanarity is the *graph thickness*. This is the minimum size of a graph decomposition in planar subgraphs. Obviously the graph thickness is one if and only if the graph is planar.
- The *splitting number* gives the minimum number of splitting operations such that the resulting graph is planar. In a *splitting operation* we replace a vertex v by two vertices v_1 and v_2 . Then we connect each neighbour of v to either v_1 or v_2 . Note that no edge is inserted between v_1 and v_2 .
- The size of a *maximum planar subgraph* also gives some measure of a graph's nonplanarity. For example, if the maximum planar subgraph is the whole graph then the graph is planar.

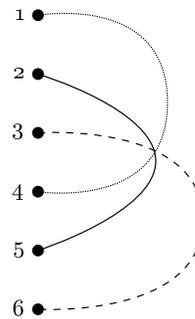
Rectilinear Crossing number



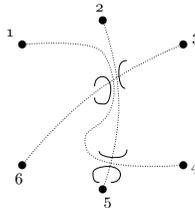
Pairwise Crossing number



Book Crossing number



Genus: Two handles



All these are ways to measure how planar a graph is. For some of them relations have been proven. For example, the genus is always less or equal

than the pairwise crossing number.

Common to all these concepts is that their computation is NP-complete. Therefore we will assume in this thesis that we are given a graph and a drawing of this graph in the plane. That way we do not need to compute an embedding.

We will see that in many cases it is not necessary to have a geometric embedding of the graph. Often it suffices to have a *combinatorial embedding*. That is, we only know the order of the edges in the vertices' incidence lists and not the positions of the vertices and edges in the plane.

One thing that makes the work with nearly-planar graphs so much more difficult than with planar graphs is that the Jordan curve theorem is not applicable in the presence of even one crossing. And many proofs for the planar case rely — often without even mentioning it — on this theorem.

There are several polynomial combinatorial problems that can be solved asymptotically faster on planar graphs than on general graphs. There are also NP-hard problems that are easier to approximate on planar graphs or even become polynomial.

One problem that can be solved asymptotically faster on planar graphs is the maximum flow problem which we will discuss in detail below. Another such problem is the *maximum matching* problem. It is solvable in time $O(\sqrt{n} m)$ in general graphs and in time $O(n^{\omega/2})$ in planar graphs [MS04]. Here ω is the exponent of the best known matrix multiplication algorithm. Since it is known that $\omega < 2.5$, the complexity of planar maximum matching is in $O(n^{1.25})$.

The maximum cut problem for general graphs is NP-hard. For planar graphs, however, it is polynomially solvable [Had75].

Another interesting case is the *edge-disjoint paths problem*. Here we are given a *supply graph* G and a *demand graph* H . The latter graph consists of non-incident edges on a subset of $V(G)$. We have to decide whether there is a set of edge-disjoint cycles in $G + H$ such that each cycle contains exactly one edge in H and every edge in H is contained in a cycle. This problem is polynomially solvable if $G + H$ is planar and Eulerian [Sey81]. For non-planar or non-Eulerian graphs, in contrast, the problem is NP-hard [MP93]. The problem is even solvable in linear time if the graph $G + H$ can be embedded into the plane such that H lies completely in one face of G 's embedding [WW95, HW04]

1.4 Maximum Flow History and Background

The earliest references to the maximum flow problem date from the 1950s. The original motivation seems to have been a Cold War thought experiment on the Russian railway system [Sch05]. In 1956 Ford and Fulkerson [FF56] described the first algorithm, an *augmenting path algorithm*. This algorithm, however, was not strongly polynomial. It did not even terminate unless the capacities were restricted to rational numbers. In 1972 Edmonds and Karp [EK72] published an improved augmenting path algorithm, which had a strongly polynomial running time in $O(m^2n)$. Here, as usual, we denote by n the number of vertices and by m the number of arcs. At around the same time, Dinic [Din70] found a strongly polynomial algorithm of a different type. His algorithm was the first in the family of *blocking flow algorithms* and achieved a running time in $O(mn^2)$. In the 1970s and 80s, better algorithms for both approaches were found. Table 1.1 shows the currently best known algorithm for each family. See [Gol98] for a complete listing.

In 1988 Goldberg and Tarjan [GT88] laid the foundation for a new family of maximum flow algorithms, the *preflow push algorithms*. They achieved a basic running time of $O(n^3)$. Using intricate data structures they improved this to $O(mn \log \frac{n^2}{m})$. Other authors found even faster preflow push algorithms (see Table 1.1).

For special cases faster algorithms are known than for general instances. For example, often instances with bounded integral capacity can be solved faster. We do not give details here. Of interest to us, however, is the planar case. We will focus on this case in the rest of Section 1.4.

In 1965 Berge [BGH65] described the first algorithm especially designed for planar networks. More precisely the algorithm works on s - t -planar networks. These are planar networks that are embedded such that the source s and the sink t lie on the border of the same face. Berge's algorithm has a worst case bound of $O(n^2)$. In 1979 Itai and Shiloach [IS79] improved Berge's approach to $O(n \log n)$. Hassin showed in 1981 [Has81] that the s - t -planar maximum flow problem can be solved by shortest path computation in the planar dual graph. This computation can be done in time $O(n)$ using the algorithm by Klein et. al. [KRRS94].

In 1997 Weihe [Wei97] found the first algorithm for the general planar case, also with a running time bound of $O(n \log n)$. In 2006 Borradaile and Klein [BK06] described a simpler algorithm for this case with the same time bound.

algorithm type	author(s)	bound	instances
augmenting paths	Sleator & Tarjan [ST83]	$O(mn \log n)$	all
	Weihe [Wei97], Borradaille & Klein [BK06]	$O(n \log n)$	planar
	Itai & Shiloach [IS79]	$O(n \log n)$	s - t -planar
dual shortest paths	Klein et. al. [KRRS94]	$O(n)$	s - t -planar
blocking flows	Galil & Naamad [GN80]	$O(mn \log^2 n)$	all
preflow push	Goldberg & Tarjan [GT88]	$O(mn \log(n^2/m))$	all

Table 1.1: Best known worst case bounds of several maxflow algorithm types.

1.5 Notation

We will be working on digraphs only. A *directed graph* (or *digraph* for short) is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of *arcs* or *edges*. Each arc or edge is a pair (v, w) such that $v, w \in V$. It is *oriented* from its *tail* to its *head*. Whenever we are not interested in an edge's orientation we use the corresponding *undirected edge* $\{v, w\} := \{(v, w), (w, v)\}$.

By definition there can be no multiple or parallel arcs. And we only consider graphs without loops, that is without arcs of the form (v, v) .

By $V(G)$ we mean the vertex set of G and by $E(G)$ the edge set. For a vertex set U we denote by $\Gamma^+(U)$ the set of outgoing arcs of U and by $\Gamma^-(U)$ the set of incoming arcs of U . The *out-degree* of a vertex v is then $\delta^+(v) := |\Gamma^+(v)|$ and the *in-degree* is $\delta^-(v) := |\Gamma^-(v)|$. Finally the *degree* of v is $\delta(v) := \delta^+(v) + \delta^-(v)$.

A *path* from v to w (or a v - w -path for short) is a connected subgraph P such that v and w have degree 1 and all other vertices on P have degree 2. A path has an *orientation*. Namely a v - w -path is oriented from v to w . We say the path is *directed*, if all its arcs are oriented conforming to the path's orientation. A *cycle* is a connected subgraph in which all vertices have degree 2.

It is sometimes necessary to consider only intervals of paths or cycles. If P is a path and v lies before w on P then we denote by $P|_{[v,w]}$ the *interval* of P between v and w . That is, $P|_{[v,w]}$ is the subgraph of P that contains the v - w -subpath contained in P , including v and w . If, for some reason, we want to exclude one or both of the end vertices from the subpath then we use the corresponding open intervals, for example $P|_{]v,w]} = P|_{(v,w]}$

or $P|_{]v,w[} = P|_{(v,w)}$.

A *directed cut* (X, Y) is the set of all arcs (u, v) such that $u \in X$ and $v \in Y$. The corresponding *undirected cut* is $\{X, Y\}$ the set of all edges $\{u, v\}$ such that $u \in X$ and $v \in Y$. A *u - v -cut* $(X, V \setminus X)$ is a cut such that $u \in X$ and $v \in V \setminus X$.

A *tree* is a connected subgraph which has more vertices than edges. We say *leaf* for a vertex of degree 1 in a tree. A tree may have a determined vertex called *root*. If the tree is a directed graph and has a root v and its edges are oriented such that the root is reachable from each leaf by a directed path then we say it is an *in-tree*. If each leaf is reachable from the root by a directed path then the tree is an *out-tree*.

A *forest* is a graph that consists of disjoint trees. Analogously, an *in-forest* consists of in-trees and an *out-forest* of out-trees.

Sometimes it is useful to have a short notation for the trivial composition of two graphs. We write $P+Q$ to mean the graph $((V(P) \cup V(Q), E(P) \cup E(Q))$.

The input to our problem is a *network* (G, c, s, t) . That is a directed graph G with a capacity $c(e)$ on each edge $e \in E(G)$ and a source s and a sink t . We sometimes call the instance just G for short.

An *s - t -flow* in a network is a function f that assigns a real number to each arc in the graph and satisfies the following constraints:

$$\begin{aligned} \forall e \in E(G) : & \quad 0 \leq f(e) \leq c(e) \quad \text{and} \\ \forall v \in V(G) \setminus \{s, t\} : & \quad ex_f(v) := \sum_{e \in \Gamma^-(v)} f(e) - \sum_{e \in \Gamma^+(v)} f(e) = 0. \end{aligned}$$

If, instead of the latter equation, f only satisfies $ex_f(v) \geq 0$ for all $v \in V \setminus \{s\}$ then we say f is a *preflow*.

The *flow value* of f is defined as $ex_f(t)$. A *maximum s - t -flow* is a flow of maximum value. The capacity of a directed cut is the sum of its arcs' capacities:

$$c(X, Y) := \sum_{e \in (X, Y)} c(e).$$

The *capacity of a directed path*, on the other hand, is the minimum of all its arcs' capacities.

1.6 The Model

As said above, we only consider simple directed graphs with neither parallel arcs nor loops. This is no restriction since, for the purpose of maximum flow, parallel arcs can be joined into one arc. And loops cannot change the flow value at all.

If the graph contains more than one connected component then for the maximum s - t -flow computation we may disregard any components that contain neither s nor t . Moreover if s and t are in different connected components then the maximum flow value is 0. Thus we will restrict ourselves to connected graphs.

In order to take advantage of the geometric or topological properties of nearly-planar instances, it is often necessary to consider an embedded graph. For theoretical considerations it is sufficient to have a *combinatorial embedding*. That is an embedding that only specifies the order of the arcs in any vertex' incidence list. It does *not* specify the absolute or relative positions of vertices in the plane.

In practice, however, we use geometrically embedded instances for our computational experiments. Here we are given the position of each vertex in the plane and assume that the edges are straight lines. For this type of embedding we require that no two vertices share the same position and that a straight line edge is disjoint to all vertices except at its endpoints.

1.7 Basic Facts about Maximum Flows

We have seen that the maximum flow problem has been well investigated. This research has generated a number of central insights which we will reproduce here.

We begin with an optimality criterion.

Theorem 1.7.1 (Max-Flow-Min-Cut Theorem [FF56, EFS56]). *The value of a maximum s - t -flow in a network equals the minimum capacity of an s - t -cut in the same network.*

Theorem 1.7.2 (Flow Decomposition Theorem [FF62]). *A flow in a network can be decomposed into a family of cycles and a family of paths. That means that there is a weight for each cycle and path such that the total weight of all cycles and paths on any edge equals the flow on that edge. Moreover, if the flow is integral then the weights can be chosen integral, as well.*

Since the cycles do not attribute anything to the flow value we have the following result.

Corollary 1.7.3. *Any network has a maximum s - t -flow that can be decomposed into paths only.*

This in turn motivates the following algorithm.

Listing 1 The Ford-Fulkerson Algorithm.

while there is an s - t -path of positive capacity **do**
 Set the weight for this path maximally.
 Subtract the path from the network.
end while
 The union of all paths found is a maximum flow.

In order to better describe this algorithm we present some commonly accepted notation. For each arc $e = (u, v)$ in the graph we also have a *reverse* arc $\overleftarrow{e} = (v, u)$. The *residual capacities* c_f with respect to f are defined as follows:

$$\forall e \in E(G) : \quad c_f(e) = c(e) - f(e) \quad \text{and} \quad c_f(\overleftarrow{e}) = f(e). \quad (1.1)$$

The *residual network* of a flow f is then the network $G_f := (G, c_f, s, t)$.

An arc e is called *augmenting* if $c_f(e) > 0$, and \overleftarrow{e} is called *augmenting* if $c_f(\overleftarrow{e}) > 0$. An arc or reverse arc is *saturated* if it has zero residual capacity. A path P is called *augmenting* if all of its arcs are augmenting. P is called *saturated* if at least one arc is saturated. The *bottleneck capacity*, or just *capacity*, of P in G_f is $\delta = \min\{c_f(e) : e \in P\}$. The operation “augment f by P ” is meant to add P ’s bottleneck capacity to the flow on all arcs on P , that is

$$\forall e \in P : \quad f(e) = f(e) + \min\{c_f(e) : e \in P\}.$$

Note that by changing f , the residual capacities c_f are changed, as well.

Listing 2 The Ford-Fulkerson Algorithm (2nd attempt).

Set $f := 0$.
while there is an augmenting s - t -path P in G_f **do**
 Augment f by P .
end while
 f is a maximum flow.

Obviously the algorithm proceeds in *iterations*. In each iteration $i = 1, 2, 3, \dots$ we look for an augmenting path in the residual instance G_i and then augment f along this path. Here the graph in the first iteration is just $G_1 := G$. The augmentation in any iteration results in the next residual instance G_{i+1} .

1.7.1 Limitations of the Ford-Fulkerson Algorithm

The Ford-Fulkerson Algorithm is not strongly polynomial, and on instances with irrational capacities, it may not terminate at all. Ford and Fulkerson [FF62] give examples of instances that demonstrate these shortcomings of their algorithm. We present these instances here because they can serve as a plausibility check for new algorithmic ideas.

The Ford-Fulkerson Algorithm is not strongly polynomial. In case of ambiguity, the algorithm does not specify which of several possible paths to choose for augmentation. Given the instance in Figure 1.7.2 we choose to augment alternately the paths (s, u, v, t) and (s, v, u, t) . This way we have to make $2C$ augmentations. Thus the running time depends not only on the problem size $m + n$ but also on the values of the capacities. Since the coding size of the capacities in this instance is in $O(\log C)$, the running time is in fact exponential in the input size.

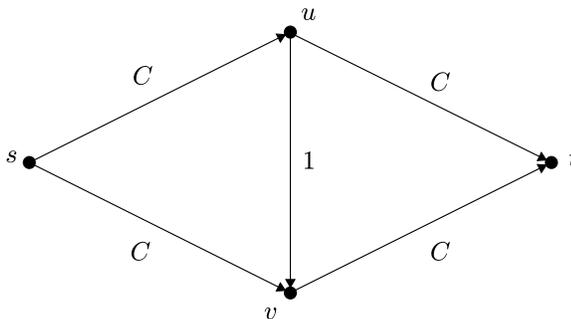


Figure 1.7.2.

The Ford-Fulkerson Algorithm may not terminate. The idea is to force the algorithm to calculate an infinite sequence $\{a_n\}$ using the recursion formula $a_{n+2} = a_n - a_{n+1}$ beginning with $a_0 = 1$ and $a_1 = r$. If we set $r = \frac{\sqrt{5}-1}{2}$ then we have $a_n = r^n$.

Ford and Fulkerson's example is quite complicated. Zwick [Zwi95] has given simpler and smaller examples to accomplish the same. We present one of these examples here. Figure 1.7.3 shows the instance and the four paths we are going to use.

The capacities of e_1 , e_2 and e_3 are $c(e_1) = a_0$, $c(e_2) = a_1$ and $c(e_3) = 1$ respectively. All other capacities are equal to some $C \geq 4$. Thus a maximum flow in this network has value $4C + 1$.

We begin by augmenting along P_0 . The vector of the three capacities is then

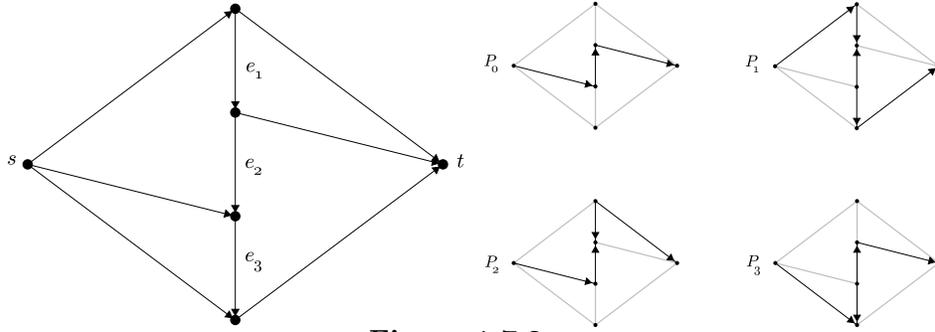


Figure 1.7.3.

$(a_0, a_1, 0)$. By augmenting the paths P_1, P_2, P_1, P_3 in this order we advance from $(a_n, a_{n+1}, 0)$ to $(a_{n+2}, a_{n+3}, 0)$ in the sequence as shown here:

$$\begin{array}{c}
 (a_n, a_{n+1}, 0) \\
 \downarrow P_1 \\
 (a_{n+2}, 0, a_{n+1}) \\
 \downarrow P_2 \\
 (a_{n+2}, a_{n+1}, 0) \\
 \downarrow P_1 \\
 (0, a_{n+3}, a_{n+2}) \\
 \downarrow P_3 \\
 (a_{n+2}, a_{n+3}, 0)
 \end{array}$$

Since $r < 1$ the values $a_n = r^n$ approach 0 and the telescopic sum $\sum_{i=0}^n a_i$ approaches $a_0 = 1$ for $n \rightarrow \infty$. The algorithm computes a flow whose value is at most $2 \sum_{i=0}^{\infty} a_i + 1 = 3$. As $C > 4$ the residual capacities on the arcs incident to s and t are never smaller than 1. Thus the residual capacity of one of the edges e_1, e_2 and e_3 is the bottleneck in any iteration and the augmentation scheme using P_1, P_2, P_1, P_3 can go on infinitely.

1.8 Problem Variants

Undirected graphs. In this thesis we focus on the maximum flow problem in directed graphs. Of course, this problem can also be considered in *undirected* graphs. In undirected graphs the flow is not restricted by edge orientations. For an edge $\{u, v\}$ it may flow from u to v or in the reverse direction or in both directions. However, the sum of the flows in either direction may not exceed the capacity of the edge.

Ford and Fulkerson [FF62] give a reduction from the undirected to the directed case. This reduction is done by replacing each edge by a small graph as shown in Figure 1.8.4.

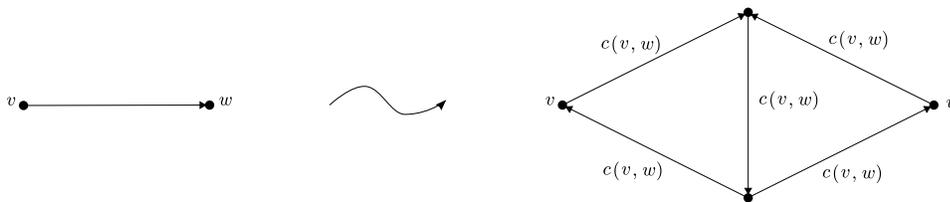


Figure 1.8.4.

Obviously this transformation of the graph can be done in time linear in the graph size $m + n$. The resulting graph is directed and of comparable size to the original graph.

Feasible Flows. We can generalize the maximum s - t -flow problem by introducing a *balance* $b(v)$ for each vertex v . Instead of requiring flow conservation at each vertex we then require that the difference of in-flow and out-flow of v is equal to $b(v)$:

$$\forall v \in V(G) \setminus \{s, t\} : \sum_{e \in \Gamma^-(v)} f(e) - \sum_{e \in \Gamma^+(v)} f(e) = b(v).$$

This is sometimes called the *feasible flow problem* or just *b-flow problem*. In this setting we usually do not have a source s and a sink t since all vertices with negative balance are sources and all vertices with positive balance are sinks. The problem then becomes a decision problem instead of an optimization problem. That is, we ask if a feasible flow for this problem exists. There is an obviously necessary condition for the existence of a solution: A solution can only exist if the sum of all balance values is zero.

This problem can be reduced to the special case $b \equiv 0$ as follows. We introduce a *super source* \bar{s} and a *super sink* \bar{t} and connect each vertex v with negative balance with \bar{s} by an edge with capacity $-b(v)$. Analogously we connect each vertex w with positive balance with \bar{t} by an edge with capacity $b(w)$. See also Figure 1.8.5.

A solution to the problem is feasible if, and only if, all edges incident to the super source or to the super sink are saturated by the flow.

In general, this reduction does not increase the complexity of the problem. However, the transformation may make a planar graph non-planar, or in-

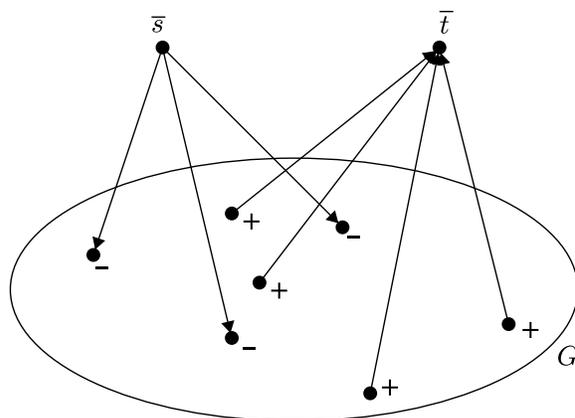


Figure 1.8.5.

crease the non-planarity of a nearly-planar graph. Therefore it has to be used with caution in our context.

Multiple sources or sinks. In the same way we can also solve the problem if we have more than one source or sink. We connect all sources to the super source and all sinks to the super sink. The upper bounds on the edge capacities are now infinite. Again this reduction is of limited use for planar or nearly-planar instances.

Lower capacity bounds. In this thesis we only consider instances with upper bounds on the flow on the edges. The lower bounds are implicitly zero. Thus the feasible interval for flow on an edge e is $[0, c(e)]$. This is no restriction, however. We can solve an instance with lower and upper bounds by computing the solutions to two derived instances with only upper bounds. See [AMO93] for the complete exposition of this approach. Since it relies on the feasible flow problem outlined above, it is not thoroughly usable for planar or nearly-planar instances.

Chapter 2

Computational Investigation

Throughout this thesis, we will use computational experiments to examine the running time of the algorithms from a practical perspective. This is necessary since first of all, the theoretical worst case bound of any algorithm rarely coincides with the running time observed on real world instances. And second, there are times when we can only derive a trivial running time bound for an algorithm. In this case the algorithm's real performance (good or bad) is revealed only by experiments.

2.1 Benchmark Algorithms

We run our tests on instance classes specially designed for the purpose of this thesis. Since the properties of these instances are not fully under control it is necessary to gain some understanding of them before using the instances in our tests. Otherwise it would be difficult to correctly interpret the results of our algorithms. For example, a small running time may indicate a fast algorithm or a trivial instance.

In order to get a feeling for the new instance classes we first run a set of well-known algorithms on them and observe the behaviour of these algorithms by logging some characteristic figures such as running time, operation counts and individual characteristics of each algorithm.

Our benchmark algorithms are

- **GT**: the Goldberg-Tarjan algorithm with FIFO rule [GT88].
- **DM**: the Goldberg-Tarjan algorithm with gap relabelling and highest level rule [DM89].

- **AO**: the shortest augmenting path algorithm with distance labels and gap heuristic by Ahuja and Orlin.
- **ST**: like AO but using the dynamic trees data structure by Sleator and Tarjan.

2.2 Real-world Data

We evaluated freely available road data sources on the Internet. We first considered the Tiger/Line® data from 2005 in the second edition [Tig05]. However, this data treats overpasses and underpasses (crossings) as junctions (vertices). Thus important information is lost and the data is unusable for our purposes.

Next we looked at the Canadian NRNC1 road data. This contains data “for all non-restricted use roads in Canada, 5 meters or more in width, drivable and no barriers denying access” [Geo06]. Here a road crossing is stored as two or more non-connected vertices at the same map position. Each of these vertices then presents one crossing edge of the crossing. However, an embedding in this form does not conform with our assumptions on the input data which prohibit vertices that share the same position. Therefore we do not use this data set, either.

2.3 Instance Classes

We are interested in nearly planar graphs. In real life there are plenty of such graphs. For example geographical networks such as streets, roads and railway tracks. But for a relevant computational study we need large graphs and many of them. It is out of the scope of this thesis to collect a sufficiently large data set of geographic networks from a variety of data sources. Therefore we generate our own instances.

We now give a short overview of the instance classes and then describe each with more detail.

The first three generators build planar graphs in different ways:

- **subdivision**: Successively subdivides faces in an initial square.
- **squaregrid**: Makes a planar grid of squares.
- **triangulated**: Successively triangulates faces in an initial triangle.

In the planar graph they then insert some edges which, in general, cannot be drawn without crossing existing edges. We call these edges *non-planar* edges. All edges — planar and non-planar — have independent and uniformly distributed random capacities. The source and the sink are chosen on the border of the graph. Thus all instances are s - t -embedded.

For each class of instances we created 100 graphs with $n = 1,000i$ for $i = 1, 2, 3, \dots, 100$ and 70 graphs with $n = 100,000 + 10,000i$ for $i = 1, 2, 3, \dots, 70$. The upper bound of 800,000 vertices per graph was dictated by the limited physical memory of our test computer.

Pre-processing. After creating the instance we performed some simplifications on them to reduce the computation time. We deleted all vertices that are not reachable from s or from which t is not reachable by an augmenting path. This reduces the instance size by about 10%. However, some instance classes are more affected by this than others. As we can see in Figure 2.3.1 the instances of class “subdivision” are rarely larger than 600,000 vertices while in “triangulated” we have instances of almost 800,000 vertices.

2.3.1 Properties of the Instances

Since we have $m \in \Theta(n)$ for nearly planar graphs, it makes no sense to look at the density of the graphs, which is usually defined as $\text{dens}(G) = \frac{m}{\binom{n}{2}}$. Instead we consider the *planar density* which we define as $\text{dens}_p(G) = \frac{m}{3n-6}$.

As we can see in Figure 2.3.1, the planar density depends only on the instance class, not on the number of vertices.

In Figure 2.3.2 we plot the number of crossings of each instance. As we can see the number of crossings for the instance set “triangulated” grows faster than that of the other two. At first glance, it seems as if all instances of “subdivision” and “squaregrid” have no crossings.

Therefore we show the same data in Figure 2.3.3 in *double logarithmic* (or *log-log*) scale. In this kind of chart both ordinate and abscissa are in logarithmic scale. This is especially useful for asymptotical considerations because a curve with $y \in \Theta(x^a)$ in log-log scale becomes a line of slope a .

We will see that the behaviour of our benchmark algorithms is different for each class of instances. The number of crossings is not sufficient to explain these differences. Therefore we will look at the distribution of vertex degrees.

First, in Figure 2.3.4, we see that the maximum degree for “squaregrid” is

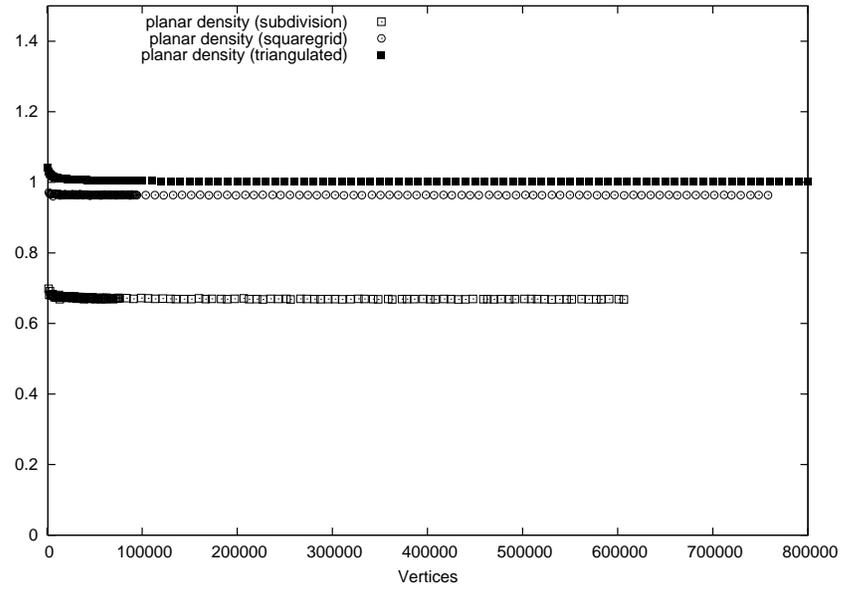


Figure 2.3.1: The planar density of the different instance classes.

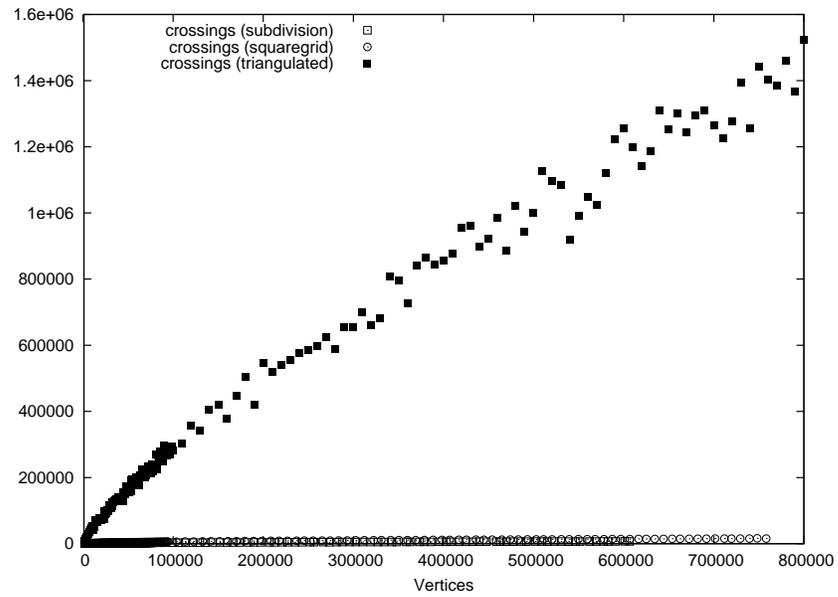


Figure 2.3.2: The number of crossings of the different instance classes.

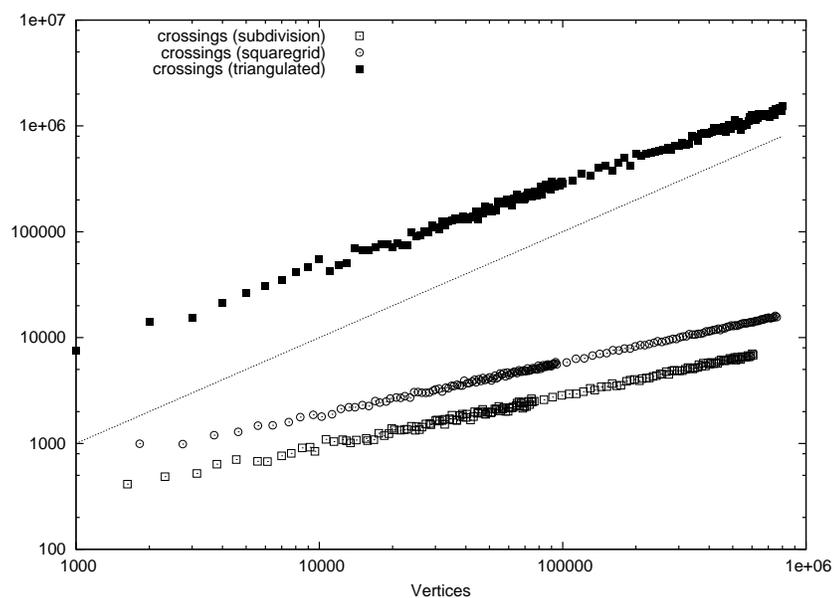


Figure 2.3.3: The number of crossings of the different instance classes in log-log scale. We also plotted $y = x$ for comparison.

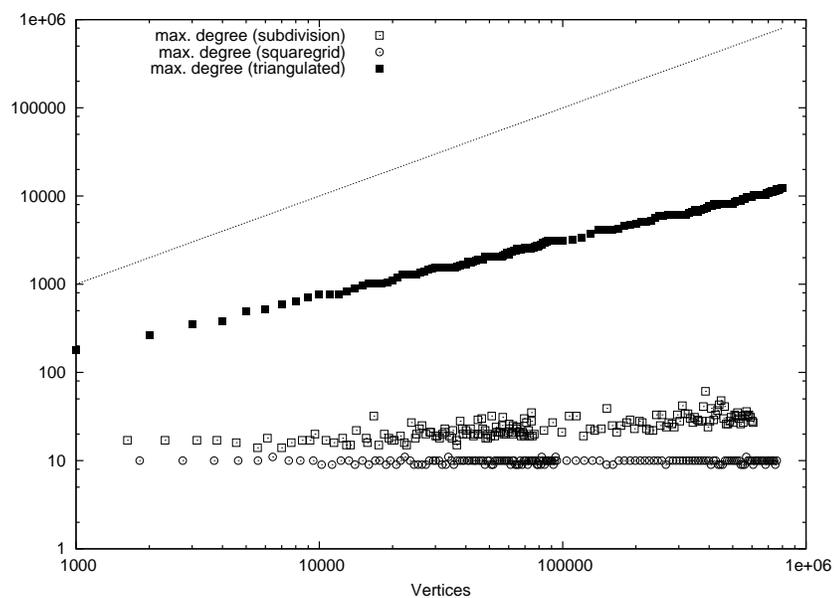


Figure 2.3.4: The maximum vertex degree of the different instance classes in log-log scale. We also plotted $y = x$ for comparison.

constantly 10. For “subdivision” it grows slowly but does not reach 100 even for the largest instances. For “triangulated” it grows faster but still sublinear.

Figure 2.3.5 then shows the complete distribution of the vertex degrees. We see that the distributions are hugely different for each instance class.

2.4 Results

How do the benchmark algorithms perform on our test instances? We show the results with combinatorially sorted incidence lists in Figure 2.4.6 and with unsorted incidence lists in Figure 2.4.7. For each instance class the preflow push algorithms (GT and DM) are faster than the augmenting path algorithms (AO and ST). On “subdivision” and “squaregrid” ST is the slowest, even slower than AO. While on “triangulated” AO is slower than ST. This is surprising since the worst case running time of AO is $O(n^2m)$ and thus higher than that of ST with $O(nm \log n)$.

Berge’s uppermost path algorithm for planar graphs (compare Section 3) takes great advantage of sorted incidence lists so it is of interest to see whether this also holds for our benchmark algorithms (Figure 2.4.7). For unsorted incidence lists we ran only two algorithms in order to save time. More accurately, we ran AO as representative of the augmenting path algorithms and DM representing the preflow push algorithms. On “subdivision” and “squaregrid” all algorithms seem to run *more slowly* by about 5% in case of sorted incidence lists. On “triangulated”, however, DM gains about 10% and AO even 20% compared to unsorted incidence lists.

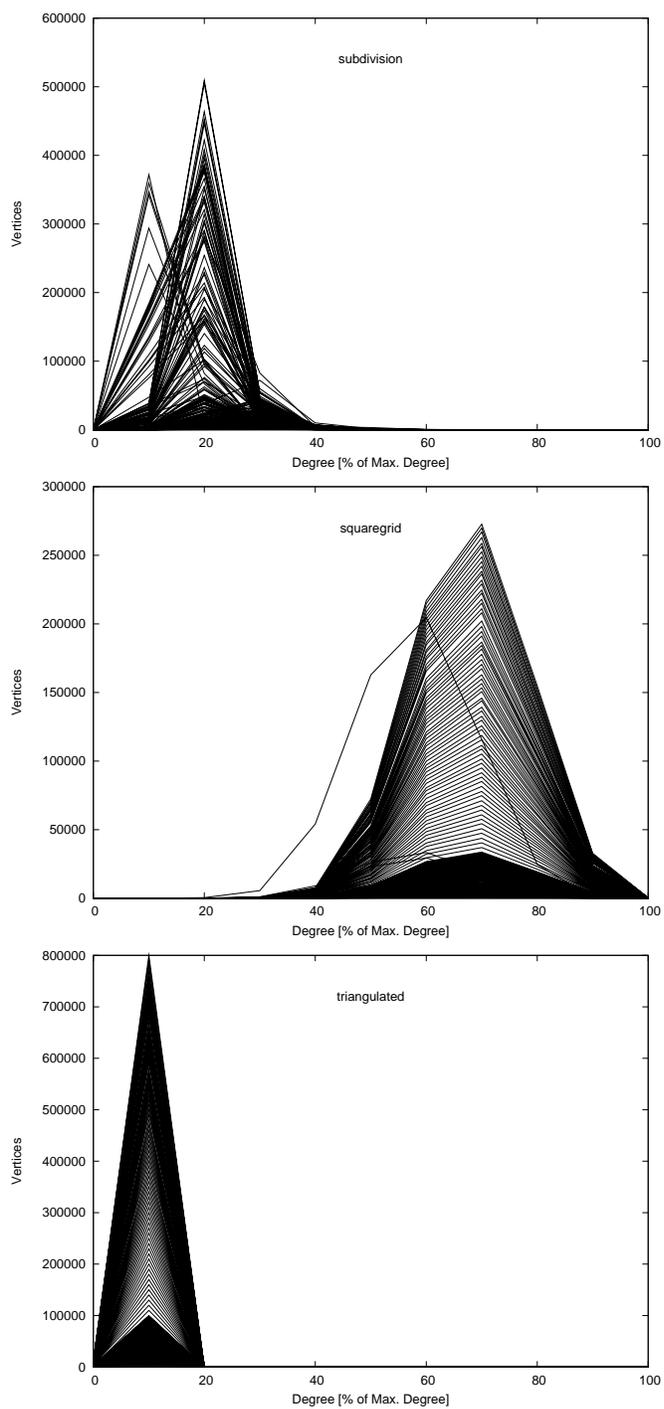


Figure 2.3.5: Distribution of vertex degrees for all instances.

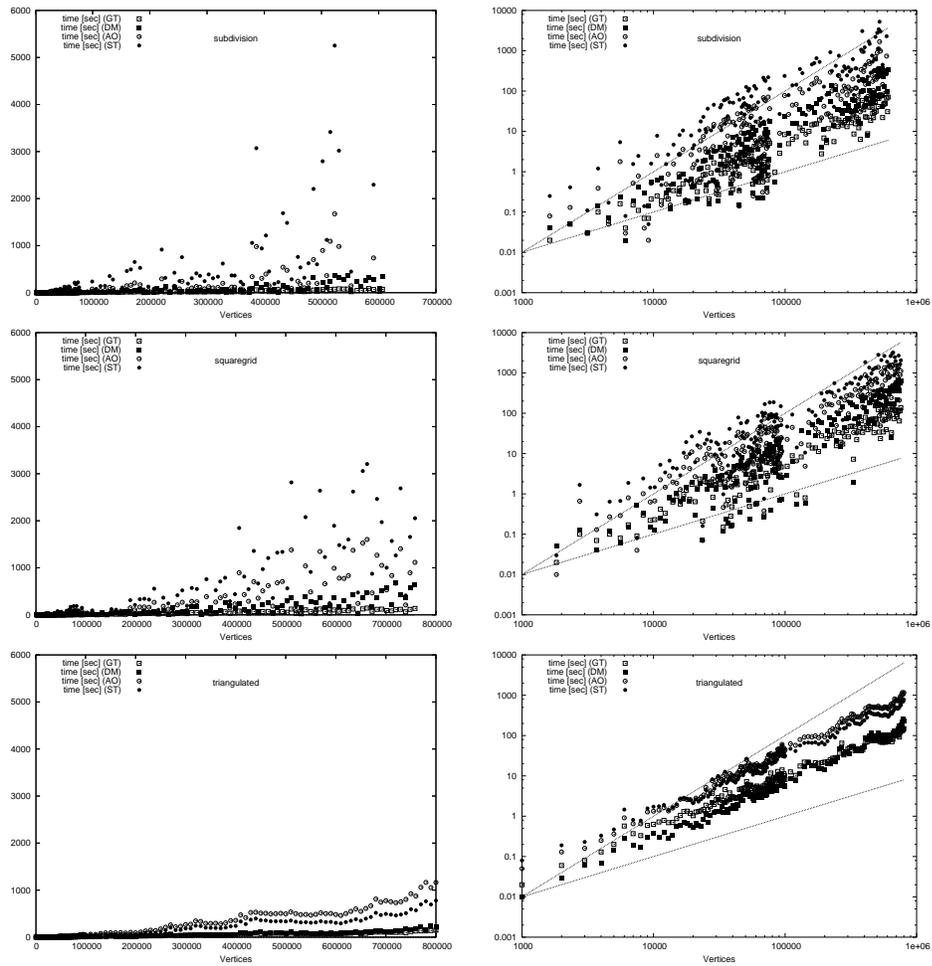


Figure 2.4.6: Running times of all benchmark algorithms using sorted adjacency lists. Plots in the left column have linear scales. On the right hand side we present the same data in log-log scale. We also plotted $y = x$ and $y = x^2$ for comparison.

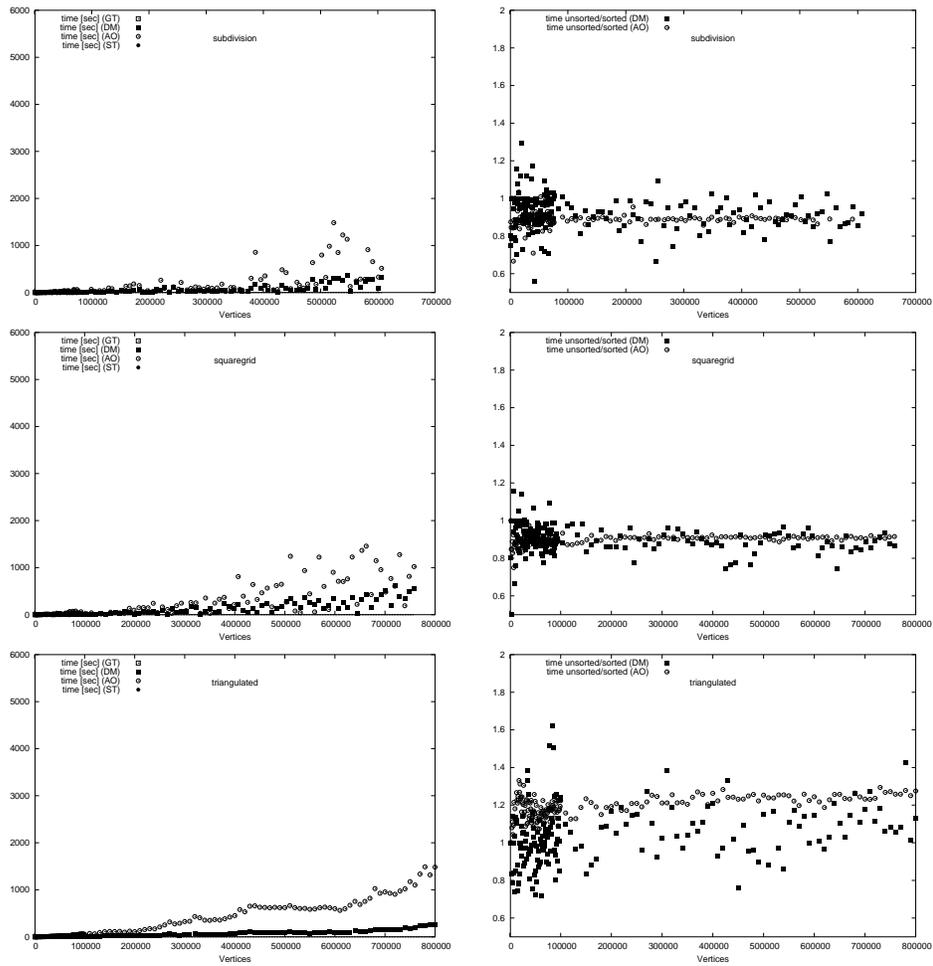


Figure 2.4.7: Left column: Running time of AO and DM using unsorted adjacency lists. Right column: ratio unsorted/sorted time.

Chapter 3

The Uppermost Path Algorithm and Some Variants

Evidently, the uppermost path algorithm [BGH65] is the version of the augmenting path algorithm with the best asymptotic running time for s - t -planar instances. It is also the simplest maximum flow algorithm for the planar case. In its implementation with implicit residual capacities it runs in time $O(n \log n)$ worst case [IS79]. For general planar instances the algorithms by Weihe [Wei97] and Borradaile and Klein [BK06] have the same worst case running time but are much more complex. We will therefore focus on nearly planar instances such that the source and the sink lie on the border of the same face. We call these instances s - t -embedded. That is a natural generalization of “ s - t -planar”.

Overview of this chapter. We first describe a generic augmenting path algorithm, then formulate the uppermost path algorithm, and finally develop modifications that can solve non-planar instances. Thereby we develop a generic framework to describe augmenting path algorithms. And we find a simple variant of the uppermost path algorithm which has, on our test instances, an empirical running time comparable to the currently best known algorithms.

3.1 A Generic Augmenting Path Algorithm

Augmenting path algorithms are algorithms that successively augment along s - t -paths with non-zero residual capacities until all those paths are saturated. An example of this class of algorithms is the Edmonds–Karp–Algorithm, which chooses the shortest augmenting path in each iteration. Even Dinic’ Algorithm can be interpreted as an augmenting path algorithm (cf. [AMO93]).

We need to introduce some more notation. We shall partition the algorithm’s running time in iterations: An *iteration* consists of one *search* for an augmenting path and one *augmentation* along a path if one is found. More accurately, an iteration is a maximal interval in the algorithm’s running time that contains exactly one search and one augmentation and ends after this augmentation.

In the search phase of an iteration an augmenting s - t -path P is built incrementally. At any time P is an augmenting s - v -path. Only at the end of the search phase, and only if an augmenting s - t -path has been found, is the *current vertex* v equal to t . For the current vertex we also maintain the *current edge* which is the edge which will be scanned next in the search.

To formulate the generic augmenting path algorithm in Listing 3, we use the following set of procedures. We will later implement these procedures as necessary for exemplary augmenting path algorithms.

<code>active(v)</code>	Returns <code>true</code> iff the algorithm should scan edges in v ’s incidence list.
<code>admissible(v, w)</code>	Returns <code>true</code> iff the algorithm may add (v, w) to its current partial path.
<code>current_arc(v)</code>	This gives the current arc in the incidence list of v . The algorithm scans this arc before calling <code>advance_arc(v)</code> .
<code>advance_arc(v)</code>	This advances the current arc of v to the next edge in the incidence list that should be scanned by the algorithm.
<code>advance(P, e)</code>	This procedure appends the edge e to the end of the partial path P . Afterwards P is augmenting.
<code>retreat(P)</code>	This procedure is used by the algorithm when the current vertex becomes inactive. It modifies P . Afterwards P is augmenting.
<code>augment(P)</code>	This procedure augments the augmenting path P by the maximum possible flow on P .

repair(P) This procedure repairs the augmenting path P after an augmentation. Afterwards P is augmenting but does not necessarily reach t .

Listing 3 The Generic Augmenting Path Algorithm

```

1: Set  $P := s$ .
2: repeat
3:   Let  $v$  be the last vertex in the partial path  $P$ .
4:   while active( $v$ ) do
5:      $e := \text{current\_arc}(v)$ .
6:     if admissible( $e$ ) then
7:       advance\_arc( $v$ ).
8:       advance( $P, (v, w)$ ).
9:        $v := w$ .
10:      if  $v = t$  then
11:        augment( $P$ ).
12:        repair( $P$ ).
13:      end if
14:    else
15:      advance\_arc( $v$ ).
16:    end if
17:  end while
18:  if  $v \neq s$ . then
19:    retreat( $P$ ).
20:  end if
21: until  $v = s$ .

```

3.2 Dynamic Tree Implementations

One factor in running time is the time needed to augment along one augmenting path. In running time analysis this is typically bounded by the length of the longest possible path L , where $L \leq n$. Using a data structure called dynamic trees [ST83] we can reduce the asymptotic time needed for this to $O(\log L)$.

We give a brief description of that data structure: it represents an in-forest F with capacities. F is a subgraph of our input graph G . The orientation of an arc e in F may differ from e 's orientation in G .

To be precise, we have to distinguish capacities in the forest F and capacities in the graph: While an arc e is in F , its current capacity is maintained by the dynamic paths data structure and may differ from its capacity in G . When e is removed from F (at the latest at the end of the algorithm), its

capacity in G is updated to be the current capacity of e in F . However, we feel it is easier to understand the algorithm if we do not consider this distinction explicitly. We use $c(e)$ to refer to an arc e 's current capacity – whether in F or in the graph G .

To access elements of F , we have the following self-explaining functions: $\text{root}(v)$ and $\text{parent}(v)$. The unique (if any) v – $\text{root}(v)$ –path in F is called $\text{path}(v)$. And the function $\text{edge_after}(v)$ returns the arc after v on $\text{path}(v)$.

The data structure guarantees a running time of $O(\log L)$ where L is the maximum path length for each call to one of the above functions or the more complex functions below:

- $\text{link}(v, w)$: Link the tree root v to the vertex w which is in another tree. In that, w becomes the parent of v .
- $\text{cut}(v)$: Cut the forest arc between v and $\text{parent}(v)$. In that, v becomes the root of its containing tree.
- $\text{mincap}(v)$: Find the vertex w closest to $\text{root}(v)$ such that $(w, \text{parent}(w))$ has minimum capacity on $\text{path}(v)$.
- $\text{update}(v, \delta)$: Augment $\text{path}(v)$ by δ .

In Listing 4 we give a basic implementation of the generic algorithm's procedures using dynamic trees.

3.3 The Uppermost Path Algorithm

This is commonly attributed to Berge [BGH65]. The name “uppermost path algorithm” graphically describes the algorithm. In order to understand it, we assume that we are given an embedding of the s – t –planar graph G in which s is on the left side and t on the right side. In this setting the algorithm in each iteration finds the uppermost augmenting path in the residual graph.

In Listing 5 we describe the algorithm in terms of the Generic Augmenting Path Algorithm. We store augmenting paths and residual capacities implicitly using the dynamic trees data structure. Itai and Shiloach [IS79] give an implementation that also has implicit residual capacities but does not use dynamic trees. This is but a technical detail. Their analysis and their proof of correctness also apply to the algorithm in the form given here.

We denote the current edge of a vertex v by $\text{current_arc}(v)$. See Section 4.5 for a complete discussion.

In the following we give some properties of the uppermost path algorithm.

Listing 4 An implementation of the procedures using dynamic trees.

```

procedure advance( $P, (v, w)$ )
  link( $v, w$ )
end procedure

procedure retreat( $P$ )
  Let  $w$  be the last vertex on  $P$ .
  for all  $(v, w) \in F$  do
    cut( $v$ )
  end for
end procedure

procedure augment( $P$ )
   $w := \text{mincap}(v)$ 
  update( $v, c(w, \text{parent}(w))$ )
end procedure

procedure repair( $P$ )
   $w := \text{mincap}(t)$ 
  while  $c(w, \text{parent}(w)) = 0$  do
    cut( $w$ )
     $w := \text{mincap}(w)$ 
  end while
end procedure

```

Itai and Shiloach [IS79] show

Lemma 3.3.1. *Each edge is inserted at most once into the augmenting path and deleted at most once from the augmenting path.*

This immediately yields

Lemma 3.3.2. *The set of iterations in which a specific edge is in the current augmenting path is an interval in time.*

Now instead of looking at one specific edge we look at a specific vertex and the unique outgoing edge of that vertex in the augmenting path. From Lemma 3.3.1 and the definition of `advance_arc`(v) in Listing 5 follows

Lemma 3.3.3. *The outgoing arc for each vertex v revolves about v at most once. This means that the sequence of outgoing arcs (e_1, e_2, \dots) of v is ordered in a clockwise fashion. Here the outgoing arc e_i is the unique outgoing arc of v in P_i . See also Figure 3.3.1.*

Listing 5 An implementation of the uppermost path algorithm using dynamic trees.

```

procedure active( $v$ )
  if current_arc( $s$ ) is not defined then
    Let current_arc( $s$ ) be the uppermost arc at  $s$ .
  end if
  if current_arc( $v$ ) has completed a rotation about  $v$  then return true.
  else return false.
end procedure

procedure advance_arc( $v$ )
  Point current_arc( $v$ ) to the clockwise next arc in the incidence list
  of  $v$ .
end procedure

procedure advance( $P, (v, w)$ )
  link( $v, w$ )
  if current_arc( $v$ ) is not defined then
    Set current_arc( $w$ ) := ( $v, w$ ).
  end if
end procedure

```

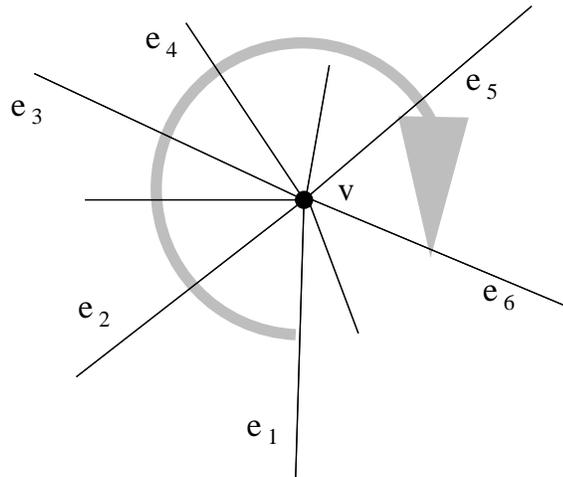


Figure 3.3.1: The sequence of outgoing arcs of v (e_1, e_2, \dots, e_6) is in clockwise order. The outgoing arc of v does not complete one revolution about v (gray arrow).

3.4 Completing a Sub-Optimal Solution

Berge's algorithm does not solve non-planar instances optimally. For an example see Figure 3.4.2. The maximum flow shown in this figure is not

found by the uppermost path algorithm. The algorithm finds just the upper of the two paths. The lower path is never considered because it requires augmenting an arc in reverse direction.

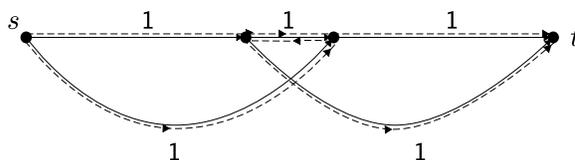


Figure 3.4.2: A non-planar network and two augmenting paths (dashed) that make up the maximum flow. Each path carries one unit of flow.

However, there is the possibility that for nearly planar instances the uppermost path algorithm already finds a “large portion of the maximum flow”. We do not know how to define this clearly. It is an intuition rather, which may be supported by experimental results or not. We do not think that this “large portion of the maximum flow” requires a large portion of the optimal flow value. For some instances the complexity might rather lie in finding one involved augmenting path of small capacity.

In order to get a clearer understanding of this intuition, we make some experiments. We run the uppermost path algorithm once on the instance. As we have seen, this gives a suboptimal solution in general. We now try to discover how much of the optimum solution has been found by this first algorithm. As noted the flow value is a poor criterion. Therefore we complete the suboptimal solution in different ways and measure the intermediate solution quality by the time needed to complete it. An intermediate (sub-optimal) solution that needs no time to complete it is already optimum. On the other hand, an intermediate solution that needs as much time (or even more time) to complete than solving the whole instance from scratch is bad. The next two sections hold the results.

3.4.1 Repeated Application

Here we repeat the uppermost path algorithm until the solution is optimum. We call one such application a *phase*. Figure 3.4.3 shows the running time of this strategy compared to two benchmark algorithms. For the instance set “triangulated”, the new strategy is asymptotically faster than AO and DM. For the other two instance sets, it is asymptotically comparable to AO and DM. The absolute running time is in the same range as that of AO for all instances. So in practice this seems to be a competitive algorithm for nearly planar instances.

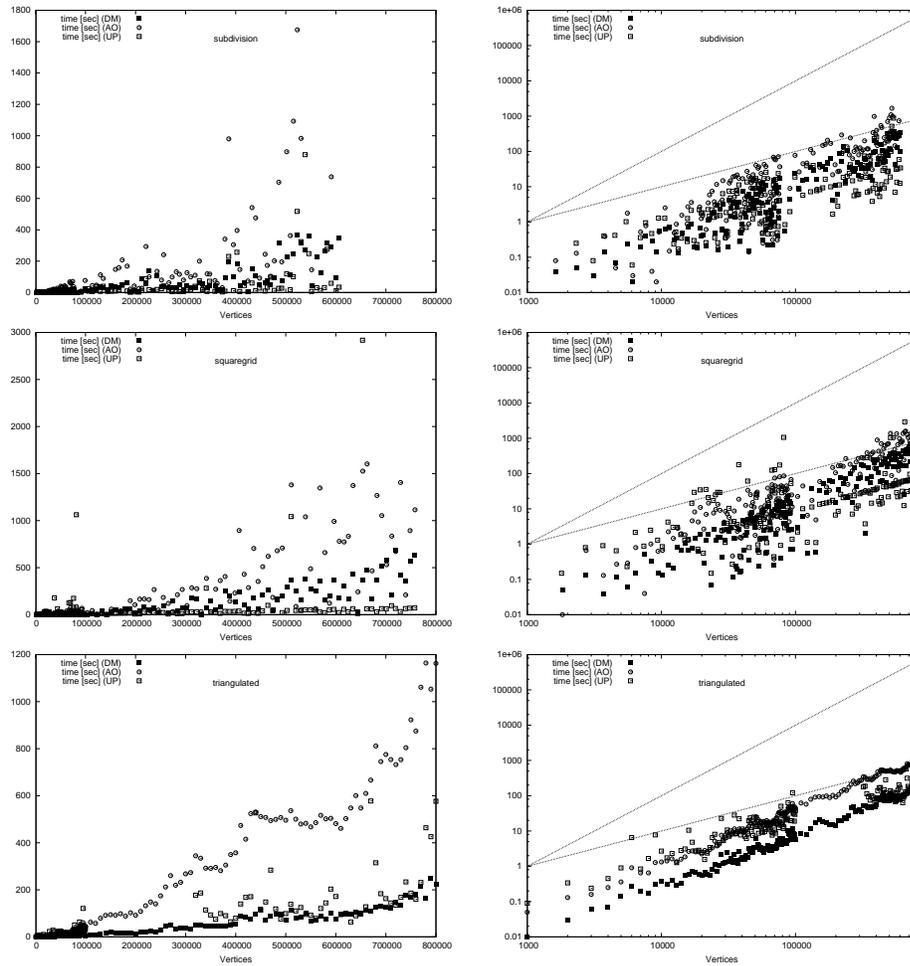


Figure 3.4.3: Comparison of running time of the repeated application of the uppermost path algorithm (UP) with the time needed to solve the instance from scratch using benchmark algorithms AO and DM. In the double logarithmic charts (right column) we also plot functions proportional to $y = x$ and $y = x^2$, respectively.

How many phases does the algorithm use? Since in instances without crossings we need only one phase, we may suspect that the number of phases depends mainly on the number of crossings. Therefore we plot the number of phases against the number of crossings in Figure 3.4.4.

In fact, it seems that the number of phases is linear in the number of crossings. However, these results only give an indication. In order to support this claim we would need a whole lot of more data or a theoretical proof. But the processing of the required amount of data is out of the scope of this thesis and we did not find a proof, either.

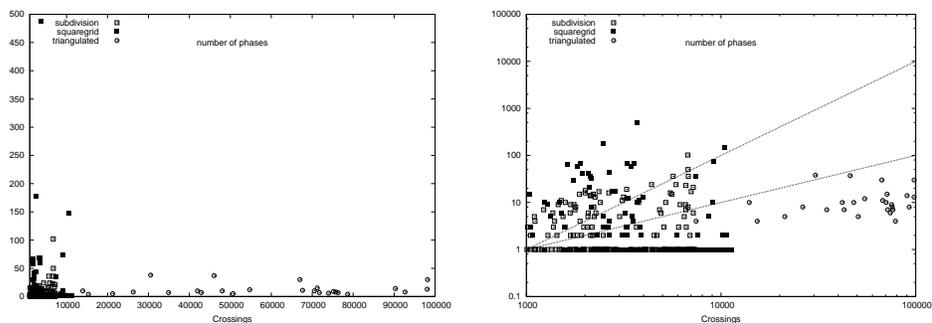


Figure 3.4.4: The left chart shows the number of phases and the right chart shows the same data in log-log scale. Note that in horizontal direction we chart the number of crossings.

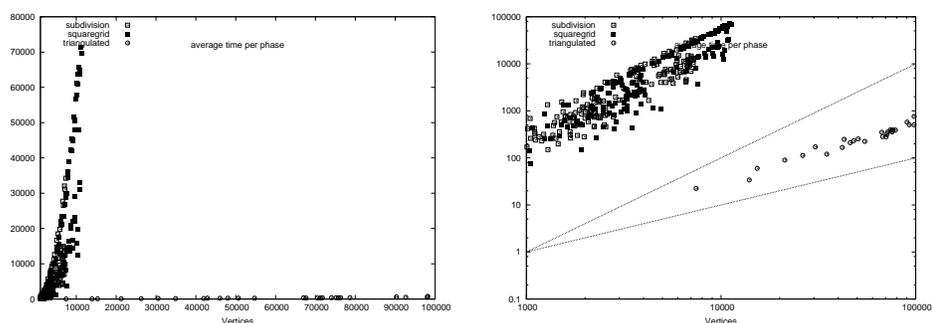


Figure 3.4.5: The left chart shows the time per phase and the right chart shows the same data in log-log scale.

In Figure 3.4.5 we see that the time needed per phase is sub-quadratic in the number of vertices for the data set “triangulated”. It seems to be quadratic, however, for the other two data sets.

To summarize, the repeated application of the uppermost path algorithm generally seems to have competitive running times, and on our instance sets it shows an equal or slightly better asymptotic behaviour than the benchmark algorithms AO and DM.

3.4.2 Complete the Solution with Benchmark Algorithms

For comparison we also ran the benchmark algorithms AO and DM on the sub-optimal solution delivered by the uppermost path algorithm. This also results in an optimal solution and shows how much “optimization work” is done by the uppermost path algorithm in its first iteration. See Figure 3.4.6.

The results are not very uniform. First we see that the variation in small

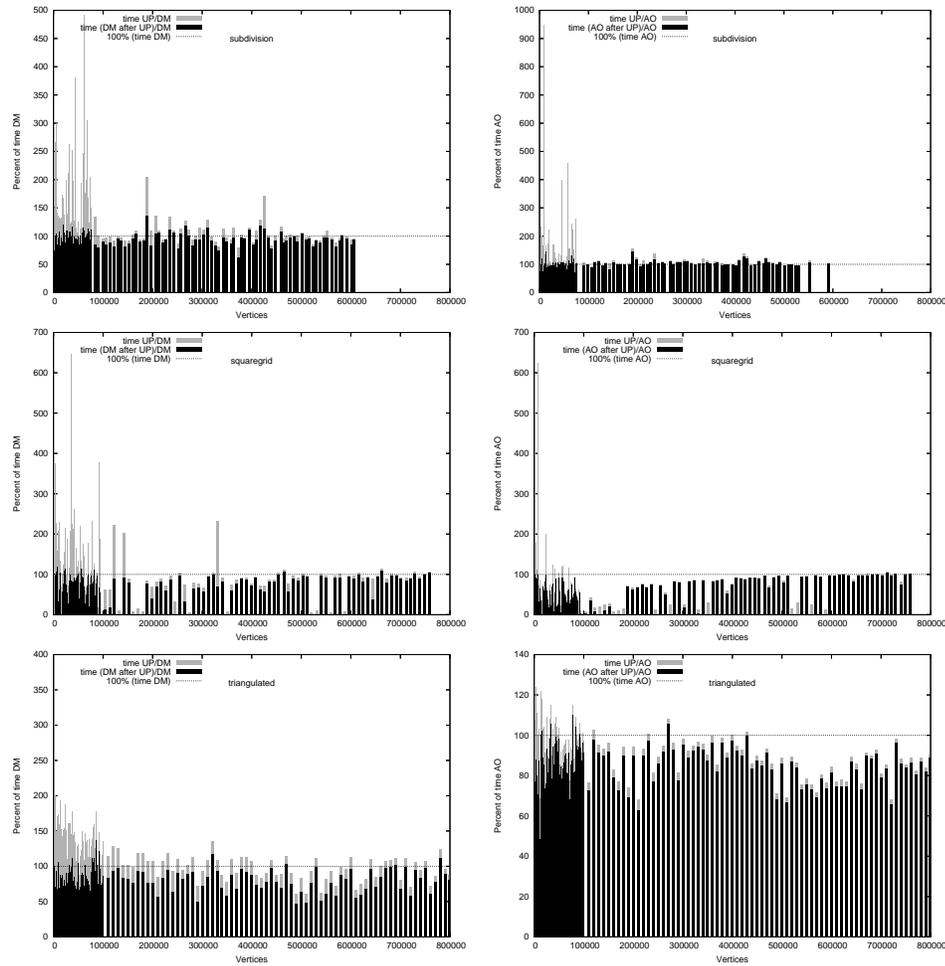


Figure 3.4.6: Comparison of running time of DM vs. UP+DM (=one iteration of UP plus completion by DM) in the left column. The black part of each bar is the running time of DM and the gray part the running time of UP. The total height of one bar is then the combined running time of UP+DM for one instance. The 100% line gives the running of DM for the respective instance. The right column shows the same for AO vs. UP+AO.

instances is quite large. Then we see that the running time of UP+DM exceeds 120% of DM in only a few instances. For “subdivision” there seems to be no gain in UP+DM over DM or in UP+AO over AO. For the other two instance sets, however, the combined algorithms have an advantage over the respective benchmark algorithm. This is most pronounced for the instance set “squaregrid” where there are quite a number of instances such that the running time of UP+DM or UP+AO is less than 50% of the running time of DM or AO, respectively.

3.5 Planarization by Force

In the previous section we used the uppermost path algorithm to obtain a sub-optimal solution and then improved upon it. Now we will start out with an *infeasible* solution (again computed by the uppermost path algorithm) and correct it until it is feasible.

We planarize the input graph before running the algorithm. Recall that we made the assumption that for the instances in our computational experiments we have a drawing in the plane. This drawing is not necessarily a true planar embedding as there may be edges that cross. Therefore we iteratively add a new vertex at any position where edges cross (compare Figure 3.5.7). This subdivides each crossing edge e in two edges e' and e'' . Let $e(e')$ be the original edge, that is e in our case. And let $E(e)$ be the set of derived edges, that is $E(e) = \{e', e''\}$ in this example. Eventually we get a planar graph G^\times along with a planar embedding of G^\times . Let V^\times be the vertices added. That is $V(G^\times) = V \cup V^\times$. And E^\times be the set of crossing edges in E . Thus we have $E(G^\times) = (E \setminus E^\times) \cup \{e' \in E(e) : e \in E\}$.

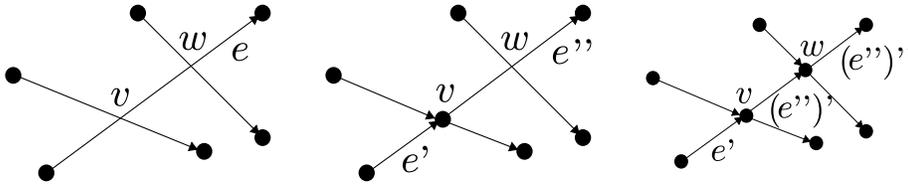


Figure 3.5.7: From left to right we show the construction of G^\times . We have an edge e that forms two crossings v and w with other edges. First a vertex v replaces the crossing v and e is subdivided in e' and e'' . Then w subdivides e'' .

We can now apply Berge's algorithm to G^\times . We get a maximum flow f^\times in G^\times that has a flow value equal or greater than a maximum flow in G . This maximum flow is not necessarily feasible for G . See Figure 3.5.8.

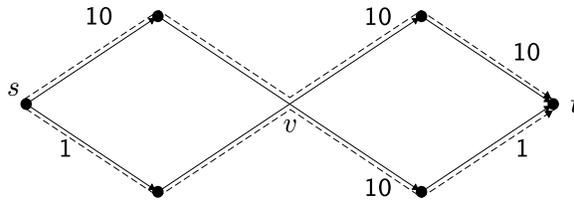


Figure 3.5.8: A network and the two augmenting paths (dashed) found by the uppermost path algorithm in G^\times : The upper path carries flow 10 and the bottom path carries flow 1.

Each edge in G^\times that has arisen from a crossing edge segment in G can have

an *imbalance* in f^\times . Consider two adjacent segments e_1 and e_2 in G^\times of the same crossing edge, and the crossing vertex v between them as shown in Figure 3.5.9.

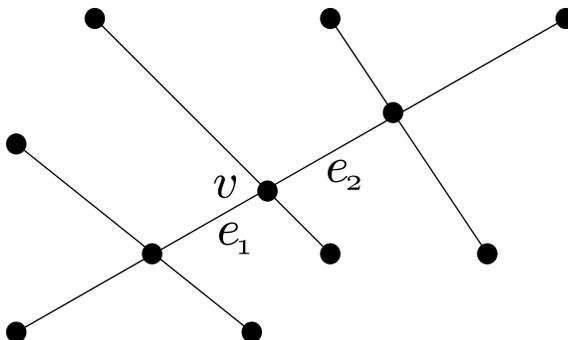


Figure 3.5.9.

We define the imbalance of e_1 at v as $\Delta_v(e_1) := f^\times(e_1) - f^\times(e_2)$ and analogously $\Delta_v(e_2) := f^\times(e_2) - f^\times(e_1)$. For example, the bottom edges at v in Figure 3.5.8 have imbalance -9 while the upper edges at v have imbalance 9 . For short we say “imbalances at vertex v ” referring to the imbalances on edges incident to v .

In order to gain a feasible solution for G we must reduce all imbalances to zero. Then and only then the flow on all segments of a crossing edge is the same and the flow is feasible for G .

To achieve this we apply a post-processing. For each crossing vertex v this post-processing has to find a flow in G^\times that

- (a) reduces the imbalances at v and
- (b) does not increase the imbalances at any other vertex.

We evaluated different variants of a search in $G_{f^\times}^\times$ that looks for augmenting cycles or paths that fulfill (a) and (b). A cycle in $G_{f^\times}^\times$ does not reduce the flow value of f^\times . Since this flow value can be greater than that of a maximum flow in G we also need t - s -paths that reduce the flow value. However, we must not use them unnecessarily. For an example of a correcting t - s -path see Figure 3.5.10.

We insert an arc (t, s) with infinite capacity into G^\times in order to limit the following analysis to cycles. Obviously, any cycle C containing (t, s) is equivalent to the t - s -path that arises from C by removing the arc (t, s) .

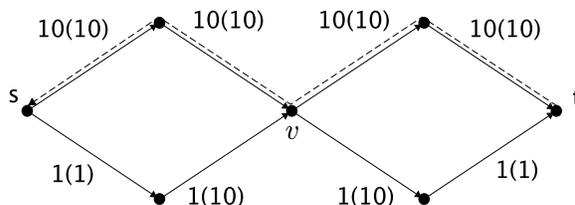


Figure 3.5.10: We show the residual network from Figure 3.5.8. The numbers denote the flow and — in parentheses — the original capacity of an edge. The dashed t - s -path with capacity 9 corrects the flow so that it becomes feasible for G .

3.5.1 Implementation Difficulties

There are quite a number of choices we have when implementing this post-processing. We list the most important ones with some sensible alternatives.

1. At which crossing, or arc should we start looking for a cycle? At the crossing or arc with the highest or lowest imbalance.
2. Should we keep balancing a crossing or try the next one? Both are possible.
3. Allow flow reducing cycles, that is cycles that contain the arc (t, s) ? Only if there are no non-reducing cycles starting at any crossing.
4. If the search reaches a crossing v and the last edge before v in the current search path is a segment of the crossing edge e in G , should we then allow the search to proceed with a segment from another crossing edge? Yes, if it does not deteriorate the balance.

This results in a number of problems. Above all we want to avoid flow reducing cycles. Therefore we must try another crossing if at one crossing there are no non-reducing cycles, anymore. This in turn results in a frequent switching from balancing one crossing to balancing another and so on. We will see in Chapter 4 that this kind of switching usually results in an unnecessarily long running time. We have made the same experience here.

The algorithm terminates, though. This can be seen by examining the total sum of imbalances which is reduced in each step and eventually reaches zero.

The use of dynamic trees is of advantage when large parts of a search tree remain unchanged between iterations. In our case the starting vertex of the search may change in every iteration. And whether an edge is admissible

or not changes as well. Therefore we cannot reasonably apply the dynamic trees data structure here.

All in all, we see no perspective for an efficient search strategy to find correcting cycles in $G_{f^\times}^\times$.

Chapter 4

Theory for Augmenting Path Algorithms

Overview of this Chapter. We will describe a number of augmenting paths algorithms. We begin with the most generic algorithm already presented in Section 3.1. This algorithm's specification has a large degree of freedom. With each further algorithm we present we reduce this freedom of choice. In addition we give theoretical results for all presented algorithms.

4.1 The Generic Augmenting Path Algorithm

Even in the most generic form given in Listing 3, we can make some assertions about the algorithm. We first prove a simple lemma about augmenting paths.

Lemma 4.1.1. *Consider a pair of paths P_i and P_j such that $X := E(P_i) \cap E(P_j)$ is non-empty. Then $P_j \setminus P_i$ contains at least $\ell - 1$ edges where ℓ is the number of connected components of $G[X]$.*

Proof. Let X_1, \dots, X_ℓ be the connected components of $G[X]$ and e_i^c and e_j^c be the first arc incident to X_c for $c = 2, \dots, \ell$ on P_i and P_j , respectively. For each c by definition the edge of e_j^c is not contained in X . Thus e_j^c is not contained in P_i , either. Therefore $P_j \setminus P_i \supset \{e_2^c, e_2^c, \dots, e_\ell^c\}$. \square

One important technique we use here is to derive limits for the number of augmentations performed by an algorithm. This is necessary because the number of augmenting paths in a graph can be in $O(2^m)$. See Figure 4.1.1. But rarely an augmenting path algorithm uses all of these paths.

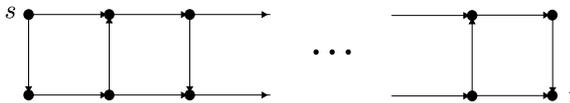


Figure 4.1.1: This class of graphs contains $2^{m/4}$ augmenting paths.

We will use the following technical lemma later on. We formulate it here because it holds even for the most generic augmenting path algorithm. From now on we will make extensive use of *intervals of paths*. Recall the definition on Page 8.

Lemma 4.1.2. *Let v be a vertex and U a vertex set in G . If in G_i there is no augmenting v - t -path disjoint to U , but in G_j ($j > i$) there is such a path Q , then the algorithm augments along at least one path P_ℓ ($i < \ell \leq j$) that contains a vertex in U .*

Moreover, Q and P_ℓ have a vertex u in common and $P_\ell|_{[u,t]}$ contains a vertex in U . And ℓ can be chosen to be the first iteration after i that has an augmenting v - t -path disjoint to U .

Proof. We choose ℓ to be the first iteration after i that has an augmenting v - t -path Q disjoint to U . Since there is no augmenting v - t -path in $G_{\ell-1} - U$, some edge must be augmented by P_ℓ to make Q possible in $G_\ell - U$. Let (u', u) be such an edge. I.e., $(u', u) \in E(P_\ell)$ and $(u, u') \in E(Q)$. Compare Figure 4.1. W.l.o.g. choose (u', u) such that $Q|_{[v,u]}$ is augmenting in $G_{\ell-1} - U$. Then the path $Q|_{[v,u]} + P_\ell|_{[u,t]}$ is an augmenting v - t -path in $G_{\ell-1}$. By our choice of ℓ it cannot be an augmenting v - t -path in $G_{\ell-1} - U$. Since $Q|_{[v,u]}$ is disjoint to U , $P_\ell|_{[u,t]}$ must contain a vertex in U . \square

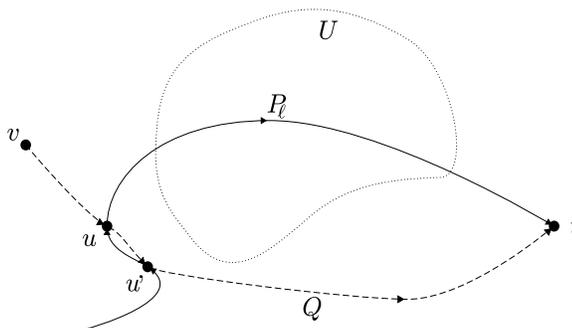


Figure 4.1.2: The situation in Lemma 4.1.2.

For an empty vertex set U this implies

Corollary 4.1.3. *If at one time in the algorithm there is no augmenting v - t -path for some vertex v , then there will be no augmenting v - t -path until the end of the algorithm.*

Similar to Lemma 4.1.2 but not quite the same is the following technical result, which we shall need later on.

Theorem 4.1.4. *Consider an arc (v, w) . Let $[i_1, i_2]$ be an interval of iterations such that $P_i|_{[s,v]} = P_{i_1}|_{[s,v]}$ for all $i \in [i_1, i_2]$. Let J be the set of iterations in which there is an augmenting w - t -path that is vertex-disjoint to $P_{i_1}|_{[s,v]}$. Then $I := [i_1, i_2] \cap J$ is an interval.*

Proof. Assume for a contradiction that I consists of at least two non-adjointing intervals $[j_1, j_2]$ and $[k_1, k_2]$ with $j_2 + 1 < k_1$. Let W be the set of vertices reachable from w by augmenting paths in $G_{j_2+1} - P_{i_1}|_{[s,v]}$. By assumption $j_2 + 1 \notin J$ and thus $t \notin W$ and in iteration k_1 there exists an augmenting w - t -path Q that is vertex-disjoint to $P_{i_1}|_{[s,v]}$. In order for Q to come into existence between iterations j_2 and k_1 , at least one arc in $E^-(W)$ must be augmented. Choose ℓ to be the first iteration when there is such an arc (u, u') . The augmenting path P_ℓ that contains (u, u') also contains a u' - t -path. This is a contradiction, since $u' \in W$ and $t \notin W$. \square

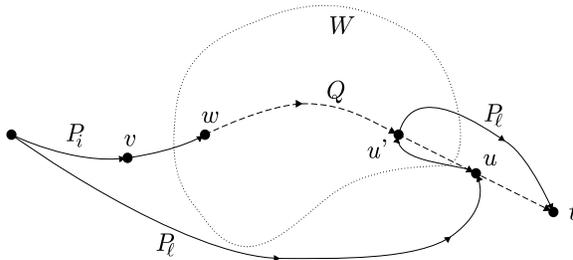


Figure 4.1.3: The situation in the proof of Theorem 4.1.4.

4.1.1 Direction Changes

A *direction change* is a vector (i, j, e) where P_i and P_j are augmenting paths and e is an edge such that $e \in P_i$ and $\overleftarrow{e} \in P_j$ and there is no $k \in]i, j[$ such that P_k contains e or \overleftarrow{e} .

Consider the classic examples that show that Ford and Fulkerson's algorithm may require exponential running time or may not terminate at all in Chapter 1. These use a lot of direction changes. Therefore it is justified to ask whether an examination of this characteristic can lead to better time bounds.

Theorem 4.1.5. *Let D be the number of direction changes an augmenting path algorithm uses while solving an instance. Then the number of augmenting paths used by the algorithm is not larger than $m + D$.*

Proof. Each augmentation along a path saturates at least one edge in the forward direction of this path. On the other hand, each direction change on an edge e makes e available for augmentation in the new direction. We start with m available edges. Each augmentation decrements this number by at least one and each direction change increments it by at most one. That is, $m - \#(\text{augmenting paths}) + D \leq 0$. \square

Thus a bound on the direction changes would yield bounds for the number of augmenting paths and the running time. Later we shall reconsider direction changes in the context of specific algorithms.

4.2 Alternative Algorithm Description

In Listing 3 we have given an exact description of the template augmenting path algorithm. In most cases an algorithm is easier to understand if it is given in an informal style, as well. Corresponding to Listing 3 is the following informal description.

(A1) While there is an augmenting path in the residual graph augment along some such path.

While our exact description uses an algorithmic template in order to enable specializations of the generic algorithm, we will formulate *algorithmic constraints* to specialize the informal description. For an example see the next section.

4.3 Iteration Labels

In some algorithms it is of advantage to have iteration labels on each arc. We define $\text{ilabel}(v, w)$ to be an integer associated with the arc (v, w) . This label gives the last iteration in which the arc was scanned by the algorithm. Note that the arcs (v, w) and (w, v) may have different label values.

One use of this is to restrict the algorithm to arcs not touched in the current iteration when it looks for an augmenting path. To do this we modify the

definition of `admissible` to check and update the label accordingly.

```

procedure admissible( $v, w$ )
  if ilabel( $v, w$ ) <  $i$  then
    ilabel( $v, w$ ) :=  $i$ 
    if  $c_i(v, w) > 0$  then true else false
  else
    return false
  end if
end procedure

```

The informal constraint is then

(A2) During a single search we touch each arc only once.

4.4 Stubborn Algorithms

Intuitively the search in the augmenting path algorithm takes less time if two subsequent paths have many arcs in common. This is especially true when using dynamic trees. Each arc that is added to the dynamic trees costs time $O(\log n)$. When searching without dynamic trees, the same arc costs only constant time.

How can we achieve that any two subsequent paths in the algorithm do not differ too much? We propose the following algorithmic constraint:

(A3) When searching for augmenting paths do not backtrack from v while there are augmenting paths that have $P_i|_{[s,v]}$ as a prefix.

An augmenting path algorithm that fulfills this constraint does not back down until absolutely necessary. Therefore we call such an algorithm *stubborn*. We also give additional constraints for the template algorithm:

- `active`(v) Returns `true` while there are augmenting paths that have $P_{[s,v]}$ as a prefix.
- `retreat`(P) Removes exactly the last edge from P .
- `repair`(P) Afterwards P is the longest augmenting prefix of the current path P .

A direct consequence of these constraints is the following lemma.

Lemma 4.4.1. *If (v, w) is contained in P_i but not in P_{i+1} , then in G_{i+1}*

- a) *there is no augmenting w - t -path that is vertex-disjoint to $P_i|_{[s,v]}$, or*
- b) *$P_i|_{[s,w]}$ is saturated.*

The following result gives an insight into the differences between our algorithm and the leftmost path algorithm for the planar case. In the latter algorithm the outgoing edge of a vertex v in the augmenting path revolves about v at most once (Lemma 3.3.3). In a stubborn algorithm we have

Lemma 4.4.2. *In an interval of iterations $[i_1, i_2]$ in which the prefix $P_i|_{[s,v]}$ of the augmenting path does not change, the outgoing arc of v in F revolves about v at most once.*

Proof. By Theorem 4.1.4 we know that $I := [i_1, i_2] \cap J$ is an interval, where J is defined in that theorem. Obviously (v, w) cannot be in the augmenting path in any iteration in $[i_1, i_2] \setminus J$. As soon as (v, w) is in the augmenting path, the algorithm does not remove it as long as there is still an augmenting w - t -path that is vertex-disjoint to $P_{i_1}|_{[s,v]}$. I.e. (v, w) is kept in the augmenting path until the end of I . That is, the time that any (v, w) is in the augmenting path during $[i_1, i_2]$ is an interval. This shows the assertion. \square

The following technical lemmata hold for stubborn algorithms.

Lemma 4.4.3. *Consider two augmenting paths P_i and P_j with $i < j$ and a vertex $w \in V(P_i) \cap V(P_j)$. If there is an iteration i' with $i \leq i' < j$ such that $G_{i'} - P_i|_{[s,w]}$ does not contain any augmenting w - t -paths, then $P_i|_{[s,w]}$ must be saturated by some path P_ℓ with $i \leq \ell < j$.*

Proof. We use induction on the distance of w from s on P_i . For $w = s$ the assertion is trivial. For $w \neq s$ we assume that the assertion is true for all j and for all w' nearer to s than w on P_i .

We apply Lemma 4.1.2 to iterations i' and j , the vertex w and the vertex set $V(P_i|_{[s,w]})$. Thus we know that there is an augmenting path $P_{j'}$ ($i' < j' \leq j$) that has a vertex u' in common with P_j and $V(P_{j'}|_{[u',t]}) \cap V(P_i|_{[s,w]}) \neq \emptyset$. See Figure 4.4. Let w' be a vertex that is contained both in $P_{j'}|_{[u',t]}$ and $P_i|_{[s,w]}$. Since the algorithm is stubborn either $P_i|_{[s,w']}$ or $P_{j'}|_{[w',t]}$ must be saturated between iterations i and j' . If $P_{j'}|_{[w',t]}$ is saturated we can apply the induction's assumption to w' which is nearer to s than w on P_i . Hence in both cases $P_i|_{[s,w]}$ must be saturated. \square

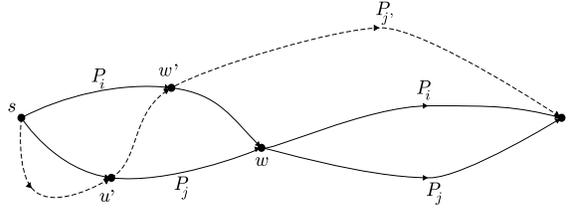


Figure 4.4.4: The situation in the proof of Lemma 4.4.3.

Next we show that a stubborn algorithm needs to scan each arc only once in between two augmentations. Compare this to the uppermost path algorithm, which has to scan each arc only once during its whole running time.

Lemma 4.4.4. *Any arc needs to be scanned only once during one iteration.*

Proof. Let Q be the current search path in an iteration i when the arc (v, w) is scanned. First, if $Q + (v, w)$ is the prefix of an augmenting s - t -path in G_i then a stubborn algorithm augments along such a path without scanning (v, w) again.

Otherwise, there is no augmenting w - t -path in $G_i - Q|_{[s,v]}$. If no augmenting s - t -path in G_i contains (v, w) then we need not scan (v, w) again in iteration i . Therefore we assume that there is an augmenting s - t -path Q' in G_i that contains (v, w) .

Thus there are also augmenting w - t -paths in G_i . And these w - t -paths all share vertices with Q . $Q'|_{[s,v]}$, on the other hand, is disjoint to at least one augmenting w - t -path P . See Figure 4.4. Let u be the first vertex on Q that also lies on P . Then the longest common prefix of Q and Q' ends before u on Q . Thus $Q|_{[s,u]} + P|_{[u,t]}$ is an augmenting s - t -path in G_i which a stubborn algorithm finds before finding Q' . Therefore (v, w) does not need to be scanned in iteration i again. \square

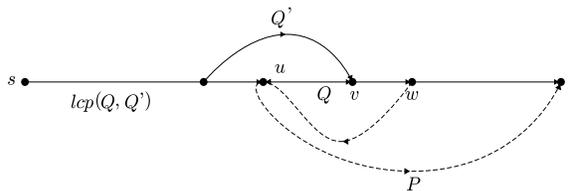


Figure 4.4.5: The situation in the proof of Lemma 4.4.4. By $\text{lcp}(Q, Q')$ we denote the longest common prefix of the two paths.

The previous lemma shows that we do not lose augmenting paths if we use iteration labels in a stubborn algorithm. The next result is even stronger.

Theorem 4.4.5. *Stubbornness and iteration labels are compatible. In other words, a stubborn algorithm finds the same sequence of augmenting paths whether it is implemented with or without iteration labels.*

Proof. We consider two algorithms, one with iteration labels (WITH) and the other without (WITHOUT). Assume for a contradiction that algorithms WITH and WITHOUT both find P_1, P_2, \dots, P_{i-1} but then WITHOUT finds P_i whereas WITH finds $P'_i \neq P_i$. Hence, WITH must have labelled at least one edge $(u, v) \in E(P_i)$ in iteration i . Let Q be the s - u -path at the moment when (u, v) was labelled. In $G_i - Q$ there is no augmenting v - t -path because otherwise it would have been augmented before backtracking from u (A3). Since $P_i|_{[v,t]}$ is augmenting in iteration i , it must have at least one vertex w in common with Q . Then $Q|_{[s,w]} + P_i|_{[w,t]}$ is the next augmenting path after P_{i-1} in both WITH and WITHOUT, contradicting the choice of P_i . \square

Search paths. As usual we denote by P_1, P_2, \dots the paths augmented by the algorithm. We say a path that the algorithm considers during a search phase is a *search path*. Note that many inclusion maximal search paths do not reach t . In general, we only consider inclusion maximal search paths and write them as Q_1, Q_2, \dots . There are usually many inclusion maximal search paths during one iteration. Thus Q_i is in general *not* in iteration i . We denote the set of all search paths in iteration i by \mathcal{Q}_i . Note further that P_i is always the last search path in \mathcal{Q}_i for all i .

We now show that a stubborn algorithm will never again need to use the prefix of a search path, once this prefix drops from the current path. In Corollary 4.1.3 we have seen a similar statement for suffixes.

Lemma 4.4.6. *If the arc (v, w) is contained in the search path Q_{i_0} in iteration i but not in the next search path Q_{i_0+1} then no augmenting s - t -path contains $Q_{i_0}|_{[s,w]}$ in any G_j for $j \geq i$.*

Proof. Let i' be the iteration Q_{i_0+1} belongs to. We use induction on the distance from s to v on Q_{i_0} . For $v = s$ the arc (v, w) is incident to s . Therefore in $G_{i'}$ (v, w) is saturated or there is no w - t -path. In the former case (v, w) is saturated until the end of the algorithm. And in the latter case there will not be a w - t -path again, by Corollary 4.1.3. In any case the assertion holds for $v = s$. For $s \neq v$ we assume that the assertion is true for all vertices v' nearer than v to s on Q_{i_0} .

Moreover, assume for a contradiction that in an iteration $j \geq i'$ there is an augmenting s - t -path that contains $Q_{i_0}|_{[s,w]}$. Let j be the first iteration greater or equal than i' that contains such a path R . We are going to show that there exists an augmentation path P_ℓ ($i' \leq \ell \leq j$) such that $\text{lcp}(Q_{i_0}, P_\ell)$ ends before a vertex u' which lies before v on Q_{i_0} . Thus the stubborn algorithm backtracks from u' between iterations i' and ℓ . And the arc immediately before u' on Q_{i_0} is removed from the path. Therefore, by induction the prefix $Q_{i_0}|_{[s,u']}$ cannot be contained in any augmenting path after $P_{\ell-1}$. In particular, R cannot contain $Q_{i_0}|_{[s,v]}$.

To show this, we first consider the case that $Q_{i_0}|_{[s,v]}$ is augmenting in $G_{i'}$. Then there are no augmenting v - t -paths in $G_{i'} - Q_{i_0}|_{[s,v]}$. We now apply Lemma 4.1.2 to the vertex v , vertex set $V(Q_{i_0}|_{[s,u]})$ and iterations i' and j . The lemma then yields an augmenting path P_ℓ ($i' < \ell \leq j$) that has a vertex u in common with R . Moreover, $P_\ell|_{[u,t]}$ shares a vertex u' with $Q_{i_0}|_{[s,v]}$. See Figure 4.4.6.

In the second case, $Q_{i_0}|_{[s,v]}$ is saturated in $G_{i'}$. Since $Q_{i_0}|_{[s,v]}$ is contained in R an edge (u, u') in $Q_{i_0}|_{[s,v]}$ must be augmented by a path P_ℓ ($i' < \ell \leq j$). Now we have $Q_{i_0}|_{[s,v]} \neq P_\ell|_{[s,v]}$ and therefore u' must be dropped from the current path between iterations i' and ℓ . This concludes the proof. \square

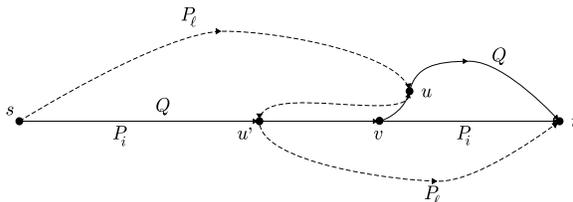


Figure 4.4.6: The situation in the first part of the proof of Lemma 4.4.7.

An immediate consequence is

Lemma 4.4.7. *If the vertex v is contained in Q_{i_0} but not in Q_{i_0+1} then no augmenting s - t -path contains $Q_{i_0}|_{[s,v]}$ in any G_j for $j > i$.*

Since there is only a finite number of prefixes and in any iteration at least one edge is saturated we have the following Corollary.

Corollary 4.4.8. *A stubborn algorithm terminates after a finite number of augmentations.*

Unfortunately, the algorithm can make very stupid choices in determining the next path. This can result in reverse augmentations even in planar graphs.

In fact, for any ℓ we can construct a graph G^ℓ such that a stubborn algorithm needs $O(m^\ell)$ augmentations to find the maximum flow in the worst case. See Figure 4.4.7. The construction is recursive, G^i contains G^{i-1} . In the figure G^2 contains G^1 (marked with a dashed line) and G^3 contains G^2 . There are m_1 edges (s_1, t_1) each of capacity $c_1 = 1$. There are m_i edges each of (s_i, s_{i-1}) and (u_{i-1}, v_{i-1}) each of capacity $c_{i-1} = m_1 m_2 \cdots m_{i-1}$. The edges (s_{i-1}, t_i) and (t_{i-1}, t_i) have capacity c_i each.

In order to prove our claim about the running time, we construct a special sequence of augmenting paths that the algorithm *may* choose and which result in the given number of augmentations. This is sufficient. Clearly this is a very special choice of augmentations. An arbitrary sequence of augmenting paths will usually need much fewer augmentations.

The augmenting paths in G^2 are these:

for $j = 1, \dots, m_2$ (
 for $i = 1, \dots, m_1$: $(s_2, s_1)_j, (s_1, t_1)_i, (t_1, t_2)$
 for $i = 1, \dots, m_1$: $(s_2, u_1), (u_1, v_1)_j, (t_1, s_1)_i, (s_1, t_2)$
)

This can easily be generalized for $\ell > 2$. If we choose $m_1 = m_2 = \dots = m_\ell$ then we need $O((\frac{m}{\ell})^\ell)$ augmentations to solve G^ℓ . That is, for any fixed ℓ we have an instance that requires $O(m^\ell)$ augmentations where $m = |E(G^\ell)|$.

Note that the counter-example strongly relies on most capacities being equal. This is not natural for real-world data.

4.5 Augmenting Path Algorithms with Arc Memory

In an algorithm *with arc memory* each incidence list has a *current edge*. Any time the search reaches a vertex it begins scanning at the current edge of that vertex and when the search goes on to the next incident edge, the current edge is updated accordingly.

An algorithm *without arc memory*, in contrast, uses no information from previous iterations to decide which edge should be the next at vertices which current incoming arc was not contained in the previous inclusion maximal search path.

The following short constraint realizes arc memory in the template algorithm:

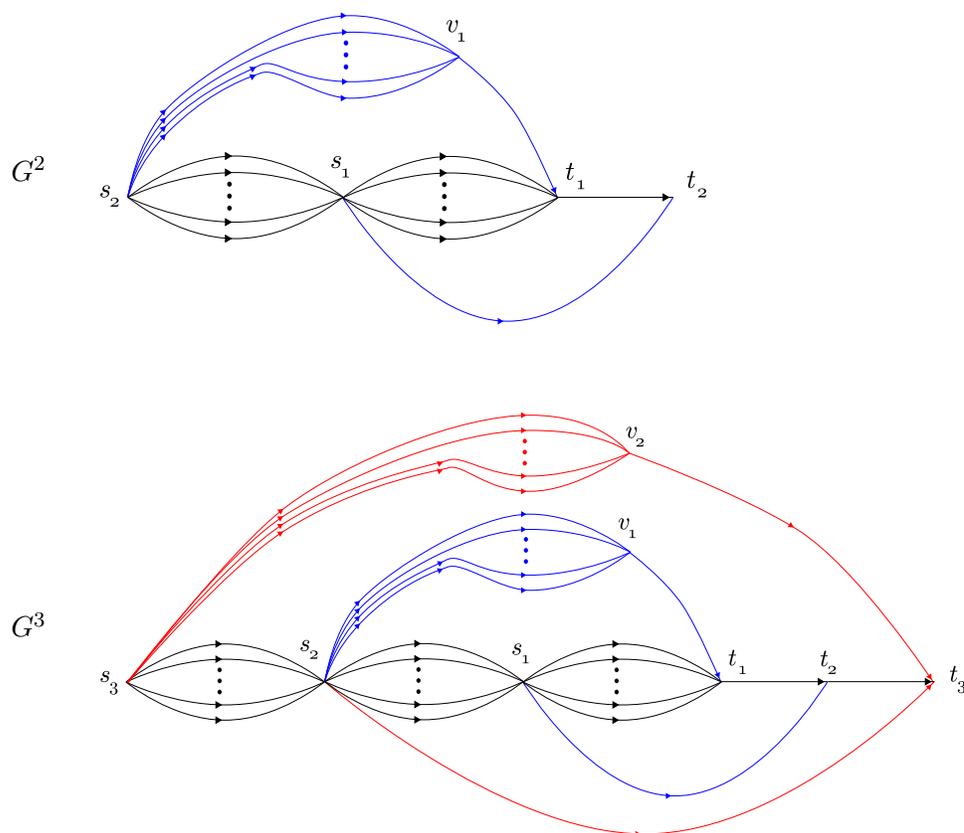


Figure 4.4.7.

`current_arc(v)` Remains unchanged while v is not in the current path.

We also give an informal algorithmic constraint.

When dropping an edge from the path, we store its current edge and start scanning at this edge when encountering the edge again.

We use arc memory in an augmenting path algorithm because it stabilizes the paths such that two subsequent paths do not differ too much. This stability is meant to improve the performance as we have noted earlier.

A stubborn algorithm that has arc memory solves the counter-example G^ℓ described in Figure 4.4.7 in $O(m)$ augmentations for any ℓ . To see this, we have to prove that the length of any sequence of augmenting paths the algorithm might find is in $O(m)$.

We show this exemplarily for G_2 : The only two edges incident to t_2 are (s_1, t_2) and (t_1, t_2) . Thus any augmenting path terminates in one of these edges. Therefore the incoming arc of t_2 is the outgoing arc of s_1 or the outgoing arc of t_1 . By arc memory neither of these outgoing arcs changes until it is saturated. Then we have at most two direction changes on each arc $(s_1, t_1)_i$. Thus the number of augmenting paths does not exceed $2m_1m_s$.

4.6 Stubborn and Balanced Algorithms with Arc Memory

We say an algorithm is *balanced*, if it conforms to the following algorithmic constraint.

- (A5) Between two scans of an arc (v, w) from v all other arcs incident to v are scanned.

We do not give a constraint for the template algorithm because we do not need it and because it is unnecessarily complicated.

However, the properties stubborn and balanced with arc memory are not strong enough to characterize an efficient algorithm: Again we can construct planar instances such that this kind of algorithm may need running time $O(m^\ell)$ for any ℓ . And, again, the counter-example relies on most capacities being equal on each augmenting path (see Figure 4.6.8).

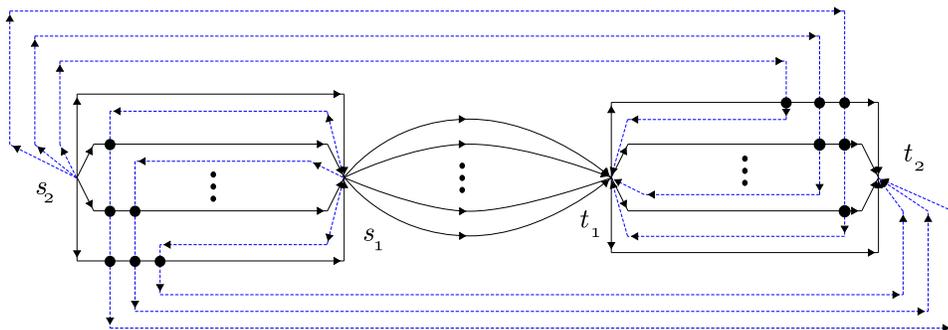


Figure 4.6.8.

Like the previous counter-example, the construction is recursive so G^i contains G^{i-1} . G^1 consists only of the edges $(s_1, t_1)_i$ for $i = 1, \dots, m_1$ with capacity $c_1 = 1$. G^i for $i \geq 2$ arises from G^{i-1} by adding $m_i/2$ solid and $m_i/2$ dashed paths in the infinite face as shown in the figure. Where paths cross we insert vertices. As we can see there are $O(m_i^2)$ new edges and vertices. All new edges have capacity $c_i = c_{i-1}m_{i-1}$.

For easier reference we number the paths as follows. The first path, $P_{1,1}$, is the leftmost solid path. Then we denote $P_{1,i} = P_{1,1}|_{[s_2,s_1]} + (s_1, t_1)_i + P_{1,1}|_{[t_1,t_2]}$. Note that in G^ℓ the paths have ℓ indices. Then $P_{2,1}$ is the leftmost dashed path and uses $(s_1, t_1)_1$ where $P_{2,i}$ use $(s_1, t_1)_i$. This goes on alternatingly. $P_{3,1}$ is the second leftmost solid path and $P_{4,1}$ is the second leftmost dashed path. And so on. Multiplying the number of indices we see that there are $m_1 m_2$ augmenting paths in G^2 , and generally $m_1 m_2 \cdots m_\ell$ in G^ℓ .

It is easy to see that each of these paths is in fact an augmenting path in the corresponding residual graph. Since all capacities in one level of the construction are equal, a stubborn algorithm can find these paths. By the same argument, arc memory does not inhibit finding these paths, either.

To prove that the same goes for balanced algorithms we need a new argument. After augmenting along $P_{1,i}$ and $P_{2,i}$ for $i = 1, \dots, m_1$ the algorithm builds a search path $Q = P_{3,1}|_{[s_2,s_1]}$. Now, a balanced algorithm scans all arcs at s_1 before scanning $(s_1, t_1)_1$ again. The ruse here is that all unsaturated dashed paths leaving s_1 share vertices with Q and all saturated solid paths entering s_1 do not share vertices with Q . The same ruse works at all vertices s_1, \dots, s_ℓ and t_1, \dots, t_ℓ . The balanced constraint has no effect at any other vertices since each of these other vertices is contained in at most two paths. Therefore the balanced algorithm can find the proposed sequence of paths, as well.

We see that this example relies upon the free choice of the next arc the algorithm has during the search. This choice is somewhat limited by the balanced constraint but not enough as we have seen.

4.7 List Scan Algorithms

One way to further limit the choices of the path search is to force it to use the edges incident to a vertex in a fixed order. I.e., each time the path search reaches v , it will first use (v, u_1) , then (v, u_2) and so on. Here (u_1, u_2, \dots) is the *scan list* of v and is some fixed permutation of the incidence list of v . This is a special case of balanced algorithms. We call such an algorithm a *list scan algorithm*.

Without arc memory the algorithm will begin again at the list's head every time it reaches the vertex. With arc memory it resumes at the current list position.

In the context of list scan algorithms with arc memory we can speak of the

revolutions of the current edge about a vertex. This concept has been introduced by Berge for his Uppermost Path Algorithm where it has a graphic interpretation. Compare Lemma 3.3.3. Here we say the current edge *revolves k times about v* if the algorithm traverses v 's scan list k times.

4.8 Left–first Search Algorithms

We say an algorithm uses *left–first search* if it is stubborn and its search obeys the following algorithmic constraints.

- (A6) The edges incident to a vertex are scanned in the order given by the combinatorial embedding.
- (A7) We start scanning the incidence list of s at the clockwise first arc after the infinite face.
- (A8) Upon first discovery of a vertex we start scanning its incidence list with the clockwise next arc after the current incoming edge.

The Constraint A6 establishes that this strategy is a special case of list scanning.

This search strategy is known as “left–first search” since it was first used on planar graphs with incidence lists ordered in a counter–clockwise fashion. See the uppermost path algorithm in Section 3. This ordering in a planar graph results in augmenting paths that are ordered (graphically) from left to right.

It is easy to see that the augmenting paths in Figure 4.6.8 are not left–first search paths in the given network. Theorem 4.8.4 below shows this in general: a left–first search algorithm solves any planar instance efficiently.

We can however modify the embedding such that even a left–first search algorithm needs running time $O(m^L)$ for any L . It is even easier to do this for Figure 4.4.7. We only need L crossings. See Figure 4.8.9 for the modified example prototype.

This is the first case where a counter-example to one of our algorithm invariants contains crossings. Theorem 4.8.4 in fact will show that we need crossings in order to build an example that left–first search algorithms do not solve efficiently.

- (A9) Upon any discovery of a vertex we start scanning its incidence list with the clockwise next arc after the current incoming edge.

We say an algorithm uses strictly left-first search if it adheres by this constraint. See Figure 4.8.10 for an example where this makes a difference.

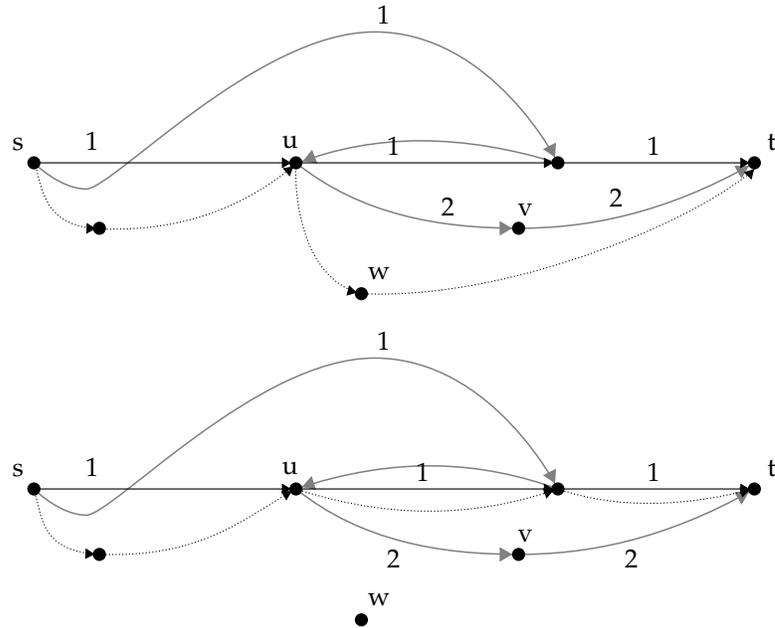


Figure 4.8.10: Two algorithms on the same instance: Left-first search with arc memory (above) or without arc memory (below). They both find first the black and then the gray path. In the third (dotted) path, however, they differ. Unused edges incident to w are not shown in the lower illustration.

4.8.2 DeleteIncoming and EnableFastForward

Consider the moment when an augmenting path P_i has been saturated. The generic algorithm then calls `repair`. The procedure `repair` must cut at least all saturated arcs in the path. Here we choose to cut exactly these arcs and maintain the then disconnected parts of P_i in a branching (directed forest) data structure. This is especially useful if we use dynamic trees because this data structure maintains the branching without further work.

This may lead to problems if we connect the head v of the augmenting path to a vertex w in an existing tree in the branching. Possibly w is not the root of the resulting tree. Since `advance` must return the root of the current path tree, it cannot simply return w .

We identify two possibilities to overcome this problem. First, we may delete all incoming tree arcs before connecting to w . Our algorithm does this if and only if the parameter *DeleteIncoming* is 1.

The second option is to extend the path to the root of the resulting tree after connecting the two trees. This is done if and only if *EnableFastForward* is 1. However, if the root of the resulting tree after linking is t then we extend the path to t regardless of the value of *EnableFastForward*. This makes the search much faster.

If both *DeleteIncoming* and *EnableFastForward* are 0 then we cut the resulting tree behind w after linking. Thus w becomes the root of the resulting tree.

These parameters are not specific to left-first search. They apply to any augmenting path algorithm which maintains a forest of arcs. We mention them here for the first time because we did not give a concrete implementation of the more the generic algorithms earlier in this chapter.

4.8.3 FinalLabels

We try to make the search faster by marking arcs which are unusable with a *final label*. The intended meaning of these labels is that the algorithm need never again scan arcs which are thus marked. Here we use a simple heuristic to find such arcs. We label an arc finally if it has zero residual capacity. And if the arc is ever augmented in contrary direction then we remove the label. The use of these labels is enabled by setting the parameter *FinalLabels* to 1.

4.8.4 The Optimal Parameter Settings

It is not at all obvious which parameter setting is the best. Therefore we ran all 16 possible combinations on our test data suite. We show the data in Figure 4.8.11 and in log-log scale in Figure 4.8.12. We can see that for “subdivision” and “squaregrid” the running time of any parameter combination is almost proportional to any other. Therefore we may compare the combinations by their accumulated running time over all instances. We show this data in Figure 4.8.13. For the data set “triangulated” it seems that some parameter combinations have a much stronger growing running time than others. Again we refer to Figure 4.8.13 for a better overview.

In Figure 4.8.13 we see that the algorithms defined by different parameter combinations perform quite differently on the three instance sets. It is

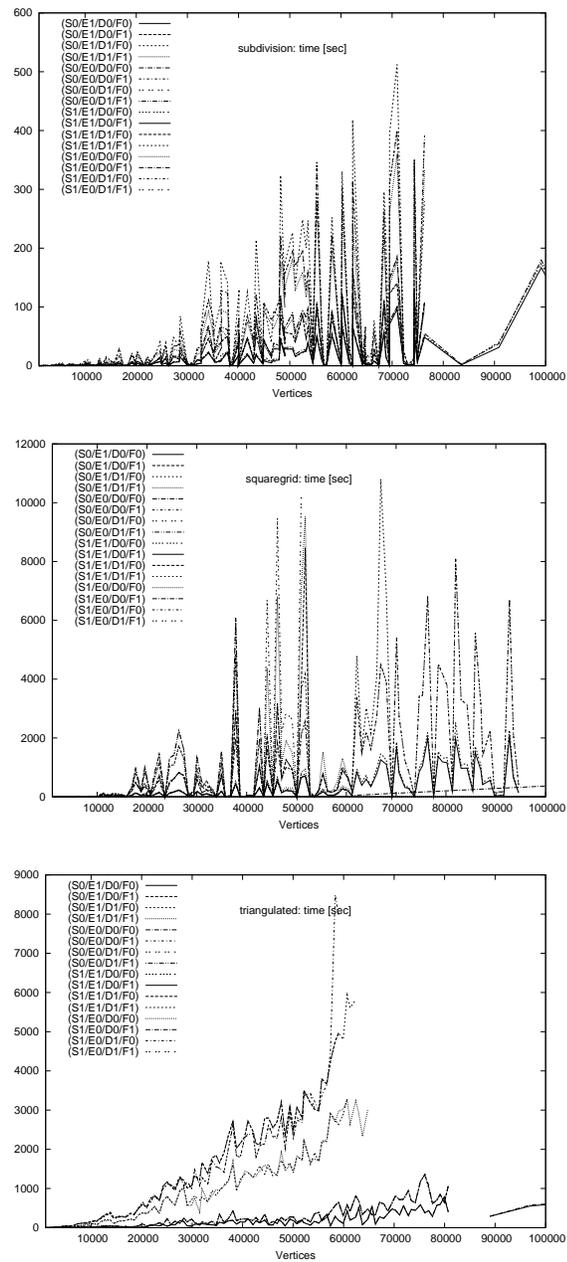


Figure 4.8.11: Running time for left-first search algorithm parameter combinations.

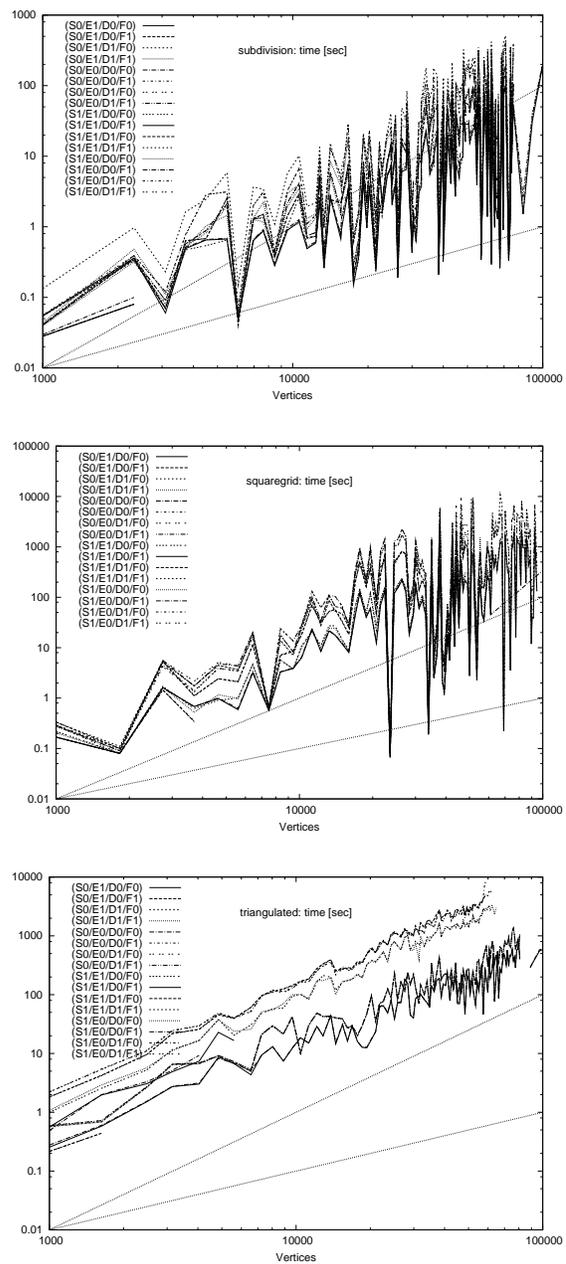


Figure 4.8.12: The same data as in Figure 4.8.11 on a log-log scale. The two straight lines show functions proportional to $y = x$ and $y = x^2$ respectively.

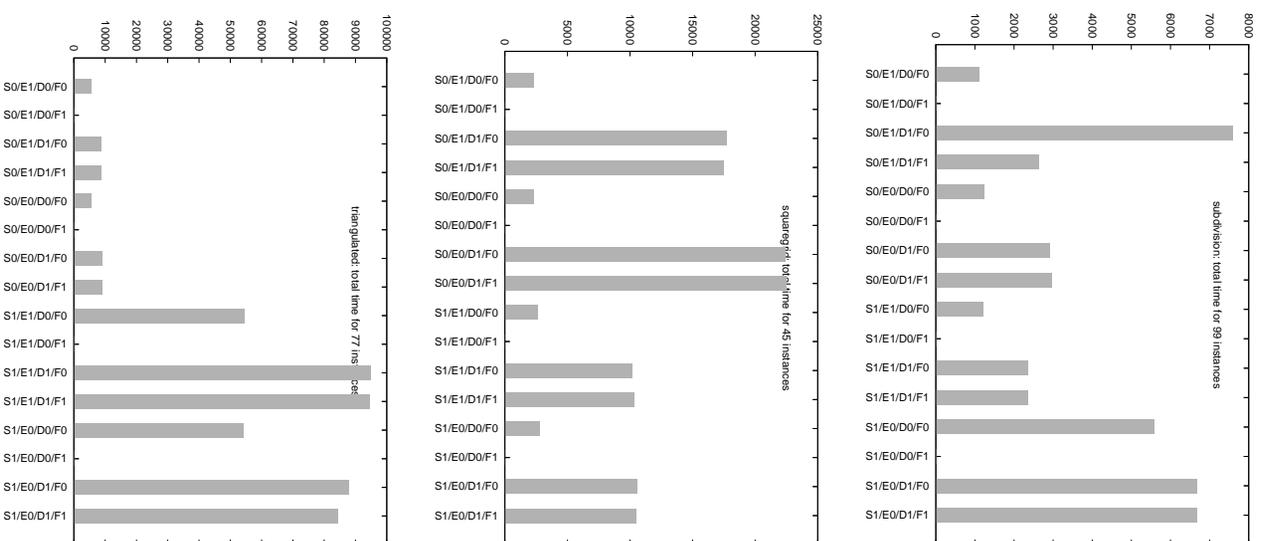


Figure 4.8.13.

interesting to see that for each instance set the combinations fall into three distinct groups: the best, the second best and the worst. However the membership in the three groups varies from one data set to the other. The only constant here is that combinations (S0,E1,D0,F0) and (S0,E0,D0,F0) are in the best group for all three instance sets.

Interestingly, for “triangulated” all non-StrictlyLeftmost combinations perform very well while for “squaregrid” the StrictlyLeftmost combinations perform better.

Although the absolute values are very different, the asymptotical performance of all parameter combinations is the same. From Figure 4.8.12 it seems that the asymptotical running time is in $\Theta(n^2)$ for all combinations.

We do not show the diagrams of operation counts here. But, as always, they are proportional to the time diagrams.

In Figure 4.8.14 we present the results for the two best parameter combinations (S0,E1,D0,F0) and (S0,E0,D0,F0) on a greater set of instances in order to analyze the asymptotic behaviour. Here the algorithm exhibits quadratic performance. Only for the instance set “triangulated” the asymptotics are slightly better.

We also draw the number of augmentations over $m + k$ in Figure 4.8.15. From this figure it seems that the number of augmentations is on $O(m + k)$. However, we did not find a way to prove this formally.

4.8.5 A Theory for Left-first Search Algorithms

We are now going to prove a couple of statements about strictly leftmost algorithms. That is, algorithms that are stubborn and adhere by A6, A7 and A9. We choose these since they seem to allow an easier analysis. We can make this choice since by our computational results the asymptotical performance is the same whether the algorithm is strictly leftmost or not.

Assumptions. W.l.o.g. we assume that t lies on the infinite face.

Left and right hand sides of paths. We say an edge $\{v, v'\}$ is *incident to the left hand side* of a path, if the path contains the arc sequence $((u, v), (v, w))$ and in the clockwise ordered cyclic incidence list of v the edge $\{v, v'\}$ lies behind $\{u, v\}$ and before $\{v, w\}$. The definition for the path’s right hand side is analogous.

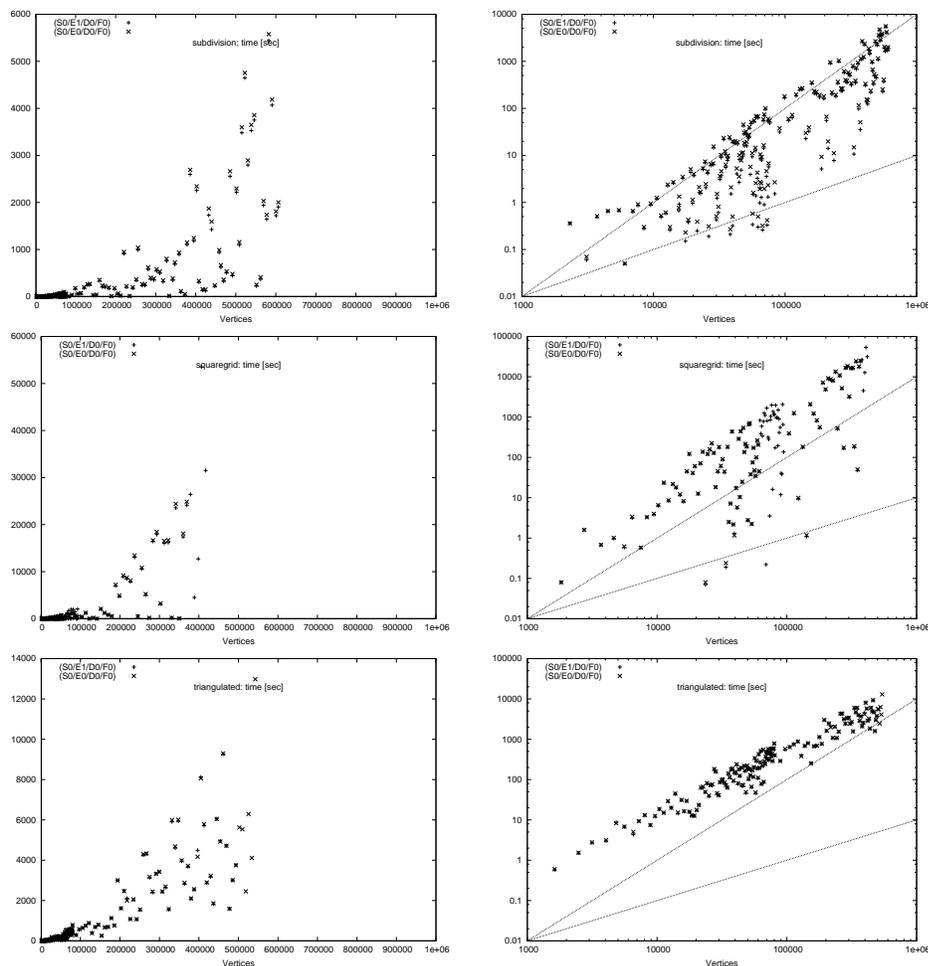


Figure 4.8.14: Left column: Running time for the best two combinations on a greater set of instances. Right column: the same data in log-log scale.

Lexicographically sorted paths. Let v be a vertex on P_i that is also used by P_j . Let e be the incoming edge of v shared by both paths. If $v = s$, then set $e = (t, s)$. Let e_i be the outgoing edge used by P_i and e_j the outgoing edge used by P_j .

We say two paths P_i and P_j with $i < j$ are *sorted* at v , if the clockwise order of these edges in the incidence list of v is (e, e_i, e_j) . If $e_i = e_j$ then that simplifies to (e, e_i) . We say P_i and P_j are *strictly sorted* at v , if $e_i \neq e_j$ in the previous definition.

If P_i and P_j are sorted at the last vertex of their common prefix then we write $P_i < P_j$. This defines a total order on any set of paths that all begin

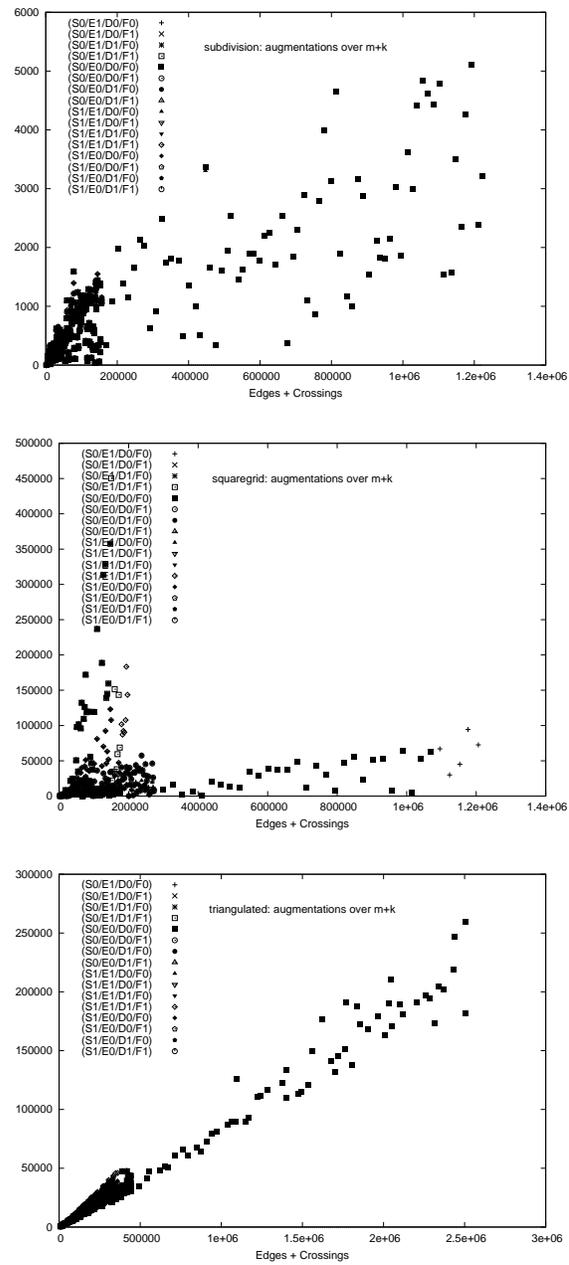


Figure 4.8.15.

at the same vertex. We say such a set is *lexicographically sorted* if for any two paths in the set $i < j$ implies $P_i < P_j$.

The strictly left–first search strategy immediately yields the following

Lemma 4.8.1. *The vector of search paths found by a left–first search algorithm is lexicographically sorted.*

The next lemma shows a strong property of left–first search algorithms. Namely that the strictly left–first search algorithm augments the leftmost path in any iteration.

Lemma 4.8.2. *If $Q \in \mathcal{Q}_i$ is a search path of a left–first search algorithm and $v \in V(Q)$ then in G_i there is no augmenting v – t –path that starts on the left side of Q . In particular, P_i is the leftmost augmenting path in G_i .*

Proof. Since the network (G, c) does not change during one iteration we only have to show the assertion for the first inclusion maximal search path in any iteration. By left–first search the assertion holds for the very first inclusion maximal search path found by the algorithm. We proceed by induction and for any $k > 1$ we can assume that the assertion is true for $i < k$.

Recall that P_i is the last search path in \mathcal{Q}_i . Let Q be the first search path in \mathcal{Q}_{i+1} and v the last vertex in the common prefix of P_i and Q . Assume for a contradiction that in G_{i+1} there is an augmenting path R that starts on the left side of Q and ends at t . R cannot start on the left side of $Q|_{[v, \text{tail}(v)]}$ since we have just scanned that part at the beginning of iteration $i + 1$. Thus R must start on the left side of $Q|_{[s, v]} = P_i|_{[s, v]}$. But that is impossible by induction. \square

Now we have the basis to prove the following technical result which we will use frequently later on.

Lemma 4.8.3. *Consider a search path $Q_{i_0} \in \mathcal{Q}_i$ and an augmentation path P_j with $i < j$. Let v be a vertex in $V(Q_{i_0}) \cap V(P_j)$ and w the vertex immediately after v on P_j . If Q_{i_0} and P_j are not sorted at v then there must be an augmentation path P_ℓ ($i \leq \ell < j$) such that P_ℓ contains (w, v) and crosses $Q_{i_0}|_{[s, v]}$ to the left or $P_j|_{[s, v]}$ to the right in a crossing. See Figure 4.8.16.*

Proof. By Lemma 4.8.2 in G_i there is no augmenting v – t –path that starts on the left side of Q_{i_0} . In G_j on the other hand there is such a path, namely $P_j|_{[v, t]}$. Therefore at least one arc in $P_j|_{[v, t]}$ must have been augmented in reverse direction by an augmentation path P_ℓ ($i \leq \ell < j$).

Now we choose ℓ minimal and assume for a contradiction that (w, v) is not in P_ℓ . Let u be the first vertex on $P_j[v, t]$ that is also in P_ℓ . Then $P_j[v, u] + P_\ell[u, t]$ is a v - t -path that starts left of Q_{i_0} . That is a contradiction to ℓ being minimal.

Now (w, v) lies on the left side of the cycle $Q_{i_0}|_{[s,v]} + \overleftarrow{P_j}[s,v]$ but P_ℓ must start on its right side by Lemma 4.8.1. Therefore $P_\ell|_{[s,w]}$ must cross the cycle to the right side. \square

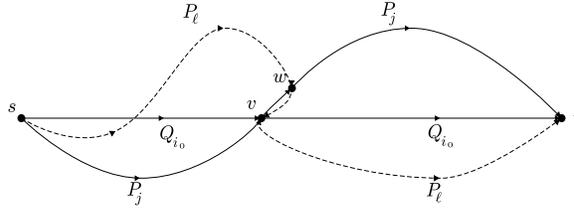


Figure 4.8.16: In this situation Lemma 4.8.3 predicts the existence of P_ℓ .

4.8.6 Regressions

A *directed crossing* is a pair of arcs $((u, v), (u', v'))$ such that the edges $\{u, v\}$ and $\{u', v'\}$ form a crossing. A *regression* is an iteration i of the algorithm such that P_{i+1} crosses P_i to the left. The *regression location* is then the first vertex or directed crossing on P_i where P_{i+1} crosses to the left.

Obviously the number of regressions is always less than the number of augmenting paths. If the algorithm makes no regressions then each edge is touched only once. That is because each path divides the planar graph in a region on the path's left side and a region on the path's right side. Without regressions the left side region is never again changed and thus will never again contain augmenting paths to t . And since any augmenting path saturates at least one arc we have the following

Theorem 4.8.4. *A left-first search algorithm solves any planar instance with at most m augmentations.*

In general, each regression makes available the left side region again which can contain up to m edges. Thus we have

Theorem 4.8.5. *The number of augmentations is not greater than $(r+1)m$ where r is the number of regressions.*

Therefore, if we can bound the number of regressions we can also derive a running time bound for the left–first search algorithm. We conjecture that the number of regressions is in $O(k)$ where k is the number of crossings. But we were not able to find a proof for this.

4.8.7 Direction Changes

Recall the definition of “direction changes” from Section 4.1.1. We try to derive a worst-case running time bound for our left–first search algorithm by counting direction changes.

What is the maximum number of direction changes on some edge $\{u, v\}$? We try to relate this number to some other characteristic figures of the algorithm. For this consideration we examine a special type of subgraph.

Tent graph. Consider a subsequence $P_{i_1}, P_{i_2}, \dots, P_{i_\ell}$ of the augmenting paths found by a left–first search algorithm with arc memory. We say the paths $P_{i_1}, P_{i_3}, P_{i_5}, \dots$ are *odd paths* and the paths $P_{i_2}, P_{i_4}, P_{i_6}, \dots$ are *even paths*. These paths form a *tent structure* if there are vertices u and v such that

1. each P_{i_j} contains u or v and
2. for odd j $P_{i_j}|_{[s,u]}$ does not contain v and
3. for even j $P_{i_j}|_{[s,v]}$ does not contain u .

Then the subgraph $T(i_1, i_2, \dots, i_\ell)$ with vertices $V(P_{i_1}) \cup V(P_{i_1+1}) \cup \dots \cup V(P_{i_{\ell-1}}) \cup V(P_{i_\ell})$ and edges $E(P_{i_1}) \cup E(P_{i_1+1}) \cup \dots \cup E(P_{i_{\ell-1}}) \cup E(P_{i_\ell})$ is the corresponding *tent graph*. Figure 4.8.17 shows such a tent graph.

We state an obvious property.

Lemma 4.8.6. *For any edge with ℓ direction changes there exists a tent subgraph with $\ell + 1$ paths in G . However, the tent graphs for different edges need not be disjoint.*

Path crossings. We define a *path crossing* of P_i and P_j to be a triple (i, j, v) where v is a vertex or crossing such that in any cyclic ordering of the incidence list of v there is exactly one edge of P_j between the two edges belonging to P_i .

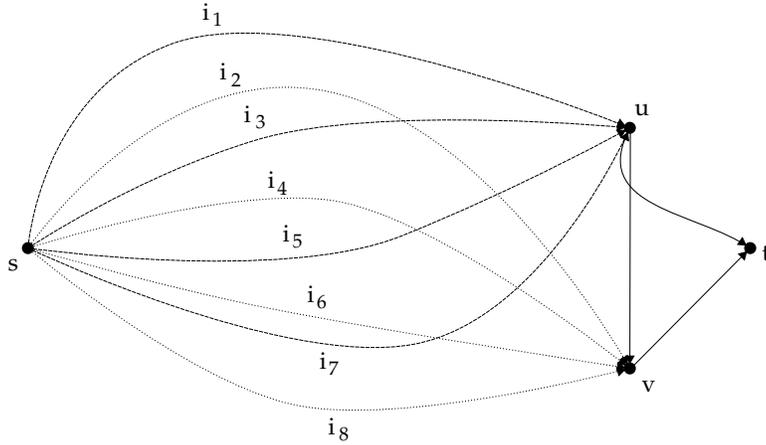


Figure 4.8.17: A tent subgraph with four odd and four even paths. This is a schematic presentation: The s - u - and s - v -paths are drawn disjoint although they may have common vertices or arcs.

If $[v, w]$ is a maximal interval such that $P_i|_{[v,w]} = P_j|_{[v,w]}$, then we shrink $P_i|_{[v,w]}$ to a vertex w' . If that w' is a path crossing of P_i and P_j , then we say that in the original graph w is a path crossing of P_i and P_j .

If P_i and P_j are anti-parallel then the first common vertex on the older path (smaller index) is the path crossing. That is, joining an anti-parallel younger path (greater index) from the left is a path crossing.

Lemma 4.8.7. *In a tent graph of ℓ paths we have $\Omega(\ell^2)$ unique path crossings between odd and even paths.*

Proof. The minimum number of path crossings for each C_j is $\min\{|A_j|, |B_j|\} = \min\{\ell/2 - 2j - 1, 2j - 1\}$. That is, for any $d \in \{0, 2, 4, \dots, \ell/4\}$ we have a C_j that is crossed by at least d paths. And the minimum number of unique path crossings of $P_{i_1}, P_{i_2}, \dots, P_{i_\ell}$ is

$$\sum_{d=0}^{\ell/8} 2d = 2 \binom{\ell/8}{2} \in \Omega(\ell^2)$$

□

However the number of path crossings is not a parameter of the instance. And its relation to parameters of the instance – for example to the number

of crossings – is unclear. There may be many path crossings in one crossing or even in one vertex.

For example, an instance may have k crossings but the algorithm finds $\binom{k}{2}$ path crossings (see Figure 4.8.18). In the non- s - t -planar setting even one crossing can lead to this number of path crossings as can be seen in Figure 4.8.19.

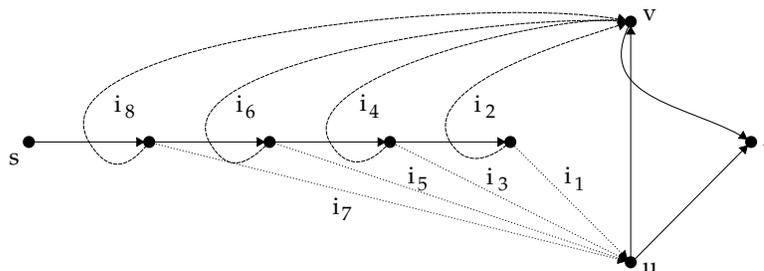


Figure 4.8.18: This is a prototype for an instance class. This prototype has the parameter value $\ell = 4$. The number of crossings is ℓ for all instances. The algorithm uses $2\ell - 1$ direction changes on $\{u, v\}$ and $\Omega(\ell^2)$ path crossings.

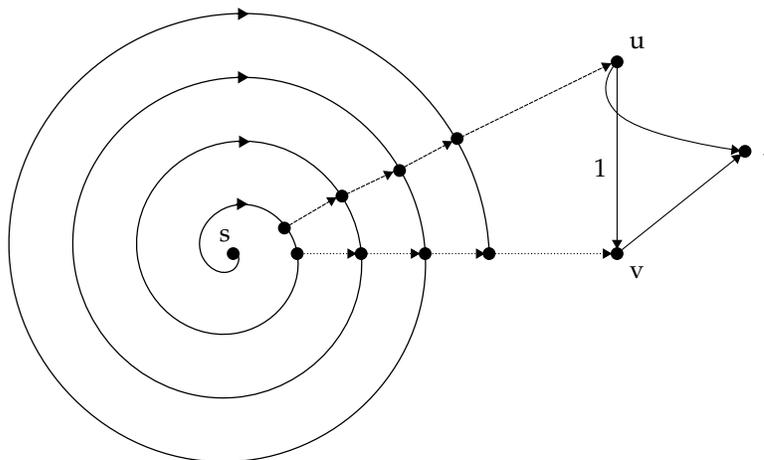


Figure 4.8.19: This is a prototype for an instance class. This prototype has the parameter value $\ell = 4$. On these instances the algorithm uses $\Theta(\ell)$ direction changes on $\{u, v\}$ and $\Omega(\ell^2)$ path crossings. The instances are not s - t -embedded and for all values of ℓ there is only one crossing.

This last example also shows that the number of crossings is not a good indicator for the number of direction changes: The number of crossings is constantly 1 while the number of direction changes is in $\Theta(n)$ for these

instances. However, this might be rooted in the fact that the instances are not s - t -embedded.

This is supported by the following argumentation. In order to make these instances s - t -embedded we have to connect s to a vertex s' in the infinity face. Then the edge $\{s, s'\}$ has at least ℓ crossings.

We conjecture that the number of direction changes is in $O(k)$ where k is the number of crossings but we were not able to find a proof for this.

Chapter 5

Blocking Flows

In this chapter we give a short overview of Dinic' blocking flow algorithm and then try to use its ideas to find a provably fast algorithm for the maximum s - t -flow problem in nearly-planar graphs.

5.1 Dinic' Algorithm

Dinic' blocking flow algorithm was published in 1970 [Din70]. At that time there was but one other strongly polynomial algorithm for the maximum s - t -flow problem – the Edmonds-Karp algorithm [EK72]. While the latter algorithm is an augmenting path algorithm with the restriction to augment along a shortest path in each iteration Dinic' algorithm is conceptually different.

The blocking flow algorithm operates in phases. In each phase a level graph is constructed. This is a subgraph of G containing only shortest s - t -paths. And then a flow saturating all augmenting paths in the level graph is found, a so-called blocking flow.

More specifically, the *level graph* G_f^L of G_f is the graph with the vertex set $V(G)$ and arcs

$$E(G_f^L) := \{(u, v) \in E(G) : c_f(u, v) > 0 \text{ and } \text{dist}_{G_f}(s, u) + 1 = \text{dist}_{G_f}(s, v)\}.$$

We see that in G_f^L the vertex set of G can be partitioned in sets of vertices all having the same distance from s , so-called *levels*. This motivates the name “level graph”. The arcs in G_f^L are exactly those arcs in G_f that connect a level to the next higher level, where levels are ordered ascendingly by the distance from s .

Therefore the level graph is acyclic. In particular, for each arc in G only the arc itself or its reverse arc but not both are contained in G_f^L . Now we define a *blocking flow* as a flow in a network that saturates all s - t -paths. Note that a blocking flow is not necessarily maximum. Figure 5.1.1 gives an example.

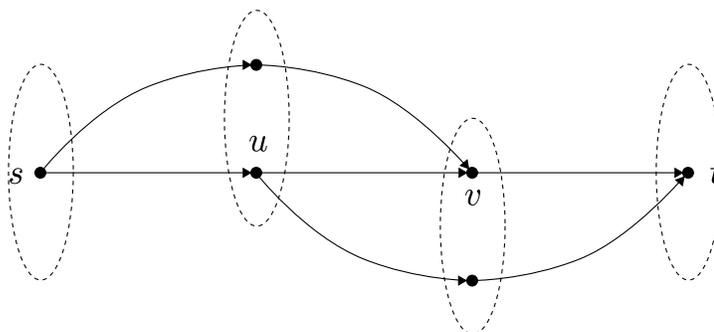


Figure 5.1.1: This is a level graph. The levels are shown with dashed ellipses. Assume that all capacities are 1. The straight through path (s, u, v, t) is a blocking flow. This flow is not maximum, though, because the path (s, v, u, t) is an augmenting path in the flow's residual graph.

In each phase we find a blocking flow in the level graph. In the next phase we construct the level graph again for the resulting residual graph and so on. This process is similar to the Edmonds-Karp algorithm in that the lengths of the augmented paths increase monotonously. And it terminates if t is not reachable from s in G_f anymore. Since the length of an s - t -path cannot exceed $n - 1$ Dinic's algorithm stops after at most $n - 1$ phases.

An augmenting path can be found in time $O(m)$ and each augmenting path saturates at least one edge. Thus a blocking flow in an acyclic network can be found in time $O(mn)$ as Dinic [Din70] notes. Using the dynamic trees data structure this bound can be improved to $O(m \log n)$ [ST83].

5.2 Generalization

Here we generalize the notion of a "blocking flow". We say a *blocking flow* is a flow consisting of a set of augmenting paths that is maximal given some constraining property.

For example for Dinic's algorithm the relevant property is that each path in the set is as long as the shortest augmenting path in the residual graph.

In order to develop a new algorithmic idea we try to build a blocking flow algorithm in the above sense that uses the uppermost path algorithm as a subprocedure in each phase. That is, the complete algorithm runs in only one blocking flow phase for s - t -planar instances. That way we can use further phases to solve difficulties brought about by non-planarity. How can we define a blocking flow for this case?

We recall some properties of the paths augmented by the uppermost path algorithm.

1. The paths are left-first search paths.
2. There are no reverse augmentations in the set of paths. That is, if (u, v) is contained in a path then (v, u) is in no path.
3. No two paths cross each other in the sense of a path crossing (see Subsection 4.8.7).

By combining properties 1 and 2 or 1 and 3 we get algorithmic ideas which we describe in the following sections.

5.2.1 No Crossing Paths

The uppermost path algorithm and Hassin's algorithm on s - t -planar instances both find a maximum s - t -flow along augmenting paths that are ordered geometrically. That is the paths do not cross. In fact the two algorithms find the same maximum s - t -flow [KNK93].

We feel that a geometrically ordered set of augmenting paths has advantages in nearly-planar graphs. Such graphs consist in large part of planar subgraphs. In order to solve these subgraphs efficiently, geometrically ordered paths are essential. Then the few non-planarities that are present in nearly-planar graphs do not justify wildly crossing paths.

Therefore consider a left-first search augmenting path algorithm with the blocking flow restriction that no path may cross an existing path or itself. One such "blocking flow" can be computed asymptotically as fast as the uppermost path algorithm. But does one blocking flow solve the given instances correctly? The answer is "no" even if we consider instances which have a maxflow that can be decomposed in a set of geometrically ordered augmenting paths (see Figure 5.2.2).

Thus we have to compute blocking flows repeatedly. More specifically, we compute each blocking flow on the residual instance of the previous one until

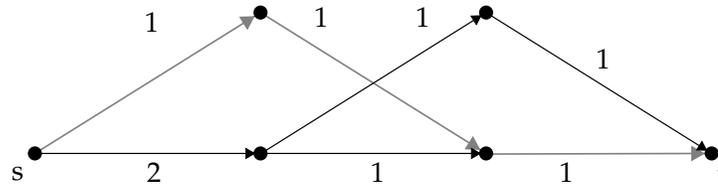


Figure 5.2.2: The instance has two non-crossing augmenting paths which give a maximum flow value of 2. But the first blocking flow consists only of the gray path with value 1.

there are no more augmenting paths. This way we always find a maximum flow. But how many blocking flows do we have to compute?

Each blocking flow consists of a set of non-crossing augmenting paths which make up a planar subgraph of G . Thus we may suspect that the number of blocking flows is bounded by the size of a planar subgraph decomposition, that is by the graph thickness. Figure 5.2.3 shows that this hypothesis is false.

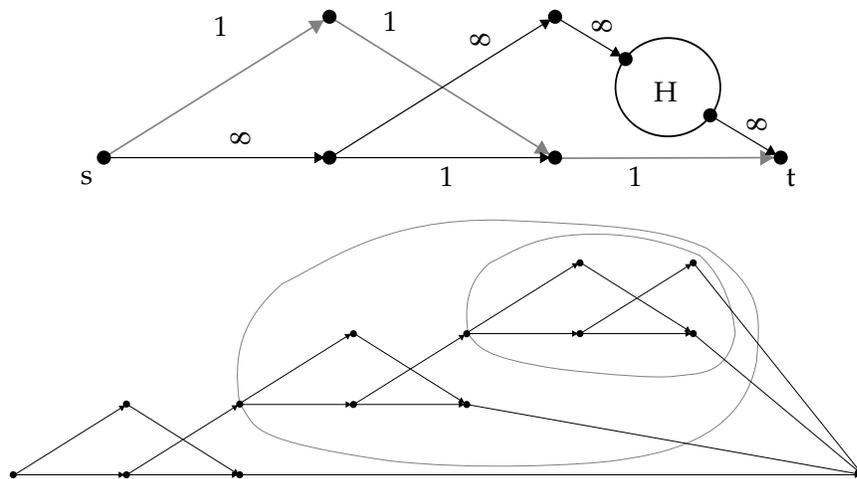


Figure 5.2.3: Based on Figure 5.2.2 we construct an instance that has graph thickness 2 but needs $o(n+m)$ blocking flow computations: We repeatedly insert the graph itself at the place indicated by H (upper illustration). Below we see the graph after two insertions.

Since the uppermost path algorithm finds on any instance only a set of geometrically ordered paths, we can describe our new algorithmic idea thus:

Repeatedly run Berge's uppermost path algorithm on the residual instance of the previous run.

And this is the same procedure we have considered in Section 3.4.1. Therefore everything said in that section also applies here and vice versa.

5.2.2 No Reverse Augmentations

The uppermost path algorithm touches each edge at most once. In particular, no edge is augmented in its reverse direction. In the non-planar case, however, we may have to reverse augment some edges (compare Figure 5.2.4).

Reverse augmentation needs time and is unnecessary: Given a flow we can always decompose it into a set of augmenting paths that contain each edge only in its forward direction. So let us look for such a set of augmenting paths by making this our blocking flow restriction: No two paths may contain an edge in opposing directions. In other words, no blocking flow may contain a direction change (see Section 4.1.1).

Again we ask: How many blocking flow computations does this algorithm require? And again we see that we have only described the algorithm from Section 3.4.1 in yet another way.

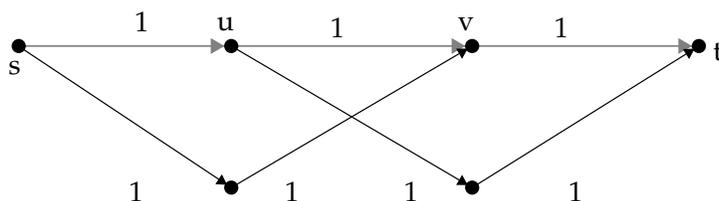


Figure 5.2.4: The uppermost path algorithm finds first the gray path. Then the arc (u, v) must be reverse augmented to find a maximum flow.

5.3 Karzanov's Algorithm

Karzanov's algorithm [Kar74] was the first algorithm that operated with preflows instead of flows. That is, the strong invariant that at all times a flow f is maintained is relaxed to maintaining a preflow f only. Recall the definition of a preflow from Section 1.5.

Considering augmenting paths, the big difference when working with preflows is that the paths need not be s - t -paths anymore but can have arbitrary start and end vertices.

The idea then is to iteratively “push” flow to a neighbouring level in Dinic’s level graph. There are various ways to do this [Che77, MKM78, Tar84].

5.3.1 Application to Nearly-planar Instances

As we have done earlier in this chapter here again we change the definitions used to describe blocking flow algorithms to describe our algorithmic idea.

Here we change the definition of the level graph. Conceptually we want the levels to contain no crossings. The advantage of this is that we can use a planar maxflow algorithm to push flow from one side of a level to the other side while the number of levels is greatly reduced. The most simple way to bring this about is to change the distance measure that is used to construct the level graph.

Let E^\times be the set of crossing edges. Now we define that the edges in E^\times have $d \equiv 1$ and all other edges have $d \equiv 0$. Thus only the crossing edges increment the distance. Using the resulting distance measure we define the level graph as usual. See Figure 5.3.5 for an example. As we can see the crossings separate the levels.

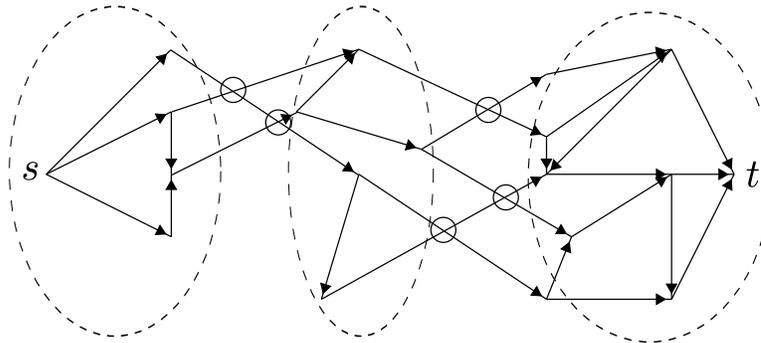


Figure 5.3.5: A maximum flow instance. Levels are surrounded by dashed ellipses and crossings are marked by circles.

As planned, we can now use a planar maxflow algorithm as a subroutine to push the flow between left and right hand side inside one level. Moreover, if there are k edges in E^\times then we have at most $k + 1$ levels and thus at most $(k + 1)^2$ iterations with k “pushes” each. Each such push needs a planar maximum flow computation in one level resulting in a total time of $O(k^3 n \log n)$ for our algorithm.

We do not go into greater detail here. In the next chapter we will describe Goldberg’s and Tarjan’s [GT88] preflow push algorithm. This can be interpreted as sort of an asynchronous version of Karzanov’s algorithm. They

form levels dynamically using an approximate *distance labelling* instead of synchronously computing the levels. Since preflow push algorithms are well investigated we formulate our algorithm as a preflow push algorithm.

Chapter 6

Generalized Preflow Push Algorithms

Again our aim is to find a maxflow algorithm that solves nearly-planar instances faster than previously known algorithms. Here we present a new class of algorithms that are preflow push algorithms on an abstract level but may use different subroutines on a lower level. In particular we use a planar maxflow algorithm as subroutine and achieve a worst case running time bound of $O(k^3 n \log n)$ where k is the number of crossings in the instance [HW07].

Overview of this chapter. We first describe the original preflow push algorithm then give an abstract definition of our generalization and in Section 6.3 an application as an algorithm that solves the maxflow problem with k crossings in time $O(k^3 n \log n)$ worst case. In Section 6.4 we give computational results for this application.

6.1 The Goldberg-Tarjan Algorithm

We give a short description of Goldberg's and Tarjan's algorithm [GT88] before we show how to generalize it in the next section.

Distance labels. The Goldberg-Tarjan-algorithm uses approximate distance labels. A function $d : V \rightarrow \mathbf{Z}_{\geq 0}$ is a *valid labelling* on V with respect to the residual capacities c_f if $d(t) = 0$ and $d(s) = |V|$ and we have

$d(v) \leq d(w) + 1$ for all augmenting edges (v, w) . If $d(v) = d(w) + 1$ then (v, w) is admissible.

We now formulate the algorithm.

Listing 6 A generic preflow push algorithm.

```

1:  $f$  is a preflow and all edges  $(s, v)$  are saturated and  $d$  is a valid distance
   labelling.
2: while there is an active vertex  $v$  do
3:   if there is a vertex  $w$  such that  $(v, w)$  is admissible then
4:     Push $(v, w)$ : Increase  $f(v, w)$  by  $\min\{c_f(v, w), ex_f(v)\}$ .
5:   else
6:     Relabel $(v)$ : Set  $d(v) := \min\{d(w) : (v, w) \text{ is augmenting}\} + 1$ .
7:   end if
8: end while

```

In this generic form the best running time bound is $O(n^2m)$. In order to improve upon the complexity we need to specify the order in which to choose the vertex and edge for the next **Push** operation.

Therefore we use a *push relabel scheme*. Examples for push relabel schemes are the FIFO implementation of the preflow push algorithm [GT88] or the highest distance rule.

For the FIFO implementation we maintain a stack of active vertices. When a vertex becomes active it is added to the top of the stack. And we take vertices from the bottom of the stack in Line 2 of Listing 6.

The highest distance (or highest label) rule, on the other hand, maintains the vertices in a queue sorted by their current distance label. In each iteration of the while-loop we take a vertex with maximum label from the queue.

To evaluate the complexity of a push relabel scheme we assume that for each push relabel scheme we are given a function n_{push} that returns the number of **Push** operations $n_{\text{push}}(H)$ that the scheme performs when operating on a graph H .

6.2 The Generic Algorithm

We generalize the Goldberg-Tarjan algorithm to a class of algorithms. All these algorithms follow Listing 6. An individual algorithm is defined by the vertices it operates on, by a specific definition of “augmenting” and a specific implementation of the operation **Push**. In Section 6.3 we give a concrete example of such an algorithm. Namely we show how to solve

instances with small crossing number.

In the original preflow push algorithm all vertices carry labels. Here we designate a subset V^\times of the vertices which we call *stop vertices*. Only stop vertices carry labels. Our generalized preflow push algorithm operates on the complete graph over the stop vertices K^\times . It proceeds exactly as shown in Listing 6. Note, however, that through the redefinition of “augmenting” and “Push” this listing now describes a number of different algorithms. Recall that in general an arc is said to be saturated if its residual capacity is zero. Thus it is saturated if and only if it is not augmenting. Here, analogously, we say an edge of K^\times is *saturated* if and only if it is not augmenting.

For the correctness and complexity proofs we need the following properties of `Relabel` which are immediate from Listing 6.

Lemma 6.2.1.

- After a call to `Relabel` d is a valid labelling.
- Each execution of `Relabel`(v) increases $d(v)$.

We will not define the procedure `Push` explicitly. We just give constraints that it must obey. This is necessary so that a specific algorithm of the generic class can define `Push` as needed.

`Push`(v, w) may only be called if (v, w) is admissible. After performing `Push`(v, w) the following three assertions must hold

P1 f is a preflow and

P2 d is a valid labelling and

P3 (v, w) is saturated or v inactive.

P4 If (v, w') for any w' is saturated before the push then it is saturated after the push as well.

P1 through **P3** are proven in [GT88]. **P4** is obvious for the standard preflow push algorithm. However, we must include it here because it is not obvious for all definitions of `Push`.

6.2.1 Correctness

Lemma 6.2.1 and assertions **P1** through **P3** yield the following properties. For a proof see [GT88].

Properties 6.2.2. (a) At all times d is a distance labelling and f is a preflow.

(b) At no time is there an augmenting s - t -path in G .

(c) Each call to $\text{Relabel}(v)$ increases $d(v)$ and $d(v)$ is never decreased.

(d) At all times $d(v) \leq 2|V^\times| - 1$ holds.

The algorithm terminates when there are no active vertices. By definition f is then a flow. And Property 6.2.2b implies that f is maximum. This shows the correctness in case it terminates. We are now going to show that the algorithm in fact terminates and how long that takes.

6.2.2 Complexity

The complexity depends on the number of relabels and pushes performed by the push relabel scheme on the complete graph over the stop vertices. It depends further on the specific implementation of Push and on how fast augmenting edges incident to a vertex can be found. Let T_{push} be the time bound for one Push operation in the concrete implementation. And T_{scan} the time needed to find the augmenting edges in K^\times that are incident to a vertex.

For k stop vertices the number of Relabel operations is always less than k^2 by Properties 3 and 4. The running time of the algorithm is then $O(n_{\text{push}}(K^k)T_{\text{push}} + k^2T_{\text{scan}})$.

The set of Push operations is usually partitioned into two subsets, the *saturating* and the *non-saturating Pushes*. If the edge (v, w) is saturated after $\text{Push}(v, w)$ that the Push is saturating otherwise it is non-saturating. We recall the following lemma which is the same as Lemma 3.9 in [GT88].

Lemma 6.2.3. *The number of saturating Push operations is at most $2nm$.*

In the standard preflow push algorithm an admissible edge stays admissible until a Push on that edge or a Relabel on an incident vertex is performed. In the general case this is not so. A Push somewhere in K^\times may saturate another edge in K^\times , as well. Furthermore it may be a complex operation to find out whether an edge in K^\times is augmenting and thus inefficient to carry out this operation before each Push .

Note, that for a $\text{Relabel}(v)$ operation we need to check all edges in K^\times incident to v . Thus after $\text{Relabel}(v)$ we know for all these edges whether they are augmenting or not.

However, in general, we do not know whether the edge over which we push is augmenting or not. Therefore we have a third kind of Pushes, the 0-Pushes, that is Pushes that transfer no flow. These are saturating Pushes, as well, but are not included in the analysis in [GT88]. For the standard preflow push algorithm this is no problem but in our generalized case it might be, as we have seen above.

For the analysis of 0-Pushes we rely on a property that holds for all push relabel schemes known to us. Namely that if one Push originates from a vertex v then all subsequent Pushes originate from v , as well, until v is relabelled or becomes inactive.

With this property we can prove the following lemma. This lemma shows that for the number of 0-Push operations the same bound holds as for the number of saturating Push operations. Therefore we need not consider the 0-Push operations further.

Lemma 6.2.4. *The number of 0-Push operations is at most $2nm$.*

Proof. We consider the Push operations originating at a vertex v . Of these we take a maximal interval I not interrupted by a Relabel(v) operation.

At the beginning of I it is known for all edges leaving v whether they are augmenting or not. That is because we are either at the very beginning of the algorithm when no Push has been performed yet or we have just performed a Relabel(v) operation.

Now at any Push(v, w) operation in I we may find that we can transfer no flow because a previous Push in I has saturated (v, w) . Then for all subsequent Push(v, w) operations in I we know by **P4** that (v, w) is not augmenting and we need not try a Push. That is, for each edge leaving v we have only one 0-Push in I . Since the number of intervals is less than or equal to the number of Relabels for any vertex, there can be at most $2n - 1$ 0-Pushes over each edge. This yields the assertion. \square

We now give a running time analysis for two well known push relabel schemes. The FIFO scheme (see [GT88]) has the number of pushes $n_{\text{push}}(G) \in O(nm + n^3)$. Thus our algorithm needs time in $O(k^3 T_{\text{push}} + k^2 T_{\text{scan}})$.

The best bound for the highest distance scheme is $O(mn + n^2 \sqrt{m})$ (see [CM89]). Since we operate on a complete graph this gives asymptotically the same complexity for our algorithm as the FIFO scheme.

6.3 Nearly-planar Maxflow

We assume that we are given a planar embedding of a planar subgraph $G_p = (V, E_p)$ of G and a set $\{e_1, \dots, e_k\}$ of *crossing edges* such that $E = E_p \cup \{e_1, \dots, e_k\}$. Note that the running time depends on the number k of these crossing edges not on the crossing number. The number of crossings may be greater than k . Let V^\times be the set of vertices incident to crossing edges.

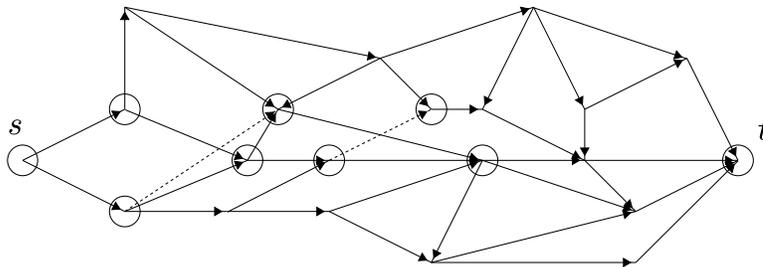


Figure 6.3.1: An instance with two crossing edges. Stop vertices are surrounded by circles.

The stop vertices in this application are s , t and the *crossing vertices* in V^\times . See Figure 6.3.1 for an example. An edge (v, w) in K^\times is *augmenting* if one or both of these is true: The edge (v, w) is an augmenting crossing edge or there is an augmenting v - w -path in G_p . We now give the description of Push.

Listing 7 The procedure Push for the nearly-planar case.

```

1: procedure Push( $v, w$ )
2:   if  $(v, w)$  is a crossing edge then
3:     Increase  $f(v, w)$  by  $\min\{c_f(v, w), ex_f(v)\}$ .
4:   else
5:     Let  $c_p(v, w)$  be the value of a maximum  $v$ - $w$ -flow in  $G_p$ .
6:     Compute a planar  $v$ - $w$ -flow with value  $\min\{c_p(v, w), ex_f(v)\}$  in  $G_p$ .
7:   end if
8: end procedure

```

We have now fully described the implementation of the algorithm in this case. We need to show that properties **P1** through **P3** hold. **P1** and **P3** are obvious from Listing 7 and **P2** follows from Corollary 6.3.3 below and **P4** from Lemma 6.3.1.

Lemma 6.3.1. *If there is no augmenting v - w -path in G_p then a Push(v, w') does not produce an augmenting v - w -path in G_p .*

Proof. There is a saturated v - w -cut $(X, V \setminus X)$ in G_p such that $v \in X$. In order to make a v - w -path augmenting it is necessary to augment an edge (y, x) in the cut such that $y \in V \setminus X$ and $x \in X$. If this is to be done by a $\text{Push}(v, w')$ operation then it would require an augmenting path containing the vertices (v, y, x, w') in this order. And that in turn would require an augmenting v - y -path. However, this path cannot exist since v and y lie on opposite sides of the saturated cut. Therefore $\text{Push}(v, w')$ cannot make any v - w -path augmenting. \square

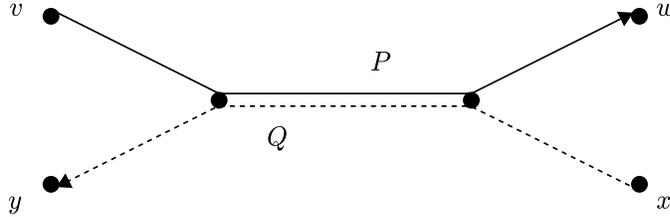


Figure 6.3.2.

Lemma 6.3.2. Consider four stop vertices v, w, x and y in G such that an augmenting v - w -path P exists and d is a valid labelling and $d(v) = d(w) + 1$. After we augment maximally along P there is no augmenting x - y -path or $d(x) \leq d(y) + 1$.

Proof. Assume that after augmenting P there is an augmenting x - y -path Q . If $E(P) \cap E(\overline{Q}) = \emptyset$ then $d(x) \leq d(y) + 1$ holds since d is a valid labelling. Therefore assume $P \cap \overline{Q} \neq \emptyset$. Thus before the augmentation there was a x - w -path and a v - y -path. Since d was a valid labelling we have

$$d(x) \leq d(w) + 1 \quad \text{and} \quad d(v) \leq d(y) + 1.$$

Then $d(v) = d(w) + 1$ implies

$$d(x) \leq d(w) + 1 = d(v) \leq d(y) + 1.$$

Observe that this proof also holds if $v = y$ or $x = w$. \square

Since we can decompose the planar maxflow found by Push into single augmenting paths this yields

Corollary 6.3.3. Performing $\text{Push}(v, w)$ does not make d invalid.

The complexity of this algorithm is then $O(k^3 n \log n)$ since a planar maximum flow computation takes time $O(n \log n)$ and all augmenting edges incident to a vertex in K^\times can be found by a graph scan in time $O(m) = O(n + k)$.

Unlike most algorithms for the non-planar case our algorithm does not have a factor m in its running time bound. Instead we separate the edges in planar and non-planar with respect to a given embedding of the vertices. The running time analysis then uses the number of non-planar edges k and the upper bound $3n - 6$ on the number of planar edges.

Perfectly analogous to this application we can also solve nearly series parallel maxflow instances. Again we divide the graph in a series parallel part and a set of edges. All definitions and arguments carry over from the nearly-planar case. We just substitute “planar” by “series parallel”.

The planar case. We might wonder what happens if the original instance is planar? In this case there are only two stop vertices, s and t . Thus K^\times contains only the edge $\{s, t\}$ and only one push namely along this edge must be performed. Obviously this is a planar push. So in this case the algorithm reduces to just one graph scan in order to find the initial labelling and one ordinary planar maximum s - t -flow calculation. No huge overhead is incurred.

Nearly-planar graphs. We now propose a formal definition of nearly-planar graphs. Our algorithm solves instances with $k \in O(\sqrt[3]{n})$ faster than the currently known algorithms. We might define exactly these instances as nearly-planar but this seems a bit arbitrary. Why to the third power and not to the fourth?

Furthermore our definition of k depends on the embedding and is not really a parameter of the graph. Therefore we use the fact that the graph genus g is always greater than our k or the pairwise crossing number. Thus if we base our definition on the graph genus instead more graphs are classified as nearly-planar. We say

Definition. *A class of graphs is nearly-planar if their graph genus $g > 0$ and their genus is sublinear in the number of vertices, that is $g \in o(n)$.*

6.4 Computational Results

We have collected only a few data points because the algorithm is very slow in practice (Figure 6.4.3). The constants incurred seem to be huge. A great part of this overhead is probably due to the dynamic trees data structure.

However there is another thing which contributes to the bad performance. As we have seen above our algorithm is *in the worst case* asymptotically better

than all other known algorithms on instances with up to $\Theta(\sqrt[3]{m})$ crossing edges. Now let us look at the *best case* running times in the case that the instance allows a non-zero flow.

The Goldberg-Tarjan algorithm in the best case exits after the initial labelling and one push path from s to t that is after time $O(k + n)$. Our algorithm, however, in each planar **Push** step runs a planar maxflow algorithm which at the very least builds a spanning tree of G_p using the dynamic trees data structure. That is, in the best case with only $O(k)$ planar pushes our algorithm needs time in $O(kn \log n)$. And even in this simple case it uses dynamic trees.

Thus there may be classes of instances near to the best case described above on which the Goldberg-Tarjan algorithm runs *asymptotically* faster than our algorithm.

6.5 Future Improvements

Our algorithm may benefit from better schemes. The dynamic trees implementation of the Goldberg-Tarjan algorithm [GT88], however, is useless for our algorithm since the subgraph induced by V^\times is not connected in general. In any way, we cannot hope for improvements by using dynamic trees in the push relabel scheme since the planar maxflow algorithms we apply already use the dynamic trees data structure internally.

In the nearly-planar algorithm the procedure **Push** can make use of a single-source multiple-sink planar maxflow algorithm in order to push flow simultaneously to all admissible crossing vertices. The currently best known solution to this subproblem is to do a single-source single-sink planar maxflow computation for each sink. But even if there were a faster way to do this then our algorithm would only benefit from it if there was a scheme that can form groups of planar pushes to do simultaneously. If the number of sinks per push is constant on average then the number of maxflow computations cannot be reduced.

The planar maximum flow algorithms [Wei97, BK06] perform some time-consuming preprocessing on start up. If we do this preprocessing only once at the beginning of our nearly-planar algorithm then this improves the performance — although not asymptotically. Thinking further in the same direction, we can interpret our algorithm as a kind of planar maxflow algorithm where the source and the sink change every now and then and sometimes crossing edges are touched. The advantage of this is that there might be an amortized time argument that lowers the time per planar push

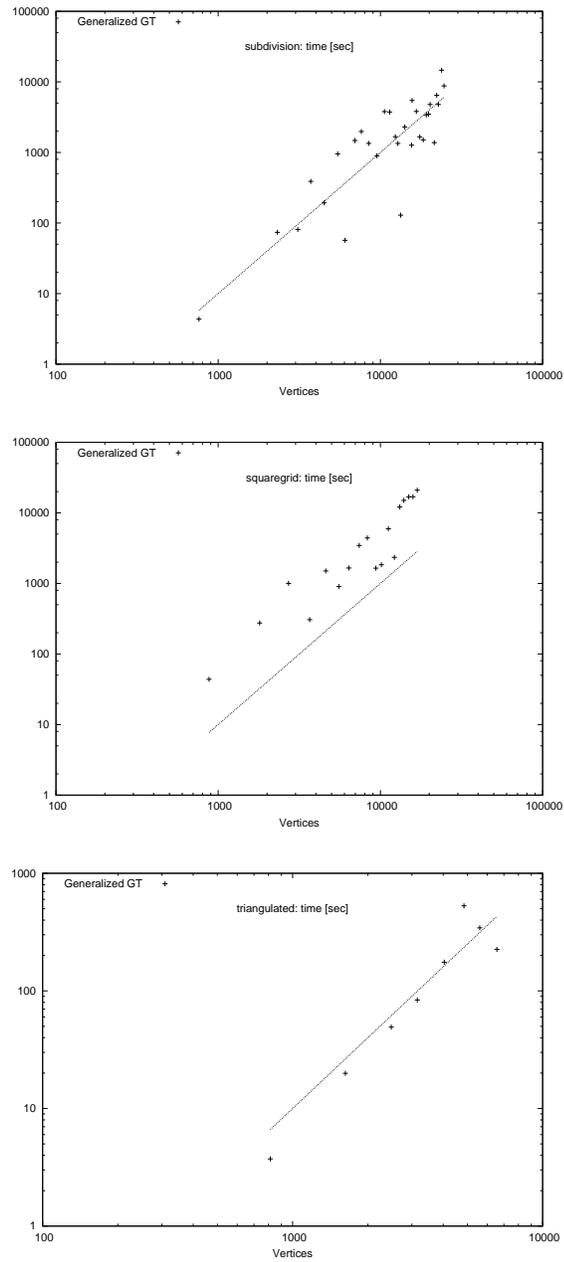


Figure 6.4.3. Log-log scale results of our algorithm on the benchmark data sets. The line indicates a function proportional to $y = x^2$.

to below $O(n \log n)$. However, at the moment no such argument is known to us.

Section 6.2 defines a class of algorithms. Of this class we describe but one specific example. At the moment we are looking for further useful algorithms in this class to solve maximum flow related problems.

Chapter 7

Dual Approaches

There is a well-known connection between the maximum flow problem in planar graphs and the shortest path problem in duals of planar graphs. We first describe this connection and existing algorithms based on it. Then we try to derive an algorithm that can solve arbitrary instance but is asymptotically faster on nearlyplanar instances.

7.1 Shortest Paths in Planar Dual Graphs

The shortest path problem requires lengths on edges. Let $d : V \times V \rightarrow \mathbf{R}_{\geq 0}$ be the *graph distance* induced by these lengths, that is $d(u, v)$ is the length of a shortest u - v -path. Now we can define a *shortest path labelling* from a source s . This is a function $\phi : V \rightarrow \mathbf{R}$ that assigns a non-negative real number to each vertex and observes the following conditions:

S1 $\phi(s) = 0$

S2 $\forall (v, w) \in E(G) : \phi(w) - \phi(v) \leq c(v, w)$

S3 $\forall v \in V(G) : \phi(v) \geq d(s, v)$

Shortest paths in planar graphs can be computed in linear time using the algorithm by Klein et. al. [KRRS94]. Since this algorithm is rather complex we will use Dijkstra's algorithm [Dij59] for our argumentation. This runs in time $O(m + n \log n)$ if implemented with Fibonacci heaps [FT87] as priority queue.

We compare the shortest path properties to the conditions imposed on a maximum flow function f .

edges [Has81]. For an edge e and its dual (v^*, w^*) he defines

$$f(e) := \phi(w^*) - \phi(v^*). \quad (7.1)$$

If f is a flow function then it has to fulfill **F1** and **F2**. We prove this now. **F1** is implied by **S2** and the zero capacity on reverse edges. For **F2** we consider the incidence list of a primal vertex v as a dual cycle C^* . Since C^* is a shortest v^*-v^* -path for any of its vertices it has length zero. That is

$$0 = \sum_{(v^*, w^*) \in E(C^*)} \phi(w^*) - \phi(v^*) = \sum_{(v, w) \in E(v)} f(v, w) \quad (7.2)$$

which is equivalent to **F2**.

As noted by [KNK93] Hassin's algorithm finds the same solution as the uppermost path algorithm.

7.3 Generalization to the Non-planar Case

If the graph is not planar then it is not clear how to define a dual graph. We first need an embedding in some topological space. In Section 3.5 we have seen a way to construct an embedding of G . In the next sections we will use that embedding to solve the maxflow problem dually. In Section 7.6 we give another method how to derive such an embedding and a corresponding dual algorithm.

7.4 Trajectory Dependent Shortest Paths

We recall a construction from Section 3.5: We planarize G by adding vertices in the place of crossings and thus get a planar graph G^\times . Let G^* be the planar dual graph of G^\times .

Hassin's method is designed for planar graphs. There each edge is contained at most once in any cut. Or, considering the dual graph, each edge is traversed at most once by a dual path.

In non-planar graphs this is not so: In Figure 7.1.1 we see that the path crosses (u, v) twice, first forward then backwards. And the edge is not contained in the corresponding cut. By the Jordan curve theorem this behaviour is impossible in planar graphs.

For non-planar graphs we have to modify the shortest path procedure to allow for a shortest path containing an edge various times.

Trajectory-dependent lengths. From now on we identify primal and dual edges to simplify the notation. The idea is this: When looking for shortest paths in the dual graph the length of a reverse crossing edge $(w, v) \in E$ is $-c(v, w)$ when the s^*-w^* -path plus (w, v) contains the edge as many times forward as backwards and zero otherwise. The length of a forward edge $(v, w) \in E$ is $c(v, w)$ when the s^*-w^* -path plus (w, v) contains the edge as many times forward as backwards and zero otherwise.

We note here that of course the path may not contain any edge in G^\times twice. But since a crossing edge in G corresponds to more than one edge in G^\times . Therefore it is possible that the path contains an edge in G more than once.

Thus we may have negative capacities on edges. Worse still we may have cycles with negative total length. The most challenging problem however is to cope with trajectory-dependent arc lengths: Two different v^*-w^* -paths may not see the same length on an arc (v^*, w^*) . See Figure 7.4.2 for an example.

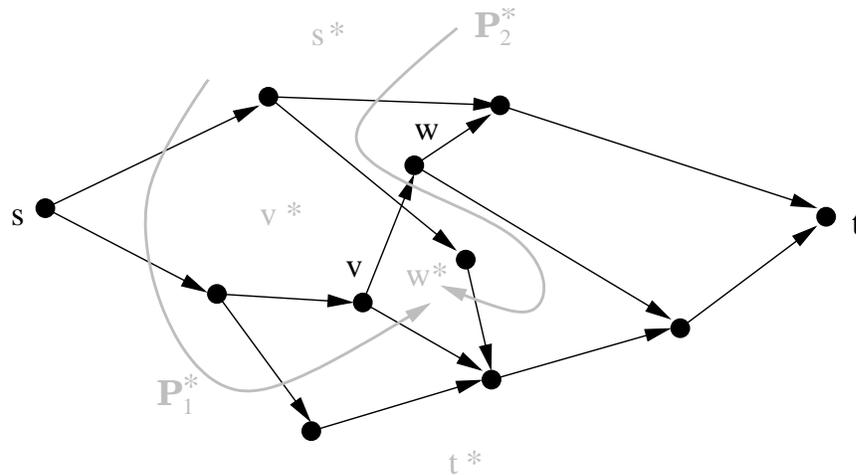


Figure 7.4.2: Two s^*-v^* -paths P_1^* and P_2^* are drawn. For P_1^* the arc (w^*, v^*) is a reverse arc with capacity zero. For P_2^* it has capacity $-c(v, w)$.

For this reason a simple breadth-first search such as Dijkstra's Algorithm will not always find a shortest path. That kind of algorithm relies strongly on the following property of the network:

Any $v'-w'$ -subpath of a shortest $v-w$ -path is a shortest $v'-w'$ -path.

Here this may not be the case if the subpath sees different arc capacities than the $v-w$ -path.

Before we describe an algorithm that solves the shortest path instance with these edge lengths we need some more notation.

Trajectories. We say the *trajectory* of a path P^* is a multiset $\tau(P^*)$ of the crossing edges in G that P^* traverses in forward orientation. That is $\tau(P^*) \subset \overline{E^\times}$. Obviously $|\tau(P^*)|$ is smaller than $|E^\times|$. Let $\tau_{(v^*, w^*)}(P^*)$ be the *multiplicity* of (v^*, w^*) in $\tau(P^*)$. $\tau_{(v^*, w^*)}(P^*)$ is zero if the arc is not contained in $\tau(P^*)$, is incremented each time P^* traverses the arc forward and decremented each time P^* traverses the arc backwards.

Now we can formulate the above idea of trajectory-dependent lengths more clearly: For any arc $(v^*, w^*) \in E(G^\times) \cap E(P^*)$ with the corresponding primal crossing arc (v, w) the capacity is

$$c(v^*, w^*) = \begin{cases} 0 & \text{if } \tau_{(v, w)}(P^*|_{[s^*, w^*]}) = 0 \\ c(v, w) & \text{otherwise.} \end{cases}$$

And for any arc $(w^*, v^*) \in \overleftarrow{E}(G^\times) \cap E(P^*)$ with the corresponding primal crossing arc (v, w) the capacity is

$$c(w^*, v^*) = \begin{cases} -c(v, w) & \text{if } \tau_{(v, w)}(P^*|_{[s^*, v^*]}) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

We say a dual path edge (v^*, w^*) has *normal capacity* if it has forward capacity $c(v^*, w^*)$ forward and zero backward capacity.

We now abstract the notion of trajectories and consider it independent of any specific path. We say v^*-w^* -trajectory for the trajectory of a v^*-w^* -path. And we consider the set $T(v^*)$ of all possible v^*-w^* -trajectories. That is the set of trajectories induced by all possible v^*-w^* -paths.

At any dual vertex v^* we need not only one shortest path label but many. More accurately we need one shortest path label for each possible s^*-v^* -trajectory. In the worst case this can be as large as $2^{|E^\times|}$.

To reduce the number of labels we define the following dominance relation for v^*-w^* -paths:

$$P_1^* \succ P_2^* \text{ if } c(P_1^*) > c(P_2^*) - \sum_{e: \tau_e(P_2^*) > 0} c(e).$$

Thus P_2^* is dominated if for any w^*-u^* -path P_3^* $c(P_1^* + P_3^*) > c(P_2^* + P_3^*)$. This allows us to remove dominated labels at vertices when we encounter them. In practice, however, this does not reduce the number of labels much.

We implemented this total enumeration of paths. And we observed an exponentially large number of labels at each vertex. Thus the running time of the algorithm presented in this section is not competitive.

7.5 Inhibit Flow Turn-off

In Section 7.4 we saw an idea for a shortest path labelling algorithm, that in its core enumerates all cuts and uses many labels at each vertex. We now describe an algorithm that at labelling time tries to decide which cut is the most relevant and thus maintains only one label at each vertex. If later on the algorithm's decision turns out to be wrong backtracking occurs.

If we apply a dual shortest path algorithm to G^\times then it does not necessarily find a feasible flow for G . Like the uppermost path algorithm on G^\times it ignores that the sub-edges in $E(e)$ that arose from the crossing edge e are not independent. More specifically the flow on all sub-edges in $E(e)$ must be the same for any crossing edge e in G . Speaking figuratively, the flow may not turn off at a crossing. In Section 3.5 we already saw an unsatisfactory way to remedy this. Here we present a shortest path labelling algorithm that observes this additional constraint. We call this constraint *flow conservation at crossings*.

First we make a further assumption on the input instance. We require that in each crossing exactly two edges cross. If we have a crossing with more than two edges then we can split it into a number of crossings with two edges each. This can be done without changing the combinatorial embedding.

The idea of the algorithm is to label the four faces incident to a crossing such that the flow conservation at the crossing is maintained at all times. To achieve this we label normally in a Dijkstra like fashion beginning at the source s^* with value 0. When we have labelled three faces incident to a crossing v then flow conservation at that crossing gives the label value for the fourth face w^* .

We then add w^* to the set of sources. This ensures that the label is not overwritten by the Dijkstra labelling routine. Now anytime we find a better label for a face incident to v we have to update the label at w^* to observe flow conservation at v . Since other label values can depend on w^* 's label it might be necessary to relabel the complete graph in the worst case.

If G is planar this algorithm is the same as Dijkstra's. If we have exactly one crossing in G then we have to add one face w^* to the set of sources. As there are no other crossings we never have to change the label at w^* after it

becomes a source. In the case of k crossings we have k additional sources and it might be necessary to relabel all $k - 1$ other sources each time a source is updated. In the worst case the dual graph has to be labelled 2^k times leading to a running time of $O(|E(G^\times)| \log |E(G^\times)| 2^k) = O((m + k) \log(m + k) 2^k)$.

Since this is exponential in the number of crossings we are not going to pursue this idea further.

7.6 Hyperfaces

Again we start with the planar embedded graph G^\times . In order to achieve flow conservation at crossings we now do not label faces in the dual graph but larger regions which we will call hyperfaces. For all sub-edges $e' \in E(e)$ of a crossing edge e in G the difference between the label on e' 's right side and the label on its left side must be the same. This is equivalent to the flow conservation at all crossings that lie on e .

Therefore we combine all faces in the dual graph that are adjacent to e 's left side into one *hyperface*. And all faces on e 's right side form another hyperface. This is the general idea. We give a better definition of hyperfaces shortly.

Let $H(G)$, or H for short, be the set of all hyperfaces. Each hyperface is a set of faces in G^\times . Conversely each face may be contained in more than one hyperface. Let $H(v^*)$ be the set of hyperfaces that contain v^* . We require that each face be part of at least one hyperface. Thus $H(v^*)$ is never empty.

We say a *labelling* is an assignment of non-negative numbers to all objects in a set. Now we define the label of each face v^* to be the sum of the labels of the hyperfaces in $H(v^*)$. Thus a hyperface labelling $H \rightarrow \mathbf{R}_{\geq 0}$ induces a labelling on the faces $F(G^\times) \rightarrow \mathbf{R}_{\geq 0}$. Since each face is part of a hyperface we need not store the faces' labels explicitly. And by Equation 7.1 a labelling on the faces of G^\times induces flow values on the edges in G^\times .

Above we wrote that we combine all faces in the dual graph that are adjacent to an edge e 's left side into one hyperface. Obviously there are cases where this simple rule does not yield what we want. For an example consider Figure 7.6.3. The dashed region consists of all faces on the right side of the crossing edge (u, v) . However this region contains only one quadrant at the crossing x . Therefore if we only label this region the resulting flow does not obey flow conservation at x . In this example the whole region below the path (u, v, w) must be one hyperface.

Now we formally state all conditions for hyperfaces:

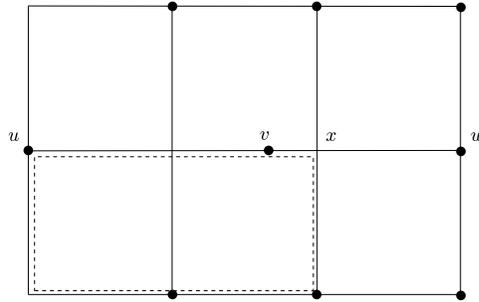


Figure 7.6.3.

- H1** The border of a hyperface consists of edges in G and does not cross itself.
- H2** Each hyperface is minimal.
- H3** Each edge in G is incident to at least two hyperfaces: one on each side of the edge.

The minimality condition **H2** means that no subset of a hyperface fulfills condition **H1**. Thus a face that is not incident to crossing edges will be contained in its “personal” hyperface which contains only the one face. We call these faces *simple faces*. Nonetheless a simple face may be contained in a larger hyperface (see Figure 7.6.4). In the planar case we have only simple faces.

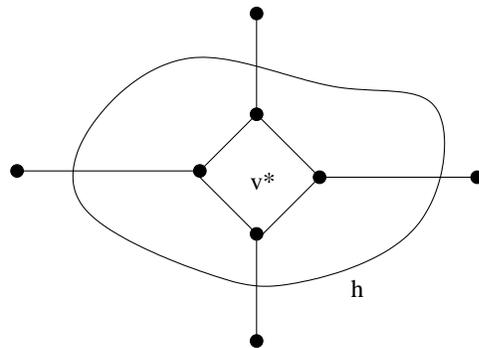


Figure 7.6.4: This subgraph contains a hyperface h which in turn contains a simple face v^* .

A simple consequence of these conditions is

Lemma 7.6.1. *Between two vertices on the border of a hyperface h there is no path in G through the interior of h which does not cross itself. In particular, h contains no simple face that shares an edge with the border of h .*

Proof. Assume for a contradiction that there is a non-self-crossing path through the interior of h . This path then divides h in two parts both of which fulfill **H1**. Therefore h violates **H2**. \square

The next result shows why the idea of a hyperface labelling is useful when considering maximum flow in the dual graph.

Lemma 7.6.2. *Any labelling $H(G) \rightarrow \mathbf{R}_{\geq 0}$ induces flow values on edges in G^\times that observe flow conservation at crossings.*

Proof. Assume a crossing $v = (e, e')$ and sub-edges e_1 and e_2 of e incident to v that have different flow values $f(e_1) \neq f(e_2)$. This is only possible if e_1 lies on the border ∂h of a hyperface h and $e_2 \notin \partial h$. However, this is impossible since e_1 and e_2 are part of the same edge in G and thus by **H1** are either both contained or not contained in any hyperface border. \square

Thus a hyperface labelling not only induces flow values on edges in G^\times but also in G .

Three questions arise:

1. Is there always a hyperface labelling that induces a maximum s - t -flow?
2. Is there always a set of hyperfaces that obeys **H1**, **H2** and **H3**?
3. How can we find the set of hyperfaces and a labelling efficiently?

7.6.1 Existence of a Solution

To answer the first question we first observe that it is sufficient to consider single cycles of flow. That is because we can complete the maximum s - t -flow to a circulation by adding the arc (t, s) . Then we can decompose this circulation into flow cycles.

Lemma 7.6.3. *Given a cycle C in G that does not cross itself there is a hyperface labelling such that each face on the right side of C has label 1 and each face on C 's left side has label 0.*

Proof. First we show that a hyperface that is incident to an edge in C on C 's right side lies completely on the right side of C . Otherwise a subpath of C would run between two vertices on the hyperface's border. Which is impossible by Lemma 7.6.1.

We now work incrementally. We choose a hyperface h incident on the right side of C to an edge on C . Then we set the label of h to 1. We now subtract h from the region on C 's right side and get a smaller region bounded by a cycle C' . Since the border of h lies in G we know that C' lies also in G . Therefore we can continue this procedure until all faces on C 's right side have label 1. \square

The *winding number* of C around v^* is the number of times the curve C passes (counterclockwise) around the face v^* . Using this definition we can write Lemma 7.6.3 thus: If C does not cross itself then there is a hyperface labelling that induces the winding number with respect to C on any face in G^* .

We think that this is true even for self-crossing cycles. However, we have not found a proof for this.

Proposition 7.6.4. *Given a cycle C there is a hyperface labelling such that each face in G^\times is assigned its winding number with respect to C .*

If this was true then so would be the following Corollary. This Corollary in turn implies that a maximum flow can always be represented by a hyperface labelling. That is, an optimal hyperface labelling yields an optimal solution to our maximum flow problem.

Corollary 7.6.5. *Given a flow function f on (G, c) and a set of hyperfaces H that obeys **H1** through **H3** there exists a labelling ϕ of H that induces f .*

Proof. The flow decomposition of f yields a set of cycles each with a weight: $\{(C_i, w_i)_i\}$. Using Proposition 7.6.4 for each cycle C_i we can find a hyperface labelling ϕ_i that represents C_i . The linear combination of these labellings then gives the wanted hyperface labelling

$$\phi = w_1\phi_1 + w_2\phi_2 + w_3\phi_3 + \dots .$$

\square

7.6.2 Finding Hyperfaces

We now show how to find all hyperfaces in G . For each arc e in G we find the leftmost and rightmost cycle without self-intersections (see Listing 8). Each of these clearly is a hyperface: It only contains edges in G and does not cross itself and since it is leftmost or rightmost it is also minimal.

Listing 8 Finding hyperfaces.

```

1: for each arc  $e$  in  $G$  that has no incident hyperface on its left side do
2:   Find a leftmost non-self-intersecting cycle  $C$  beginning at  $e$ .
3:   Remove a possible leading path from  $C$ .
4:   Make the left side of  $C$  a new hyperface.
5: end for
6: for each arc  $e$  in  $G$  that has no incident hyperface on its right side do
7:   Find a rightmost non-self-intersecting cycle  $C$  beginning at  $e$ .
8:   Remove a possible leading path from  $C$ .
9:   Make the right side of  $C$  a new hyperface.
10: end for

```

However, the arc we begin with may be contained in the leading path we cut off in steps 3 or 6. So maybe **H2** is not fulfilled. See Figure 7.6.5 for an example: If we start the non-self-crossing left-first search at e then in either direction we find only C_1 or C_2 but never a cycle containing e .

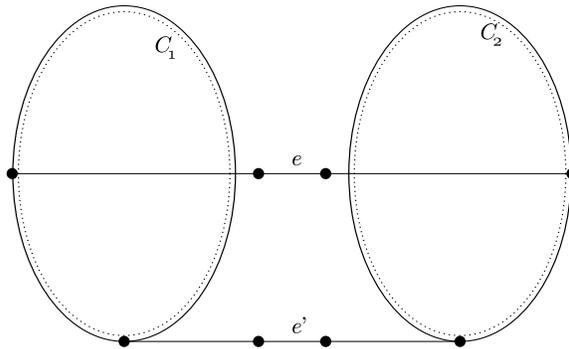


Figure 7.6.5.

Therefore we backtrack in the left-first search or right-first search if we are about to close a path that has incident hyperfaces on all edges on its left or right side, respectively. Listing 9 shows the detailed procedure that replaces step 2 in Listing 8 for left-first search. For right-first search in step 7 of Listing 8 the procedure is analogous so we do not show it here.

The procedure in Listing 9 will by design always find a leftmost cycle which does not cross itself. We apply it repeatedly while there are still arcs which are not incident to a hyperface on their left side. Then we will eventually find for any edge an incident cycle/hyperface. Therefore conditions **H1** and **H3** are fulfilled for a set of hyperfaces found in this manner. The following theorem shows that **H2** is fulfilled, as well.

Theorem 7.6.6. *Consider a cycle C found by the procedure in Listing 9 and an edge e on C which is not incident to a hyperface on its left side.*

Listing 9 Left-first search to find a hyperface.

Input: (v, w) is the start edge for the search.

Output: We build a cycle C which is initially empty.

```

1: repeat
2:   Set  $(u, v) := (v, w)$ .
3:   if  $(v, u)$  is the last arc in  $C$  then
4:      $C := C - (v, u)$ . // backtrack
5:     Let  $(u, v)$  be the arc which is now the last in  $C$ .
6:     Let  $(v, w)$  be the clockwise next arc after  $(u, v)$  at  $v$ .
7:   else if  $(u, v)$  crosses an arc contained in  $C$  then
8:     Set  $(v, w) := (v, u)$ . // prepare backtrack
9:   else
10:    if  $v$  is not contained in  $C$  then
11:      Let  $(v, w)$  be the clockwise next arc after  $(u, v)$  at  $v$ .
12:    else
13:      Set  $(v, w) := (v, u)$ . // prepare backtrack
14:    end if
15:    Set  $C := C + (u, v)$ . // extend cycle
16:  end if
17: until  $C$  contains an edge which is not incident to a hyperface on  $C$ 's
    left side.
18: Remove from  $C$  the prefix which begins at  $v$ .

```

Let h be the hyperface defined by the region on C 's left side. Then h does not contain a non-self-crossing path in G that connects two vertices on the border of h .

Proof. Assume for a contradiction that h contains such a path P . Then this path divides h in two regions h_1 and h_2 bounded by non-self-crossing cycles C_1 and C_2 , respectively, in G . Since P lies in the interior of h left-first search (or right-first search) finds C only if the borders of C_1 and C_2 are already covered with hyperfaces on their left sides. But C_1 and C_2 completely cover the edges in C . This is a contradiction since we assumed that $e \in C$ is not incident to a hyperface on its left side. \square

We conclude this section with a short summary. We have seen can always find a set of hyperfaces that obeys **H1**, **H2** and **H3**. However, the proof that there is always a labelling of this set that induces a maximum flow misses a crucial part. Therefore it remains unclear whether hyperface labelling is a suitable method to solve non-planar maximum flow instances.

Chapter 8

Conclusion

In this chapter we want to give an overview over the results of the thesis. Recall that we wanted to find an algorithm for the maximum s - t -flow problem that runs asymptotically faster on nearly planar instances than the best currently known algorithms and to identify algorithms that have a good empirical running time on these instances.

We have accomplished both. In Chapter 6 we have presented the first maxflow result for nearly planar graphs. Namely an algorithm that runs in time $O(k^3 n \log n)$ and is thus asymptotically faster than previous algorithms on all instances with $k \in O(\sqrt[3]{n})$. Furthermore this motivates a formal definition of nearly planar graphs.

On the empirical side we have found in Section 3.4.1 a simple modification of the Uppermost Path Algorithm that exhibits a very good running time behaviour and is competitive with the best currently known algorithms for our test instances.

Apart from these major results we have made a number of smaller advances. We introduced interesting new concept as for example the *hyperfaces* in Section 7.6. In Section 4 we give several new results in the theory of augmenting path algorithms. We also found many special instances which have served us as counter examples for some seemingly promising approaches. These concepts, proofs and counter examples can be very helpful in future research.

8.1 Results in Detail

In Chapter 3 we investigated the Uppermost Path Algorithm. We first introduced the concept of augmenting path algorithms and formulated the

most general augmenting path algorithm using template procedures. Then we briefly considered how augmenting path algorithms can make use of the dynamic trees data structure and gave a corresponding modification of the template procedures. Next we explored three modifications of the Uppermost Path Algorithm which seemed promising for nearly-planar instances.

Our experiments have shown that for our test instances two of these approaches are slower than our benchmark algorithms while one approach is as good as these benchmark algorithms or even better. This approach is very simple. It is just a repetition of the Uppermost Path Algorithm until maximality is reached.

We dedicated Chapter 4 to the theory of augmenting path algorithms. We began with the most generic augmenting path algorithm, already presented in Chapter 3, and refined it by gradually eliminating degrees of freedom. During the course of this refinement we found the concepts of *direction changes* and *regressions* which are measures of complexity of an augmenting path algorithm's execution. We tried to link these numbers to parameters of the instance in order to derive bounds for the running time. We proved many intermediate results, however, we were not able to prove any such connection completely. We also gave several classes of examples that disproved seemingly plausible statements.

We further proved that many of the nice properties of left-first search algorithms are already present in a new type of algorithm which we call *stubborn* algorithms. In fact, left-first search algorithms are a special case of stubborn algorithms. We then formulated a generalization of the Uppermost Path Algorithm which can solve general instances. This algorithm is given in full in Appendix A. Finally we computationally investigated the parameters inherent in this algorithm.

In Chapter 5 we considered several modifications of Dinic' blocking flow algorithm. We found that most of these modifications are just re-formulations of algorithms we had already considered in Chapter 3.

Karzanov's algorithm is a variant of Dinic' algorithm. We considered instances that can be partitioned into a planar part and k crossing edges. And we found a simple modification of Karzanov's algorithm that solves these instances in time $O(k^3 n \log n)$. However, we considered this modification only briefly in Chapter 5.

In Chapter 6 we gave the complete formulation of this algorithm. We began by generalizing the original preflow push algorithm. This generalized algorithm may be applicable to a number of problems. We gave concrete applications for nearly-planar instances and for nearly series-parallel instances.

The application to nearly-planar instances runs in time $O(k^3 n \log n)$ where k is the same as above. This motivated a definition of *nearly-planar*: We now say a graph is nearly planar if the graph genus $g \in o(n)$.

In Chapter 7 we investigated several new maximum flow algorithms working on planar dual graphs. One of these algorithms uses the new concept of *hyperfaces*. These are regions in the planar dual graph generally consisting of more than one face. Using these hyperfaces we modelled the dual of the maximum flow problem even for non-planar instances.

8.2 Outlook

In the theory of left-first search algorithms we conjectured a link between a crossing number, the genus or another measure of non-planarity on one side and a measure of algorithm complexity such as number of regressions or direction changes on the other side. Whether such a link exists remains an open question.

The generalized preflow push algorithm presented in Chapter 6 might allow more applications than the maximum flow problem in the nearly-planar and nearly series parallel cases. It would be interesting to look for such applications.

The theory of hyperfaces we presented in Chapter 7 is incomplete. It remains unknown whether a efficient algorithm based on this concept exists. On the other hand this concept might be useful in other contexts where planar dual graphs are considered.

Appendix A

The Implementation of the Left–first Search Algorithm

We build upon an existing algorithm implementation A as that provides all template procedures. This can be for example the dynamic tree implementation given in Chapter 3. If we want to call a procedure of A , for example *repair*, here we write $A.\textit{repair}$. That way we can focus here on the left–first search specific functionality.

Recall from Section 4.8 that we work on a forest F . The augmenting path we build is just one path in the forest. It may be part of a larger tree and there may be other trees in the forest. We modify and query the tree with procedures as defined for dynamic trees in Section 3.2. However, this is just an *interface* definition for us. We do not require that F really is a dynamic trees data structure.

In order to support the parameter `FinalLabels` we must overwrite the procedure `cut` as shown in Listing 10.

Listing 10 The procedure `cut`.

```
procedure cut( $v$ )  
  Let  $w := \text{parent}(v)$ .  
   $F.\text{cut}(v)$ .  
  if  $c_i(w, v) > 0$  and  $\text{ilabel}(w, v) = \infty$  then  $\text{ilabel}(w, v) = 0$   
end procedure
```

This implementation uses iteration labels (see Section 4.3). The current iteration is denoted by i .

Listing 11 The procedures `active` and `advance_arc`.

```

procedure active( $v$ )
  if current_arc( $s$ ) is not defined then
    Let current_arc( $s$ ) be the uppermost arc at  $s$ .
  end if
  if current_arc( $v$ ) has completed a rotation about  $v$  then
    if  $v \neq s$  then
      Start a new rotation about  $v$ .
    else
      return false.
    end if
  end if
  return true.
end procedure

procedure advance_arc( $v$ )
  Point current_arc( $v$ ) to the clockwise next arc in the incidence list
  of  $v$ .
end procedure

```

Listing 12 The procedure `admissible`.

```

procedure admissible( $(v, w)$ )
  if  $(v, w) \neq \text{edge\_before}(v)$  then
    if  $\text{ilabel}(v, w) \leq i$  then
      Set  $\text{ilabel}(v, w) := i + 1$ .
      if  $c_i(v, w) > 0$  then
        if  $v$  is not a descendant of  $w$  or if  $\text{parent}(w) = v$  then
          return true.
        end if
      end if
    else if FinalLabels then
      Set  $\text{ilabel}(v, w) := \infty$ .
    end if
  end if
  return false.
end procedure

```

Listing 13 The procedures `advance` and `retreat`.

```

procedure advance( $P, (v, w)$ )
  if DeleteIncoming then
    Delete all incoming arcs of  $w$ .
  else
    if parent( $w$ ) ==  $v$  then cut( $w$ ).
  end if
  if parent( $v$ ) is defined and not EnableFastForward then cut( $v$ ).
  A.advance( $P, (v, w)$ )
  if not EnableFastForward then
    if head( $P$ )  $\neq w$  and head( $P$ )  $\neq t$  then
      cut( $w$ )
    end if
    if StrictlyLeftmost then
      Set current_arc( $w$ ) := ( $v, w$ ).
    end if
  end if
end procedure

procedure retreat( $P$ )
  A.retreat( $P$ )
end procedure

```

Listing 14 The procedures `augment` and `repair`.

```

procedure augment( $P$ )
  A.augment( $P$ )
end procedure

procedure repair( $P$ )
  A.repair( $P$ )
end procedure

```

Bibliography

- [AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
- [BGH65] C. Berge and A. Ghouila-Houri. *Programming, Games and Transportation Networks*, page 190. Methuen, Agincourt, Ontario, 1965.
- [BK06] Glencora Borradaile and Philip Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. In *SODA 2006: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithms*, pages 524–533, 2006.
- [CE82] L.H. Cox and L.R. Ernst. Controlled rounding. *INFOR*, 20(4):423, 1982.
- [Che77] B. V. Cherkassky. An algorithm for constructing a maximal flow through a network requiring $O(n^2 \sqrt{p})$ operations. In *Mathematical Methods for Solving Economic Problems*, pages 117–126. Nauka, Moscow, 1977. (Russian).
- [CM89] J. Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM J. Comput.*, 18(6):1057–1086, 1989.
- [Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
- [DM89] U. Dergis and W. Meier. Implementing goldberg’s max-flow-algorithm – a computational investigation. *Mathematical Methods of Operations Research (ZOR)*, 33(6):383–403, 1989.

- [EFS56] P. Elias, A. Feinstein, and C.E. Shannon. Note on maximal flow through a network. *IRE Transactions on Information Theory*, IT-2:117–119, 1956.
- [EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, April 1972.
- [FF56] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
- [FF62] L. R. Ford and D. R. Fulkerson. *Flows in network*. Princeton University Press, 1962.
- [FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
- [Geo06] GeoBase Steering Committee. *National Road Network, Canada, Level 1 (NRNC1)*, 2006.
- [GN80] Zvi Galil and Amnon Naamad. An $O(EV \log V)$ algorithm for the maximal flow problem. *J. Comput. Syst. Sci.*, 21(2):203–217, 1980.
- [Gol98] Andrew V. Goldberg. Recent developments in maximum flow algorithms. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, volume 1432 of *Lecture Notes in Computer Science*, 1998.
- [GT88] Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
- [Had75] F. O. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing*, 4:221–225, 1975.
- [Has81] Refael Hassin. Maximum flow in (s, t) planar networks. *Inf. Process. Lett.*, 13(3):107, 1981.
- [HW04] J. M. Hochstein and K. Weihe. Edge-disjoint routing in plane switch graphs in linear time. *Journal of the ACM*, 51(4):636–670, July 2004.
- [HW07] J. M. Hochstein and K. Weihe. Maximum s – t –flow with k crossings in $O(k^3 n \log n)$ time. In *To appear in Proceedings of the 18th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.

- [IG98] H. Ishikawa and D. Geiger. Segmentation by grouping junctions. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 125–131, 1998.
- [IS79] Alon Itai and Yossi Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.
- [Jor69] Camille Jordan. Sur les assemblages de lignes. *J. Reine Angew. Math.*, 70:185–190, 1869.
- [Kar74] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.
- [KNK93] Samir Khuller, Joseph Naor, and Philip Klein. The lattice structure of flow in planar graphs. *SIAM Journal on Discrete Mathematics*, 6(3):477–490, August 1993.
- [KRRS94] Philip Klein, Satish Rao, Monika Rauch, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 27–37, 1994.
- [Kur30] Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, (15):271–283, 1930.
- [MKM78] V. M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7(6):277–278, oct 1978.
- [MMHW95] Rolf H. Möhring, Matthias Müller-Hannemann, and Karsten Weihe. Using network flows for surface modeling. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 350–359. Society for Industrial and Applied Mathematics, 1995.
- [MP93] M. Middendorff and F. Pfeiffer. On the complexity of the disjoint paths problem. *Combinatorica*, 13:97–107, 1993.
- [MS04] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via gaussian elimination. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2004)*, pages 248–255, 2004.
- [Sch05] A. Schrijver. On the history of combinatorial optimization (till 1960). In *Handbook of Discrete Optimization*, pages 1–68. Elsevier, Amsterdam, 2005.

- [Sey81] P. D. Seymour. On odd cuts and plane multicommodity flows. *Proceedings of the London Mathematical Society*, 42:178–192, 1981.
- [ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and Systems Sciences*, 26(3):362–391, jun 1983.
- [Tar84] Robert Endre Tarjan. A simple version of Karzanov’s blocking flow algorithm. *Operations Research Letters*, 2(6):265–268, mar 1984.
- [Tig05] U.S. Census Bureau. *2005 Second Edition TIGER/Line Files*, 2005.
- [Wei97] Karsten Weihe. Maximum (s, t) -flows in planar networks in $\mathcal{O}(|V| \log |V|)$ time. *Journal of Computer and System Sciences*, 55(3):454–475, December 1997.
- [WW95] D. Wagner and K. Weihe. A linear-time algorithm for edge-disjoint paths in planar graphs. *Combinatorica*, 15:135–150, 1995.
- [Zwi95] Uri Zwick. The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148(1):165–170, August 1995.