Weihe's algorithm for maximum flow in planar graphs
(project report)

Broňa Brejová and Tomáš Vinař

University of Waterloo, Course CS760K
Fall 1999

Contents

Chapter 1

Introduction

Network is a graph $G = (V, E)$ with two special vertices *source* s and *sink* t and weight function $c : E \to \mathbb{R}^+$ (*capacities* of edges). A *flow* in the network is a function $f : E \to \mathbb{R}$ satisfying two following conditions.

Capacity constraint: for all edges $e \in E$ holds $0 \leq f(e) \leq c(e)$,

Balance constraint: for all vertices $v \in V - \{s, t\}$ flow through incoming edges is equal to flow through outgoing edges in v.

Value of a flow is the difference between flow through incoming edges and flow through outgoing edges in s. The *maximum flow problem* is a problem to find a flow with maximum value.

The maximum flow problem has been one of the most well-studied problems in algorithmic graph theory over the last 50 years. It has many important applications in many different areas. Also many modifications of the problem were studied. In this project we are interested in a special case of planar graphs.

Hu and Young in 1969 [Hu69] gave the algorithm working in $O(n \log n)$ for a special case of planar graphs where source and sink are on the same face ((s, t)-planar graph). In 1985 Hassin and Johnson gave algorithm for general undirected planar graphs in $O(n \log^2 n)$ time. It is possible to improve these results using linear-time planar shortest path algorithm [KRRS94] obtaining $O(n)$ time in the case of (s, t)-planar graphs and $O(n \log n)$ for undirected planar graphs. The first $O(n \log n)$ algorithm for general planar (directed) case was developed by Weihe in 1997 [Wei97]. This algorithm is of our particular interest.

The main goal of this project was to implement Weihe's algorithm. During the work on the project we have found several errors in the article and we tried to correct them in our implementation. However after we have finished the implementation we have discovered additional errors in implemented algorithms. Therefore we decided to describe **the correct algorithms** separately from **algorithms that we have implemented**.

The rest of this report is organized as follows:

Algorithms. In this part we describe algorithms used and developed during work on the project in detail. Also we give proofs of correctness (except the proof of the core of Weihe's algorithm which can be found completely in [Wei97])

Implementation. Here we describe details of implementation of Weihe's algorithm using LEDA library.

History. We include a short history of our project and our experiences.

Error Summary. Here we summarize all errors we have found in the original article and corrections for them.

Achievements

We have achieved following results:

1. We have implemented Weihe's algorithm and tested our implementation extensively.
2. We have found several errors and ommitions in [Wei97] and summarized them in Appendix A.
3. We have completely described algorithm for removing clockwise cycles in planar network. As far as we know it is the first such complete description.
4. We have developed new algorithm for removing useless edges in general graphs working in $O(m \log n)$ time. As far as we know this is the first solution of this problem. The question whether there exists more efficient solution at least for planar graphs remains open.
5. We have developed relatively simple method for implementing back transformation. Back transformation was not described in the original article at all.

Chapter 2

Algorithms

In this chapter we study Weihe's algorithm [Wei97] in detail. The algorithm for finding maximum flow requires that all input graphs satisfy strong assumptions. Fortunatelly it is possible to transform arbitrary planar graph into planar graph meeting these assumptions. We divide the whole algorithm into three parts:

Preprocessing. Start with simple planar graph G with fixed planar embedding[1] such that t is on the outerface. Transform graph G to graph G' such that maximum flow in G has the same value as maximum flow in G' and following assumptions hold.

The first assumption prohibits clockwise cycles in a graph. Note, that directed cycle is called clockwise if its right-hand region does not contain the outerface. Therefore the sense of cycle orientation is strictly related to the choice of planar embedding.

Assumption 2.1 *Graph G contains no clockwise directed cycles.*

The second assumption prohibits so called useless edges. Notice that along these edges we cannot push any flow.

Definition 2.1 *Edge $e = (v, w)$ is called* useless *if at least one of the following conditions is NOT satisfied:*

- *e belongs to some simple directed path from s to w*
- *e belongs to some simple directed path from v to t*

Assumption 2.2 *Graph G does not contain any useless edges.*

Finally the third assumption put constraint on degree of vertices.

Assumption 2.3 *The in-degree of s is zero, its out-degree is one. The out-degree of t is zero and for any vertex $v \in V - \{s, t\}$ the degree is exactly 3.*

Augmenting. Compute maximum flow f' in graph G'. In this part we will describe the core of Weihe's algorithm including several minor corrections.

Back transformation. We have computed maximum flow f' in G' and using this we compute the maximum flow f in original graph G.

[1] If planar embedding is not provided we can compute it in $O(n)$ time. For details see [MN00].

2.1 Preprocessing and back transformation

In this section we study algorithms for preprocessing and we also briefly mention back transformation. Notice that assumptions 2.1, 2.2 and 2.3 put strong restrictions on the resulting graph. Preprocessing is divided into several steps which are performed in the following order:

1. **Removing clockwise cycles** (in order to satisfy assumption 2.1).
2. **Removing useless edges** (in order to satisfy assumption 2.2).
3. **Satisfying degree assumptions** (for assumption 2.3).

For each step we must ensure that the following holds:

1. transformation does not change the maximum flow in the graph,
2. transformation does not clear the effect of the previous transformations.

Wherever it is not obvious we try to prove these two statements. Also it is important that planarity is maintained in all steps and t remains on the outerface. Notice, that after preprocessing the graph does not contain any parallel or anti-parallel edges, however this is not necessarily true during all stages of the preprocessing.

This section contains three subsections, each devoted to one of the steps of preprocessing, and in the last subsection we discuss back transformation which is closely related to preprocessing.

2.1.1 Removing clockwise cycles

Assume that we have directed plane graph G with non-negative capacities. We want to construct directed plane graph G' that contains no clockwise cycles and that has the same value of maximum flow as graph G. This goal is to be achieved by finding a specific flow in the graph and "removing it from of the graph".

First, we introduce necessary definitions. *Circulation* is a special kind of flow, in which the balance constraint holds even for sink and source. It means that the value of flow is 0 (flow only "circulates" in network, it does not flow in and out).

Definition 2.2 *Let G be a directed graph with non-negative capacities.* Circulation *is a flow in G with zero value.*

Following two terms will help us to describe the process of removing clockwise cycles more precisely. The later of the two, residual graph of a circulation, is especially important, because it defines a notion of "removing a circulation from a graph".

Definition 2.3 *Circulation f* saturates *cycle C in a directed graph iff there is an edge $e \in C$ that is saturated by f ($f(e) = c(e)$, where $c(e)$ is capacity of e).*

Definition 2.4 *Let $G = (V, E)$ be a directed graph with non-negative capacities and f be a circulation in G.* Residual graph *of f is a graph $G' = (V, E')$ with capacities c' such that E' contains for each edge $e = (v, w) \in E$ up to two edges added according these rules:*

- *If $c(e) > f(e)$ then $e \in E'$ with capacity $c'(e) = c(e) - f(e)$.*
- *If $f(e) > 0$ then $(w, v) \in E'$ with capacity $c'(w, v) = f(e)$.*

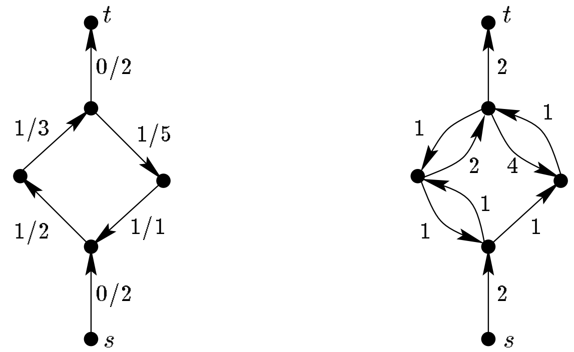


Figure 2.1: Residual graph of a circulation. On the left is an original graph with labels of the form $f(e)/c(e)$, on the right is the residual graph of circulation f.

In other words, residual graph can be obtained from the original graph so that each edge is split into two edges in opposite directions, where the edge in original direction has capacity equal to its reserve in original graph and the edge in opposite direction has the capacity equal to the flow in original edge. Afterwards all edges with 0 capacities are removed.

For example on figure 2.1 you can see a graph with circulation f that satures the only cycle in this graph. In the right part you see the residual graph of this circulation.

It is quite obvious that if G was a planar graph then residual graph of a circulation is planar as well. In the algorithm we choose the circulation so that the residual graph has no clockwise cycles.

Now we are ready to state the algorithm more precisely. It works in two steps.

1. Find circulation f that saturates all clockwise cycles.
2. Compute residual graph G' of circulation f.

The second step is easy – we can compute a residual graph of a circulation by simply following the definition. However, notice that directed cycle orientation is related to the position of the outerface. Therefore we work with fixed planar embedding of G and we also need to provide an embedding for G'.

We start with embedding for G. When we have an edge e such that both e and its reverse e' are in E' then we place them in an embedding "one next to the other" (more formally, there will be a face that is adjacent to both e and e' and to no other edge, and this face is not an outer-face). Moreover e and e' are ordered so that cycle formed by them is counter-clockwise. In case that only one of e and e' is in E', it simply replaces the original edge e. The residual graph in figure 2.1 is an example of such embedding of G'. Further we always assume that G' has this fixed embedding.

It is possible to compute residual graph G' and its embedding in $O(n)$ time.

The rest of this section is divided into three parts. The first of them shows an algorithm for constructing circulation f in $O(n \log n)$ time, the second one contains a proof that G' contains no clockwise cycles and the last part shows that G' has the same maximum flow as G and describes how to perform a back transformation (from flow in G' to flow in G).

Construction of the circulation f

The algorithm described here resembles very much the algorithm for computing the maximum flow in (s,t)-planar graphs by Hassin[2] [Has81]. It works in three steps. In the first step we compute a modified dual

[2]Let us note for a reader that knows details of the algorithm by Hassin that the algorithm described here differs in only two points. First, we need to modify slightly the dual graph so that it can handle oriented edges. Second, we do not add extra edge

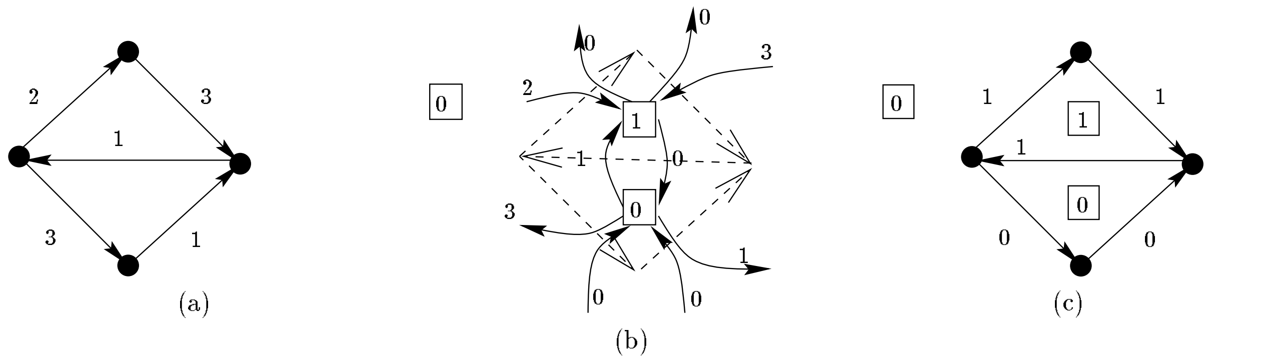


Figure 2.2: Computation of circulation f. (a) original graph G, (b) modified dual G^* with computed distances, (c) resulting circulation f in G.

graph G^* of G. In the second step we compute distances in this dual graph. In the third step we assign flow values to edges based on distances in a dual graph. The process is illustrated in the figure 2.2.

The first step. We start by constructing the modified dual graph $G^* = (V^*, E^*)$ of graph G and assigning weights to the edges of G^*. V^* are the faces of G (as in normal dual graph). E^* can be constructed as follows. Let $e \in E$ and let F_L be a face incident to e on the left-hand side and F_R be a face incident to e on the right-hand side. We add two edges e_1^* and e_2^* to E^* such that

- $e_1^* = (F_L, F_R)$ and its weight is $w^*(e_1^*) = c(e)$
- $e_2^* = (F_R, F_L)$ and its weight is $w^*(e_2^*) = 0$

The second step. Compute the shortest paths from the outer-face of G to all other faces based on weights assigned to edges in E^*. This can be done by Dijkstra algorithm in $O(n \log n)$.

The third step. Let $d(F)$ be a distance from the outer-face to the face F in G^*. Let $e \in E$ be an edge with face F_L on the left and face F_R on the right. The flow on e will be $f(e) = d(F_R) - d(F_L)$.

Time complexity. The first and the third step can be done in $O(n)$ time (provided that the graph is stored in suitable data structures). The second step works in $O(n \log n)$ using Dijkstra algorithm where estimates of the distances to nodes are stored in binary heap or Fibonacci heap.

Correctness. We will prove here that f is indeed a circulation. Proof that f saturates all clockwise cycles will appear later (see Theorem 2.5).

Claim 2.4 *The algorithm described above computes a circulation in G.*

Proof. We need to show that f fulfills both capacity constrains for each edge and balance constrains for each node.

Let e be an edge in G with face F_L on the left and face F_R on the right. We need to show that $0 \le f(e) \le c(e)$. Recall that $f(e) = d(F_R) - d(F_L)$ where $d(F)$ is the distance from the outer-face to face F in G^*. Graph G^* was constructed so that it contains edge (F_L, F_R) with weight $c(e)$ and edge (F_R, F_L)

connecting s and t in order to obtain a circulation instead of a flow.

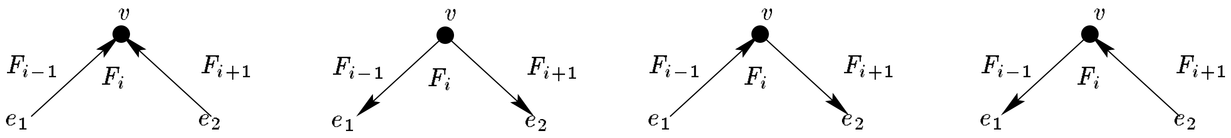


Figure 2.3: Four cases from the proof of claim 2.4.

with weight 0. Using triangle inequality we get that $d(F_R) \leq d(F_L) + c(e)$ which implies that $f(e) \leq c(e)$ and similarly $d(F_L) \leq d(F_R) + 0$ which implies that $f(e) \geq 0$. This proves capacity constrains for edge v.

Let v be a node in G. We want to prove that the amount of flow entering v is the same as the amount of flow leaving v. Let $x = \sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e)$, we need to show that $x = 0$. Let F_0, F_1, F_2,... F_{k-1} be faces incident to v in counter-clockwise order. Each $f(e)$ is a difference between some $d(F_i)$ and $d(F_j)$ for some faces F_i and F_j incident to e. Therefore x can be expressed as a sum in which each $d(F_i)$ appears twice, each time either with plus or minus sign. We will show that each $d(F_i)$ appears once with minus and once with plus which would mean that all $d(F_i)$ cancel out and x is 0. Let us take face F_i and let e_1 and e_2 are edges incident to F_i and v (in counter-clockwise order). We distinguish four cases (see Figure 2.3):

Both e_1 and e_2 are incoming edges of v. In this case $f(e_1) = d(F_i) - d(F_{i-1})$ and $f(e_2) = d(F_{i+1}) - d(F_i)$ ($+1$ and -1 are computed $\bmod k$). Both $f(e_1)$ and $f(e_2)$ have plus sign in x (both are incoming). We see that $d(F_i)$ cancels out.

Both e_1 and e_2 are outgoing edges of v. Then, $f(e_1) = d(F_{i-1}) - d(F_i)$ and $f(e_2) = d(F_i) - d(F_{i+1})$. Both $f(e_1)$ and $f(e_2)$ have minus sign in x. Again $d(F_i)$ cancels out.

Edge e_1 is incoming and e_2 is outgoing. Then, $f(e_1) = d(F_i) - d(F_{i-1})$ and $f(e_2) = d(F_i) - d(F_{i+1})$. In this case $f(e_1)$ has plus sign in x (it is incoming) whereas $f(e_2)$ has minus sign (it is outgoing). Clearly $d(F_i)$ cancels out.

Edge e_1 is outgoing and e_2 is incoming. Then, $f(e_1) = d(F_{i-1}) - d(F_i)$, $f(e_2) = d(F_{i+1}) - d(F_i)$, $f(e_1)$ has minus sign in x and $f(e_2)$ has plus sign in x. Also in this case $d(F_i)$ cancels out.

This finishes the proof that f is a circulation in G. $\square$

No-clockwise-cycles proof

In this part we will show that residual graph G' of circulation f contains no clockwise cycles. Our aim was to construct f so that it saturates all clockwise cycles in G. We will not explicitly prove this here, however it is implied by the fact that no clockwise cycles remain in G'.

Theorem 2.5 *Let f be the circulation in G constructed by the given algorithm and G' be its residual graph. Then G' does not contain any clockwise cycles.*

Proof. We prove theorem by contradiction. Assume that there is a clockwise cycle C in G'. For each edge $e \in C$ let e_G be an edge in G from which e was created in construction of G'. Cycle C does not contain two reversal edges e and e' that were created from one edge in G because we have added these edges so that they form a counter-clockwise cycle. Therefore edges e_G for $e \in C$ are all distinct and they form a cycle in G (however this cycle is not necessarily directed, some edges may go in clockwise direction, others in counter-clockwise). Denote this cycle C_G.

Now recall that in construction of f we have built a modified dual graph G^* and we have computed shortest paths from the outer-face to all other faces. Let us now consider tree in G^* defined by the shortest

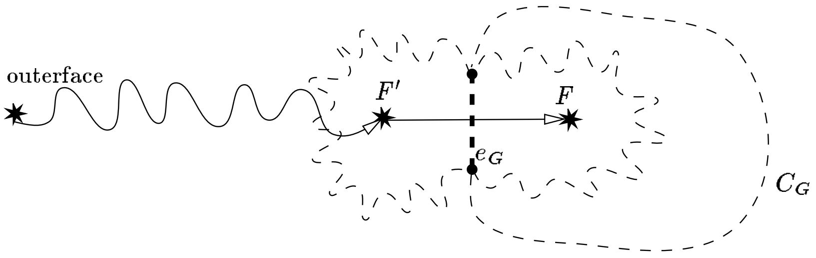


Figure 2.4: Illustration to the proof of theorem 2.5. Graph G is dashed and G^* is in solid line. Figure shows cycle C_G, faces F and F' and part of the tree.

paths (i.e. outer-face is a root and each other vertex v has as a parent its predecessor on the shortest path computed by the algorithm from the outer-face to v).

The outer-face is outside the cycle C_G and there is at least one face inside C_G. Therefore the tree must enter interior of C_G somewhere and there must be a face F inside C_G that has its parent F' in the tree outside C_G. Note, that $d(F) = d(F') + w^*(F', F)$ because F' is a parent of F in the shortest paths tree. Let e_G be an edge in G that separates F from its parent F' (see Figure 2.4). There are two cases according to the orientation of e_G.

Edge e_G is oriented clockwise in cycle C_G. Then F' is on the left and F is on the right of e_G. Therefore $w^*(F', F) = c(e_G)$ and $d(F) = d(F') + c(e_G)$. Flow on e_G is $f(e_G) = d(F) - d(F') = c(e_G)$. We see that e_G will be in G' replaced by the edge with capacity $c(e_G)$ going in the opposite (i.e. counter-clockwise) direction. Therefore there is no edge e in C such that e_G is its corresponding edge in G because C is a clockwise cycle. We have a contradiction.

Edge e_G is oriented counter-clockwise in cycle C_G. Then F' is on the right and F is on the left of e_G. Therefore $w^*(F', F) = 0$ and $d(F) = d(F') + 0$. Flow on e_G is $f(e_G) = d(F') - d(F) = 0$. We see that e_G will be in G' replaced by the edge with capacity $c(e_G)$ going in the same direction and no edge in opposite (i.e. clockwise) direction will be created. Therefore there is no edge e in C such that e_G is its counter-part in G because C is clockwise cycle. We have a contradiction.

□

Maximum flow in G'

In the previous part we have shown how to construct graph G' that has no clockwise cycles for given graph G. In order to be able to use this construction we need to prove that G' has the same value of the maximum flow as G and that having the maximum flow in G' we can easily find a maximum flow in G.

In the following lemma we establish a relation between flows in G and flows in G'. In order to make things easier we will state it for an auxiliary graph G'' instead of G'. This G'' is essentially the same as G' but it contains for each edge in G two opposite edges, even in the case that one of them has zero capacity. Thus G' and G'' differs only in edges of zero capacities and these edges cannot be used in any flow.

Lemma 2.6 *Let G be a graph, f be a circulation in G found by the algorithm for removing clockwise cycles and let G'' be a modified residual graph of f as described above. For each flow g in G there exists flow g' in G'' (resp. for each flow g' in G'' there exists flow g in G) such that for each edge $e = (v, w)$ in G and its corresponding edges $e_1 = (v, w)$ and $e_2 = (w, v)$ in G'' holds*

$$g(e) = f(e) + g'(e_1) - g'(e_2) \tag{2.1}$$

Proof. Let g be a flow in G. We construct g' in the following way. Let $e = (v, w)$ be an edge in G, $e_1 = (v, w)$ and $e_2 = (w, v)$ be edges in G''.

$$g'(e_1) = \begin{cases} g(e) - f(e) & \text{if } g(e) \geq f(e) \\ 0 & \text{if } g(e) < f(e) \end{cases} \qquad g'(e_2) = \begin{cases} f(e) - g(e) & \text{if } f(e) \geq g(e) \\ 0 & \text{if } g(e) < f(e) \end{cases}$$

It is easy to verify that g' is a flow and that $g(e) = f(e) + g'(e_1) - g'(e_2)$.

Similarly let g' be a flow in G'' and e, e_1 and e_2 be edges as before. We construct g using directly formula (2.1). We can easily check that the resulting function is a flow in G. □

The following two corollaries relate values of flow in G and in G'.

Corollary 2.7 *Let g be a flow in G and g' be a flow in G'' such that we either constructed g from g' using the lemma 2.6 or vice-versa. Then g and g' have the same value.*

Proof. It can be shown easily that value of g is value of g' plus value of f. However f is a circulation, i.e. flow of zero value and therefore value of g and value of g' are the same. □

Corollary 2.8 *Maximum flow in G has the same value as the maximum flow in G'.*

Proof. Let g be the maximum flow in G. Then there is a flow g' in G'' as described in lemma 2.6 and according to the corollary 2.7 this flow has the same value as g. Similarly for a maximum flow in G'' we can construct a flow in G with the same value. Therefore maximum flow in G and in G'' have the same value. We can obtain G' from G'' by deleting edges with capacities 0 and thus G' has also the same value of the maximum flow as G. □

To conclude this section, we have shown how to construct a graph G' such that it does not contain any clockwise cycles and the maximum flow in G is the same as the maximum flow in G'. Moreover, once we have the maximum flow in G' we can construct the maximum flow in G using formula (2.1).

Bibliography

[Wei97] does not describe how to construct circulation f, it provides only reference to [KNK93]. However [KNK93] deals with much wider topic of lattice structure of flow and information related to a construction of the circulation saturating all clockwise cycles is spread across several places in the paper. Namely, the claim 2.4 is stated in [KNK93, page 480] without the proof. Proof here is based on lecture notes [Bie99b]. Theorem 2.5 is based on [KNK93, theorem 3.6], but this theorem states only that f saturates all clockwise cycles so we needed to modify the proof slightly. Lemma 2.6 and its corollaries are based on statement in [Wei97] that was left without the proof as "well known from general flow theory".

As can be seen from the previous discussion this is probably the first place that summarizes all details and necessary proofs concerning removing clockwise cycles.

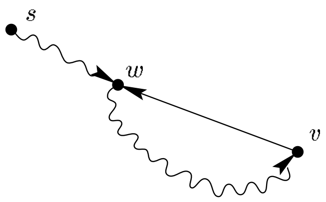


Figure 2.5: In order to find out whether edge (v, w) is useless we need to know whether there is a path from s to v avoiding w.

2.1.2 Removing useless edges

Assume that we have directed plane graph G. We want to find and delete all its useless edges (see definition 2.1). Deleting edges once they are found is not an operation worth theoretical interest. Therefore we will concentrate solely on finding such edges.

We want to find all edges that either do not belong to any simple directed path starting in s or that do not belong to any simple directed path ending in t. These two conditions are symmetrical and once we can decide for an edge whether it belongs to some simple directed path from s we can use the same algorithm for deciding whether the edge belongs to a simple path ending in t (we simply change the orientation of all edges, exchange s and t and run the same algorithm). Therefore we will discuss only how to decide for an edge whether it belongs to some directed simple path from s.

At the first sight it might seem that this task is very easy and can be accomplished for example by some simple depth-first search. Indeed, it is easy to find out whether given edge $e = (v, w)$ lies on some directed path from s – we can easily for a directed path by appending edge e to the path in the tree from root s to v. However it might be the case that this path does contains vertex w twice and it is hard to decide whether there exists a different path from s to v that avoids w (see Figure 2.5). In order to solve this task we need to employ more complicated methods.

We will present an algorithm that finds useless edges in arbitrary directed graph in time $O(m \log n)$. The algorithm uses two special data structures — interval trees and dynamic trees. Let us first briefly describe these two structures.

Interval tree Interval tree is a structure that can store a set of intervals that have end-points belonging to some fixed set X of n points. It is possible to do the following operations in $O(\log n)$ time:

- **Insert**(a,b) inserts interval (a, b) to the tree where $a, b \in X$, $a \leq b$.
- **Delete**(a,b) deletes interval (a, b) from the tree
- **Covered**(x) return true if there is interval (a, b) in the tree such that $a < x < b$.

The structure can be initialized in $O(n)$ time as an empty tree. Interval trees are often used in computational geometry and details of their implementation can be found for example in [PS85].

Dynamic trees Dynamic trees structure stores a collection of vertex-disjoint rooted trees with weighted edges.[3] The structure operates on a fixed set of vertices and can be initialized in $O(n)$ time as a set of n isolated vertices. In the algorithm we will use only the following operations (each can be implemented in $O(\log n)$ time):

[3] We will use the structure again in the algorithm for flow construction.

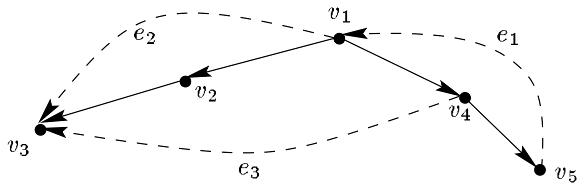


Figure 2.6: Types of edges in DFS search of a directed graph. Tree edges are in solid lines, e_1 is a back edge, e_2 is a forward egde and e_3 is a cross edge. Vertices are numbered using DFS numbers.

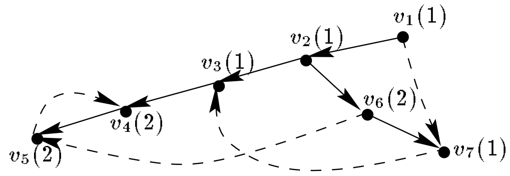


Figure 2.7: Detours in a graph. DFS tree is in solid lines, label of each vertex has form $v_i(detour(v_i))$.

- **Link**(u,v,x) inserts edge (u, v) with weight x (v will be a parent of u, u was a root before).
- **Change_value**(u,x) changes value on the edge from u to parent(u) to x.
- **Min_value**(u) returns the minimum weight on the path from u to the root of its tree.

Algorithms for dynamic trees can be found[4] in [ST83].

Overview of the algorithm.

The algorithm works in three traversals of the graph. The first traversal is simply depth-first search starting in s in which we compute DFS tree T and for each vertex v its DFS number $num(v)$. Edges that were not examined during this search surely do not belong to a directed path from s and therefore are all useless and we will not consider them in the further discussion.

Let v_i be a vertex such that $num(v_i) = i$. Recall that DFS in the directed graph divides all edges into following categories (see Figure 2.6):

- **Tree edges** are the edges included in the DFS tree T.
- **Back edges** are the edges leading from a vertex to its ancestor.
- **Forward edges** are the edges leading from a vertex to its descendant.
- **Cross edges** are the edges leading from a vertex to a vertex in a different subtree (they always lead to a vertex with lower DFS number).

In the second traversal we compute for each vertex v value $detour(v)$ defined in Definition 2.5 (see also Figure 2.7). For u and v such that u is an ancestor of v let $T(u, v)$ denotes set of all vertices on a path in T between u and v (excluding u and v).

[4]Dynamic trees described in the given reference do not contain exactly the operations described here but our operations can be easily expressed in terms of operations given in the paper.

Figure 2.8: Illustration to the proof of formula (2.2).

Definition 2.5 *Directed path P from vertex v to its descendant w will be called* detour *if it does not go through any vertex in $T(v, w)$. For vertex v define $detour(v)$ as the minimum i such that*

- *v_i is an ancestor of v, and*
- *there is a detour from v_i to v.*

In the third traversal we finally compute for each edge whether it is useless or not. The second and the third traversal are non-trivial and will be explained in more detail below.

The second traversal.

Let us assume that we already know $detour(v_j)$ for all $num(v_j) > num(v)$ and we want to compute $detour(v)$. We take all edges incoming to v one by one and for each edge e we compute the best detour $detour(e)$ ending with edge e. Value of $detour(v)$ will be the minimum of $detour(e)$ for all incoming edges. So let us examine edge $e = (w, v)$. There are several cases corresponding to the type of edge e.

- **Tree edge or forward edge.** Then $detour(e) = num(w)$. It can be seen simply from the fact that w is an ancestor of v and edge e is itself a detour.
- **Cross edge.** Let a be the nearest common ancestor of v and w. Then we have

$$detour(e) = \min_{u \in T(a,w) \cup \{w\}} detour(u) \tag{2.2}$$

  Proof of this formula is as follows (see figure 2.8). First prove that $detour(e)$ is not greater than right hand side of the formula. Let u be a vertex in $T(a, w) \cup \{w\}$ with minimum value of $detour(u)$, let u_d be the vertex for which $num(u_d) = detour(u)$ and let P be a detour from u_d to u. Note, that u_d is an ancestor of a and thus also an ancestor of v. Path P' starting with path P, then continuing along path from u to w in T and finally using edge e is a detour from u_d to v which gives us our inequality.

  On the other hand let P be a detour from x to v ending with edge e where x is an ancestor of v. We prove that $num(x) \geq \min_{u \in T(a,w) \cup \{w\}} detour(u)$. If $num(x) > num(a)$ than the inequality clearly holds. Let us assume that $num(x) \leq num(a)$. In this case x is also an ancestor of w. Path P avoid all vertices in $T(x, a)$. Let y be the first vertex on P that belongs to $T(a, w) \cup \{w\}$. The first part of P from x to y is therefore a detour and thus $detour(y) \leq num(x)$ which implies our inequality.

- **Back edge.** In this case we have

$$detour(e) = \min_{u \in T(v,w) \cup \{w\}} detour(u)$$

  Since v is the nearest common ancestor of v and w we get the same formula as in the previous case. Its correctness can be also shown using a similar argument.

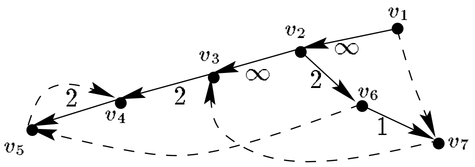


Figure 2.9: Example of the second traversal. The figure shows the values of the tree edges stored in dynamic trees structure at the time of processing node v_3. Note that Min_Value(v_7) returns 1 (as a minimum of $\infty, 2, 1$) and therefore for edge $e = (v_7, v_3)$ we have $detour(e) = 1$ and this will be also value of $detour(v_3)$.

Now we can describe the entire algorithm for the second traversal. We will use the dynamic trees structure in this part. We first initialize it to the set of isolated vertices in $O(\log n)$ time and afterwards we insert all tree edges to it, each with weight set to ∞.

Afterwards we process the vertices in descending order of DFS numbers (i.e. from v_n to v_1). After we compute $detour(v_i)$ we perform Change_Value(v_i, $detour(v_i)$), i.e. we change the value of an edge connecting v_i to its parent to $detour(v_i)$. Therefore when we compute v_i all values $detour(v_j)$ for $j > i$ are already known and all tree edges leading to such v_j have value $detour(v_j)$ associated with them in the dynamic trees (see Figure 2.9). We will determine value $detour(e)$ for each edge $e = (w, v_i)$ incoming to v_i in the way described above. In the case that e is tree edge or forward edge, we simply use value $detour(w)$ computed before. In the case that e is a cross edge or backward edge we need to compute $detour(e)$ as

$$detour(e) = \min_{u \in T(a,w) \cup \{w\}} detour(u)$$

However this can be simply computed as Min_Value(w). Note that from the properties of DFS $num(a) < num(v)$ and $num(u) > num(v)$ for all $u \in T(a, w) \cup \{w\}$. Therefore if we consider the path in T leading from w to s all edges in it have value ∞ except edges leading from vertices in $T(a, w) \cup \{w\}$. The later edges have the value already set to $detour(u)$. Therefore $\min_{u \in T(a,w) \cup \{w\}} detour(u)$ is the same as the minimum along the path from w to s.

All together we need $O(n)$ time for initialization of dynamic trees structure, $O(n \log n)$ time for initial inserting vertices and for each edge we need time $O(\log n)$ for calling operation Min_Value. In addition for each vertex we call operation Change_Value again in $O(\log n)$ time. Together we have $O((n + m) \log n)$ time which gives us $O(m \log n)$ time for connected graphs.

The third traversal.

Now we have to show how to determine for each edge whether it is useless or not. Obviously, if we have an edge (v, w) that is either tree edge, forward edge or cross edge, then the path from s to v in the DFS tree T followed by e is simple and directed and therefore e is not useless. We can need to consider only back edges. First we will prove the following lemma

Lemma 2.9 *Let $e = (v, w)$ is a back edge. Then e is not useless if and only if for some $u \in T(w, v) \cup \{v\}$ holds $detour(u) < num(w)$.*

Proof. First, recall that e is not useless if and only if there is a directed path from s to v not going through w (this path does not need to be necessarily simple because even if it is not we may easily make it simple by removing some edges).

Let there is a vertex $u \in T(w, v) \cup \{v\}$ for which holds $detour(u) < w$. We will prove that e is not useless. Let u_d be the vertex for which $num(u_d) = detour(u)$ and let P be a detour from u_d to u. Path P avoids w.

Main Program

1. compute DFS tree T and DFS numbers
2. initialize dynamic trees structure DT with set of vertices $\{1, 2, \ldots, n\}$
   for each $e = (v, w) \in T$ do
   DT.Link(w, v, ∞)
   for $v_n, v_{n-1}, \ldots, v_1$ do
   Compute_detour(v_i)
   DT.Change_value$(v_i, detour(v_i))$
3. initialize interval tree IT with set of endpoints $\{1, 2, \ldots, n\}$
   Visit(s)

procedure Compute_detour(v)
$detour(v) \leftarrow \infty$
for each edge $e = (w, v)$ do
if e is a tree edge or a forward edge then
$detour(e) \leftarrow num(w)$
if e is a back edge or a cross edge then
$detour(e) \leftarrow$ DT.Min_value(w)
$detour(v) \leftarrow \min(detour(v), detour(e))$

procedure Visit(v)
IT.Insert$(detour(v), num(v))$
for each edge $e = (v, w)$ do
if e is a tree edge then
visit(w)
if e is a back edge then
if IT.Covered$(num(w))$ = false then
mark e as useless
IT.Delete$(detour(v), num(v))$

Figure 2.10: Pseudocode for algorithm $\mathcal{G}$

The path starting by the path in T from s to u_d, then continuing with P and ending with the path in T from u to v is a directed path from s to v that avoids w. Therefore e is not useless.

Now let us suppose that e is not useless. Then there is a directed path from s to v not containing w. Let u be the first vertex on P that belongs to $T(w, v) \cup \{v\}$ and let u_d be the last vertex on P that is before u_d on P and belongs to $T(s, w) \cup \{s\}$. Then the part of P from u_d to u is a detour and therefore $detour(u) \leq num(u_d) < num(w)$. □

In our algorithm for the third traversal we will use interval trees with set of endpoints equal to $\{1, 2, \ldots, n\}$. We will perform a depth first search on a graph. When we are in vertex v the interval tree contains intervals of the form $(detour(u), num(u))$ for all ancestors of u of v (including v itself). Therefore when we enter node v for the first time we insert interval $(detour(v), num(v))$ into the tree and when we are finished with searching v and its children we delete $(detour(v), num(v))$ from the interval tree.

For each back edge (v, w) leading from v we need to check whether there is vertex $u \in T(w, v) \cup \{v\}$ for which $detour(u) < num(w)$. But this happens if and only if there is a vertex u that is an ancestor of v such that $detour(u) < num(w) < num(u)$. But for each ancestor u of v we have interval $(detour(u), num(u))$ stored in interval tree. Therefore all we need to do for a back edge (v, w) is to call Covers$(num(w))$ and if it is false then label w as useless.

The time complexity of the third traversal is $O(m \log n)$ because we perform two operations of interval trees for each node and one such operation for each back-edge.

Conclusion.

We have presented algorithm that finds useless edges in arbitrary graph. It works in three traversals of graph. The first traversal is simply depth-first search and works in $O(n)$ time. The second traversal computes *detour* numbers of vertices. This traversal uses dynamic trees structure and works in $O(m \log n)$ time. The last

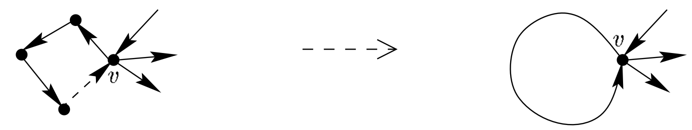


Figure 2.11: Deleting vertices of degree 2 does not create loops. Assume it does — on the right there is a resulting loop, on the left the corresponding cycle in the original graph with an useless edge in dashed line.

traversal finally marks useless edges. It works in $O(m \log n)$ and it uses interval tree structure. The overall complexity is therefore $O(m \log n)$. The pseudocode for the algorithm is given in Figure 2.10.

Bibliography

Weihe's paper on maximal flow [Wei97] does not describe how to find useless edges. It states "simply remove them". The algorithm presented here was developed by us. During the work on project also two other approaches were considered but unluckily none of them works (see section 3.1.1 and section A page 41).

2.1.3 Satisfying degree assumption

Assume that we have a planar graph and we have to transform it so that it satisfies assumption 2.3. We also assume that the graph does not contain any useless edges. Moreover we need to prove that our transformation does not create new useless edges. The assumption states different conditions for source s, sink t and all other vertices. We will discuss them one by one.

For source we need to ensure that it has only one outgoing edge and no incoming edges. This can be achieved simply by adding a new vertex s' and an edge (s', s) with sufficiently large capacity (for example the capacity might be the sum of capacities of all edges leaving s). Afterwards we make s' to be a source of the network. In this way source clearly satisfies the assumption. The maximum flow does not change. We also do not add any useless edge.

Situation with sink is even simpler. We want to have a graph in which no edge leaves t. But all such edges are useless. Therefore (after useless edges removal) there are no such edges in our graph.

Now consider vertices not equal to s or t. We need to ensure that all such vertices will have degree 3. First realize, that there is no vertex of degree 1, because the only edge incident to such vertex would be useless. The rest is done in two steps. First we modify the graph so that it does not contain vertices of degree 2 and then we "blow up" all vertices so that all resulting vertices will have degree 3.

Removing vertices of degree 2

Since there are no useless edges in a graph, any vertex of degree 2 has one incoming and one outgoing edge. This path of length 2 can be replaced by a single edge with capacity equal to minimum of capacities of original edges. We can do this transformation for all vertices of degree 2. The maximum flow does not change in this way and no useless edges are added.

Note that the transformation does not change degree of any vertex that is not deleted but it can introduce some multiple edges. However we might show that we never produce any loops in this way. If we produce a loop in vertex v it means that in there was an directed cycle going through vertex v such that all other vertices on this cycle had degree 2. But the edge in this cycle entering v is useless because the only possible directed path from s going through this edge goes through v twice (see Figure 2.11).

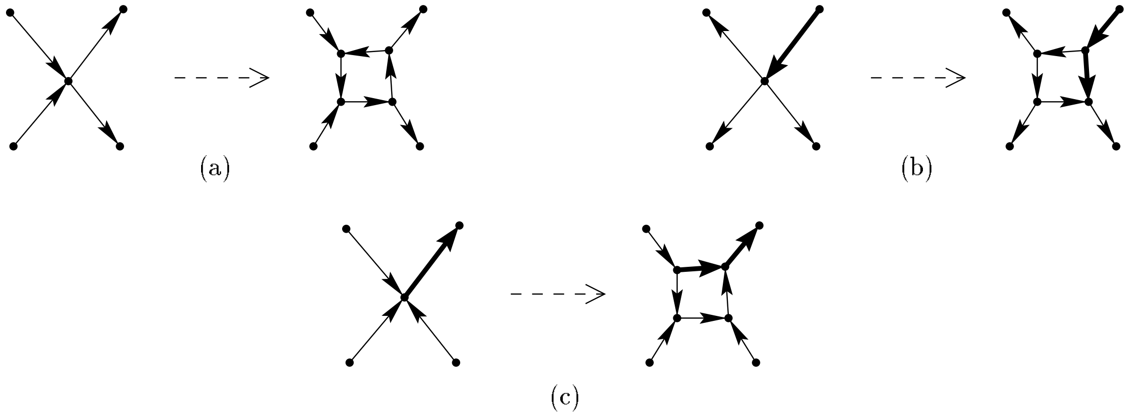


Figure 2.12: Blowing up vertices. In cases (b) and (c) we have only one incoming/outgoing edge (in bold line). One of the edges in cycle C_v is directed in clockwise direction (also in bold line).

"Blowing up" vertices

In this part we assume that all vertices (except s and t) have degree at least 3 and we will transform the graph so that all vertices have degree 3. We will replace each vertex $v \neq s, t$ of degree $k \geq 3$ by a (not always directed) cycle C_v with k vertices and each edge incident to v will now lead to a different vertex on a cycle. In order to preserve planarity we will do it this way. Let $e_1, e_2, \ldots, e_k$ are edges incident to v in counter-clockwise order and let $v_1, v_2, \ldots, v_k$ be new vertices in a cycle. Then e_i is replaced by an edge that has one endpoint in v_i.

Now we need to show how to construct cycle C_v. There are several cases:

- Vertex v is incident to at least two incoming and two outgoing edges. Then all edges in C_v are oriented in counter-clockwise direction (see Figure 2.12(a)).
- Vertex v is incident to only one incoming edge e. Let w be a vertex in C_v incident to the edge corresponding to e. Then we make all edges in C_v in counterclockwise direction except the edge incoming to w which will be in clockwise direction (see Figure 2.12(b)).
- Vertex v is incident to only one outgoing edge e. Let w be a vertex in C_v incident to the edge corresponding to e. Then all edges in C_v will be in counterclockwise direction except the edge leaving w, which will be in clockwise direction (see Figure 2.12(c)).

All edges in C_v will have sufficiently large capacity (for example equal to the sum of all edges incoming to v).

After this transformation the new graph has properties summarized in the following lemmas.

Lemma 2.10 *The value of maximum flow does not change during blowing up vertices.*

Proof. It is easy to see that for any flow in the original graph we can construct a corresponding flow with the same value in the new graph. Original edges stay in the graph and flow through vertices can be routed via edges on cycles C_v because they have sufficient capacity.

Similarly for any flow in the new graph we can construct a flow with the same value in the original graph. Flow along original edge will be the same as the flow along the corresponding edge in the new graph. Both capacity and balance constraints will be satisfied. □

Lemma 2.11 *The modified graph does not contain any useless edges.*

Proof. First let us prove that for any simple directed path in the original graph there is a corresponding simple directed path in the new graph. We take the original path and whenever it goes through some vertex v we replace vertex v by a path in cycle C_v. Note that such path always exists — for each edge entering the cycle and each edge leaving the cycle there is a directed path connecting them, even in the case that C_v contains one edge in clockwise direction.

Now let us take any edge e' corresponding to some edge e in the original graph. Edge e was not useless and therefore there was a path from s and a path to t containing this edge. There are corresponding paths to these paths in the modified graph, too and these two paths are also directed and simple. Therefore e' is not useless.

Let $e = (w, w')$ be an edge in some cycle C_v. There is an edge e' incoming to the cycle C_v such that it is not incident to w' and there is a directed path in C_v from the end of e' to w. This can be proved for any form of the cycle C_v – whether it contains an edge in clockwise direction or not.

We know that e' is not useless and that there is a simple directed path from s ending with e'. This path does not contain any edge in C_v and therefore we can add a part of C_v that forms a path connecting e' to w and we get a simple directed path from s containing (w, w'). Similarly we can also find a path to t that is simple and directed and contains (w, w'). Therefore e is not useless. □

Lemma 2.12 *Let v and w be two vertices in the modified graph such that at least one of them belongs to $V - \{s, t\}$. Then (v, w) is not a multiple edge.*

Proof. Clearly, blowing vertices does not add new multiple edges. If there was some multiple edge in the original graph such that it has at least one endpoint in $V - \{s, t\}$, then this endpoint was "blown up" and individual edges are now incident to different vertices in the corresponding cycle. Therefore the multiple edge has disappeared. □

Lemma 2.13 *If the original graph does not contain any clockwise cycles, neither the modified graph does.*

Proof. No cycle C_v is clockwise (it can contain at most one edge in clockwise direction). Let there is a different clockwise cycle C in the modified graph. Then it contains some edges from the original graph. If we contract all cycles C_v in the modified graph we get the original graph and the cycle C will be still a clockwise cycle. But it is a contradiction because the original graph does not contain any clockwise cycles. □

Bibliography

This part of preprocessing is described in detail in [Wei97]. There is a small mistake concerning removing vertices of degree 2 and an error in "blowing up vertices" (always replaces vertex by counter-clockwise cycle which can produce useless edges). We have discovered and solved these two problems.

2.1.4 Back transformation

Recall that during preprocessing we have transformed input graph G into graph G' satisfying assumptions 2.1–2.3. After we compute the maximum flow in G' we need to transform it back to G. During preprocessing we have performed various operations including adding new edges, deleting edges, adding vertices, removing vertices and so on. For each one elementary operation done during preprocessing it is quite obvious how we can compute flow in the original graph from the flow in the modified graph. However it is not that obvious how to keep track of all these operations in order to be able to do the back transformation.

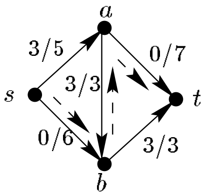


Figure 2.13: Example of the augmentation. Labels are labeled with current flow and capacity. Dashed path is augmenting path with reserve 3. Edge (a, b) is used on the path in the backward direction. After augmenting along this path the flow will have value 6.

We propose a simple solution of the problem as follows. Most operation have a property that for each edge e in the original graph there is an edge e' in the modified graph such that flow on e has to be the same as the flow on e'. Even if we have a sequence of the operations with these properties we can compute corresponding edges simply (we can imagine it as a map from edges in original graph to edges in modified graph and doing several operation corresponds to composition of such maps).

The only operation that does not have this property is removing clockwise cycles. Lemma 2.6 shows that it is sufficient for every edge in the original graph to keep track of two edges in the modified graph and one value – flow on the edge in the circulation (here again we can represent the kept information as three maps and we can compose them with single maps resulting from other operations).

As a result we have that only two pointers and one value for each edge in G can store enough information for doing back transformation.

2.2 Augmenting

In this section we will present the core of the Weihe's algorithm, namely the part that finds the maximum flow for given simple plane graph with t on the outerface satisfying assumptions 2.1–2.3. We will describe the algorithm in detail, however we will not reproduce the proof of the correctness which is very complicated and can be found in [Wei97].

The algorithm is based on the augmenting path algorithm [FF62]. In order to explain it better we need the following definition.

Definition 2.6 *Let G be a network, f be a flow in G. Then* reserve *of edge e in G* in the forward direction *is $c(e) - f(e)$ and its reserve in* backward direction *is $f(e)$.* Reserve of a path *is minimum of reserves of edges on this path where edges can be used on the path in both forward and backward directions and reserve is taken from the appropriate direction.* Augmenting path *is a path with positive reserve.*

When we have augmenting path P from s to t with reserve δ we can increase the value of the flow by δ so that we increase flow by δ on each forward edge and decrease flow by δ on each backward edge (see Figure 2.13). This process is called augmentation and it is the basic operation of the augmenting path algorithms. Such algorithms work as follows.

1. Start with zero flow.

2. Find augmenting path from s to t with reserve δ.

3. Augment the flow along the augmenting path by δ.

4. Repeat steps 2,3 until no augmenting path can be found.

Figure 2.14: Two possible types of vertices and numbering of their incident edges.

Correctness of this algorithm is based on the following theorem.

Theorem 2.14 (Ford, Fulkerson [FF62]) *The flow in the graph is maximum if and only if there is no augmenting path.*

There are many algorithms for finding maximum flow based on this generic algorithm. Efficiency of such algorithm is influenced by two things. First, we need to find augmenting paths efficiently and second, in case that there are several augmenting paths available we need to choose among them such path that not many augmenting steps are needed. The Weihe's algorithm achieves $O(n \log n)$ time in the following way.

- Augmenting paths are chosen so that only $O(n)$ augmentations are necessary,
- Appropriate augmenting path is found and flow augmented in $O(\log n)$ time using dynamic trees data structure.

Details of dynamic trees data structure will be provided later, first we will describe in more details how does the algorithm work. The algorithm stores certain subset of augmenting paths in a tree T. The tree is rooted in t and contains a subset of the vertices of the graph. For each vertex v included in T the only path in T going from v to t is an augmenting path. Edges of the graph can be inserted to tree T either in forward or in backward direction but in both cases they should have positive reserve in direction from child to parent.

As far as s is in the tree we can take the path in T from s to t and augment along it. However augmentation causes that some edges will have 0 reserve and must be deleted from the tree. Deleting edges can split the tree into several parts and we need to do some modifications in order to restore the tree. This is achieved by a combination of adding new edges to the tree and isolating some vertices from the tree. The algorithm can be outlined as follows.

1. Construct initial tree T containing all vertices.
2. **while** s is in the tree **do**
3. augment flow along the path in T from s to t
4. find saturated edge e (i.e. with 0 reserve) on this path
5. remove it from tree T and rebuild tree T

Note that augmentation can cause that more than one edge will become saturated. In this case the algorithm chooses the edge nearest to t and removes it. Therefore the tree can temporarily contain some edges with zero reserve. Several augmentation steps can occur with $\delta = 0$ which help to remove the edges with 0 reserve. This is equivalent to the algorithm which would after each augmentation remove edges with 0 reserve until the reserve on the path from s to t would be positive.

Our goal is to maintain the tree so that after s becomes isolated (disconnected from t) the flow will be a maximum flow. Next part describes in more detail how tree T is created and updated.

2.2.1 Tree construction and maintenance

Take vertex $v \neq s, t$. Recall that it has degree 3, and since the graph does not contain useless edges, vertex v has at least one incoming and one outgoing edge. Therefore v has one of the two forms depicted in Figure 2.14. For each vertex v we number its incident edges with numbers $1, 2, 3$ as is shown in the figure. For s the only incident edge will have number 1. Edge number i incident to vertex v will be denoted $e_v[i]$.

Tree T is initialized very simply. We include in T edge $e_v[3]$ for each $v \neq s, t$ and edge $e_s[1]$. All these edges are included in forward direction (and therefore they have positive reserve). It can be shown that these edges make a spanning tree of the graph.

Maintenance of the tree is much more complicated. Recall that in each step we augment a path from s to t by its reserve which causes one or more edges to be saturated. We pick the one of them which is closest to t and remove it from T. The removed edge connected some node v to its parent in the tree. Node v and its subtree are now disconnected from the tree. We will try to reconnect v to the tree.

Important feature is this. Each vertex can be connected to its parent in T by any of its three incident edges (recall that edges can be inserted to tree T backwards as well). In initial tree each vertex v is connected to its parent by edge $e_v[3]$. When we remove this edge from T we will try to reconnect v to the tree with edge $e_v[2]$. If it is not possible we will use $e_v[1]$. Edge that is either removed from the tree or is unsuccessfully considered for inserting to the tree is called *rejected.* The rule is that for vertex v we consider edges in order $e_v[3]$, $e_v[2]$, $e_v[1]$ (i.e. in counterclockwise order) and once some edge is rejected it is never considered again in the algorithm in this direction (of course it can be used in opposite direction to connect the other endpoint as a child of v).

If we fail to connect v to tree T by its non-rejected edges we will leave v isolated from the tree. In this case however we need to disconnect all children of v and try to reconnect them to T in the same way as we tried to reconnect v. This reconnecting is done by calling recursive procedure link_to_tree_or_isolate depicted in Figure 2.15. The procedure is called for a vertex v such that one of its edges was recently rejected. The procedure either reconnects v to tree T using non-rejected edges or isolates v leaving the tree in a consistent state. The procedure might call itself recursivelly after it rejects some other edge. Here we will try to describe the procedure in more detail.

Assume that edge $e_v[i]$ was recently rejected. It might be the case that $i = 1$ and in this case we cannot reconnect v to the tree because all incident edges were already rejected. Therefore we have to isolate v. It means that for each child w of v we remove edge connecting it to v from the tree and recursivelly call link_to_tree_or_isolate for vertex w.

In the case that the rejected edge was not $e_v[1]$ we consider edge $e_v[i-1]$ and we try to connect v to T using this edge. In the ideal case this happens and we are done. However there are several cases in which it is not possible and other solutions should be used. We will now discuss these cases (let w be the other end of edge $e_v[i-1]$).

- **Vertex w is already isolated.** In this case inserting edge $e_v[i-1]$ does not help to connect v to T. Therefore this edge is rejected and we call link_to_tree_or_isolate for vertex v again.

- **Edge $e_v[i-1]$ has 0 reserve in direction from v.** We do not want to insert edges with 0 reserve to T and therefore we reject this edge and do a recursive call for v.

- **Vertex w is a descendant of v.** In this case inserting edge $e_v[i-1]$ would create a cycle and we for sure cannot have cycles in a tree. We solve this case in maybe slightly surprising way (see Figure 2.16). We first remove the edge connecting w to its parent from the tree and insert edge $e_v[i-1]$. Now v is connected to w and w is disconnected from the tree. Therefore we do a recursive call for w to link it back to T or to isolate it.

  Two remarks should be made about this case. First, it applies also when w is directly a child of v – in this case we simply change direction of edge (v, w) in the tree and then call procedure for w. One important invariant is that the cycle formed by the path from w to v in T and edge $e_v[i-1]$ is always

Procedure Link_to_tree_or_isolate(v)
let $e_v[i]$ be the edge recently rejected for v

if $i = 1$ then
    isolate(v)   { *no edges left* }
    return

let w be the other end of $e_v[i-1]$

if exceptional1 then
    reject $e_v[2] = (w', v)$
    disconnect w' from its parent (removing (w', v))
    Link_to_tree_or_isolate(w')
    Link_to_tree_or_isolate(v)
    return

if one of the following cases occurs:
    $e_v[i-1]$ has 0 reserve in direction from v, or
    w is an isolated vertex, or
    exceptional2
then
    reject edge $e_v[i-1]$
    Link_to_tree_or_isolate(v)
    return

if w is descendant of v then   { *cycle* }
    disconnect w from its parent (removing edge)
    insert (v, w) into the tree
    Link_to_tree_or_isolate(w)
    return

otherwise
    insert (v, w) into the tree

Figure 2.15: Pseudocode for procedure Link_to_tree_or_isolate(v)

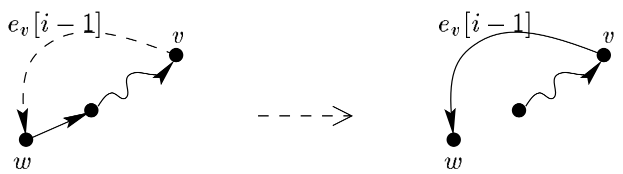


Figure 2.16: Vertex w is a descendant of v in T and adding edge $e_v[i-1]$ would create a cycle. Figure shows only edges in T directed always from child to parent (they can be directed in opposite way in G).

counter-clockwise (note that this is not directly implied by the fact that the graph does not contain clockwise cycles because T can contain edges in the opposite direction as well). In order to make the algorithm working we need to avoid clockwise cycles and this is achieved by prohibiting certain special exceptional cases which are discussed below.

- **Inserting of $e_v[i-1]$ would cause an exceptional situation.** Exceptional situation is defined as follows: there is a vertex u such that it is incident to two incoming and one outgoing edge (as on the left side of the figure 2.14), edge $e_u[2]$ connects u to some child in T and edge $e_u[1]$ connects u to its parent in T. Invariant is that such situation may never occur in T. Inserting edge $e_v[i-1]$ may cause an exceptional situation either in vertex v or in vertex w. We consider these two subcases separately.

  First let us assume that the exceptional situation would occur at w. This situation is called exceptional2 in pseudocode and is defined as follows:

  - $e_v[i-1] = e_w[2]$, and
  - $e_w[2]$ is incoming to w, and
  - w is linked to the tree either by edge $e_w[1]$ or by edge $e_w[2]$

  Note that if w is linked by $e_w[1]$ we have exceptional immediately whereas if it is linked by $e_w[2]$ we would get exceptional case in the next step because we would reject $e_w[2]$ as a consequence of adding $e_v[i-1]$ and we would try to use $e_w[1]$. If exceptional2 occurs we simply reject edge $e_v[i-1]$ and thus we prevent this type of exceptional situation.

  The second case is when the exceptional situation occurs at v. It means that some child is connected to v via $e_v[2]$ and we consider the edge $e_v[1]$ for linking v to T. Since we consider edge $e_v[1]$ it means that we rejected $e_v[2]$ recently. Edge $e_v[2]$ cannot be used in both directions (from v and from the other incident vertex w') and therefore there are two cases how this may happen – either w' decided to use $e_v[2]$ for connecting to v and thus removed it from the tree in direction from v and inserted it in the opposite direction. However this case is already covered in exceptional2. The second case is when we remove $e_v[3]$ from tree, consider edge $e_v[2]$ but reject it and afterwards consider $e_v[1]$. We reject $e_v[2]$ only if it has 0 reserve in direction from v. This is the only case we are interested in. It is caught by condition called exceptional1 which is defined as follows:

  - $e_v[2]$ is incomming and has 0 reserve in direction from v
  - we consider $e_v[2]$ (i.e. $i-1=2$) and this edge is connects w' as a child of v

  In this case we do the following. We remove $e_v[2]$ from the tree, thus disconnecting w', then we do a recursive call first for w' and then for v.

  Therefore we detect the danger of the exceptional situation in v in advance (one step before it could happen) and we treat it in more complicated way than simply rejecting edge $e_v[1]$. Note that if we rejected $e_v[1]$ we would isolate v and afterwards we would disconnect w' from v rejecting one of its edges. In our algorithm we disconnect w' as well but afterwards there is no danger of exceptional situation and we might as well use $e_v[1]$ for connecting v to the tree. This might help us to find some augmenting path which would be otherwise discarded.

We have now concluded description of maintainance of tree T. However in order to obtain $O(n \log n)$ time complexity we need to store tree T in a special data structure. And here is where dynamic trees structure comes to the stage. Details will be given in the following section.

2.2.2 Use of dynamic trees data structure

Dynamic trees is a data structure published in [ST83]. It stores collection of vertex disjoint rooted trees with weighted edges. The structure can be initialized in $O(n)$ time as a collection of isolated vertices and a number of operations can be performed on this structure in $O(\log n)$ time. Some of these operations are very usefull in Weihe's algorithm.

We store tree T and possibly some isolated vertices in dynamic trees structure. Weight of an edge in T will be the reserve on this edge in direction from child to parent. Notice that it is possible to compute the flow on an edge if we know its capacity and reserve in one direction. Therefore for edges in T we will not store current value of flow, it is stored solely as a reserve of the edge inside the dynamic trees structure. This is indeed necessary because an augmenting path may contain up to $\Omega(n)$ edges and if we explicitly store and update flow for each edge it would yield $O(n^2)$ algorithm. We store the current flow value only for edges which are not currently in the tree and flow on these edges is not changed during augmenting.

link(v,w,r) links v as a child of w (v should be root before) and the new edge (v, w) will have weight r.

This operation is used for inserting edges to T where r is set as a reserve of the corresponding edge in direction from v to w.

cut(v) disconnects v from tree by removing the edge connecting v to its parent in the tree.

It is used for removing edges. Each time we remove an edge from T we first find out its reserve in T (using operation value(e) described below), we compute the flow on the edge and store this flow.

value(e) returns a value of the edge e in the tree.

This is used on various places in the program whenever we need to know the current flow on some edge that is currently stored in T. One example is the procedure Link_to_tree_or_isolate where we need to test whether the edge has 0 reserve or not.

root(v) returns the root of the tree containing vertex v.

This is used in main loop of the algorithm for testing whether s is still in the tree – we check whether root(s) $= t$. It is also used in testing whether node w is descendant of v in procedure Link_to_tree_or_isolate(v). In this case we now that v is disconnected from the rest of the tree and therefore if w is descendant of v, then root(w) $= v$.

findmin(v) returns the edge with minimum value on the path from v to the root. If there are more such edges it returns one which is the nearest to the root.

The operation is used to find edge e with the smallest reserve on the augmenting path from s to t. This reserve δ will be then used for augmenting and afterwards e will be removed from T.

augment(v,δ) decreases values of all edges on the path from v to the root by value δ.

This is of course used in augmenting the path from s to t in T.

2.2.3 Time complexity

Now let us summarize the time complexity of the algorithm. Each edge is rejected at most twice – once for each endpoint. Before each call of Link_to_tree_or_isolate we reject one edge and between any two augmentation step we reject one edge. By carefull implementation we may achieve that between rejecting two edges we perform $O(1)$ dynamic trees operations and the rest of work can be done $O(1)$ time. In this way we spend $O(n \log n)$ time in dynamic trees operations and $O(n)$ time in other work performed. Initialization of T and construction of initial zero flow takes $O(n)$ time. In the final phase we need to remove all edges from T and record their flow which again takes $O(n \log n)$ time for $O(n)$ dynamic trees operations. Together we see that the core part of the maximum flow algorithm can be performed in $O(n \log n)$ time. Recall that preprocessing phase can be done in $O(n \log n)$ time and back transformation in $O(n)$ time and therefore we have $O(n \log n)$ time algorithm for computing maximum flow in arbitrary directed planar graph.

2.2.4 Correctness

We will not provide proof of correctness of the algorithm in this report. The proof can be found in [Wei97]. However we do provide some details, namely the invariants on which the proof is based.

First of all it is easy to see that the algorithm computes some flow, because it updates the flow only using augmenting. The problem is to prove that the found flow is maximum, because it is not clear that no augmenting paths are left in the graph upon the termination of the algorithm.

The idea behind the algorithm is to "sweep" the graph with the augmenting path from right to left, so that we start with the right-most path and changing it so that we always move to the left in some node (recall that we start with edge $e_v[3]$ for each vertex and continue to consider edges in counter-clockwise order). There might be some augmenting paths on the right side of the current augmenting path but maximum flow can be reached even if these paths are never used in augmentation.

This was, however only rough idea, and the invariants used in real proof are more complicated for technical reasons. For stating them we need one additional term. *Exceptional path* is a path that goes through a vertex v with 2 incomming edges so that it enters v via $e_v[2]$ and leaves v via $e_v[1]$.

Invariant 1. Before each augmentation the dynamic trees structure contains a tree rooted in t and possibly some isolated vertices.

Invariant 2. Let e be an edge that was already rejected for vertex v. Then all augmenting paths from v to t are exceptional.

Invariant 3. There is a maximum flow f^* such that it can be obtained from current flow f by augmenting only along non-exceptional paths.

It can be proved quite easily that if these invariants hold the algorithm computes a maximum flow. However the proof of invariants (especially invariants 2 and 3) is quite complicated and involves extensive case analysis.

Chapter 3

Implementation

We have implemented Weihe's algorithm including preprocessing and back transformation. We have used following resources:

Platform: Linux (Debian distribution ver. 2.1 was used) on PC

Programming language: GNU C++ (egcs-1.1.1 release)

Libraries: LEDA ver. 3.8 (see [LED99]) object package (research license)

Unfortunately our implementation does not follow exactly algorithms described in section 2. The reason is that we have discovered some of the errors in the original article (see section A) only during preparation of this report. Therefore we decided not to rework our implementation in this phase and rather to explain what are differences between the implementation and the algorithm described in section 2.

The rest of this chapter is divided as follows:

What we have implemented is a section describing differences between the implementation and the algorithms described in section 2.

Data structures. This part is devoted to description of used data structures, including data structures provided by LEDA library.

Usage and testing. In this section we are describing interface to our implementation (how can be parts of our implementation reused), testing environment (including results of testing) and algorithm animation module.

3.1 What we have implemented

The augmenting and back transformation part of the algorithm follows description in 2. The difference between the implementation and the algorithm described in section 2 is in the preprocessing phase. The preprocessing phase consists of the following sequence of transformations:

1. **Clockwise cycles removal.** This part works as it is described in section 2.1.1.
2. **Source and sink modifications.** This part adds new source with the only outgoing edge with enough capacity and removes all edges outgoing from sink. Removal of edges outgoing from sink was not required in the algorithm described in section 2.

```
for each vertex v in G do
    v' = v

    while v' is of degree 2 do
        // one edge must be incomming and one outgoing
        incoming=incoming edge to v', u=source of incoming
        outgoing=outgoing edge to v', w=target of outgoing

        if u = w then
            remove incoming and outgoing edges
            v' = u        // degree of u has changed, it can be 2
        else
            insert edge (u, w) into embedding
            remove incoming and outgoing

remove all vertices of degree 0
```

Figure 3.1: Removing vertices of degree 2

3. **Unreachable vertices removal.** Removes vertices which cannot be reached from s or cannot reach t (using depth first search). Together with blow-up of vertices as it was implemented this part was supposed to replace removal of useless edges. See section 3.1.1 for more detailed description.

4. **Removal of vertices of degree 2.** This part is little more complicated than corresponding part in section 2. The problem is that since we have not removed useless edges so far, we can have vertex u connected by two antiparallel edges with vertex v which is of degree 2. In this case we have to remove both edges and also vertex v. Unfortunately this also affects degree of vertex u which now can become 2. Therefore it was necessary to code this part carefully to obtain correct implementation in $O(n)$ time (see figure 3.1 for details).

5. **Blow-up vertices.** This part follows original description in Weihe's article (different from blow-up transformation described in section 2) with one slight change: we are blowing up also verices of degree 3. For the description see section 3.1.1.

As we will describe later (section 3.1.1) removal of unreachable vertices together with blow-up of vertices was supposed to satisfy assumption 2.2. Unfortunately, blow-up procedure **introduces new useless edges to the graph** (Weihe supposed it does not) and therefore cannot be used as part of preprocessing routine. We have reworked blow-up procedure (see section 2.1.3) to correct this problem, however it cannot be used with our approach to useless edges any more.

Conclusion. We have implemented algorithm with preprocessing which does not guarantee satisfaction of assumption 2.2. Instead of this assumption we can guarantee only the following weakened assumption.

Assumption 3.1

- *For each vertex $v \in V$ there exists directed path from s to v and directed path from v to t.*
- *There are no parallel or antiparallel edges in G.*

We have tested our implementation extensively (see section 3.3) and we have not encountered any problem with it. Therefore the following conjecture can be formulated:

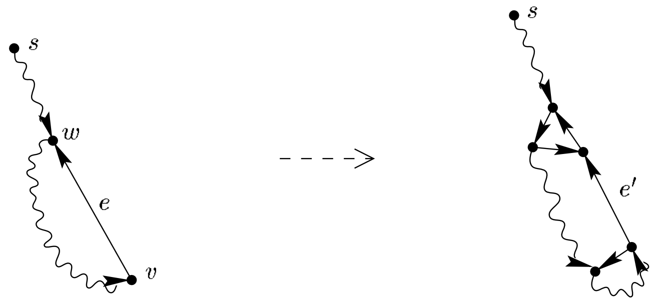


Figure 3.2: Useless edges are not longer useless after blow-up transformation

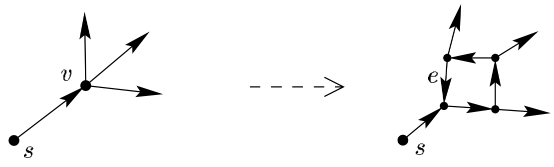


Figure 3.3: Assume that all three edges leaving v are not useless. After blowing up vertex v edge e is useless (there is no simple directed path from s going through e).

Conjecture 3.2 *Weihe's algorithm is correct for graphs satisfying assumptions 2.1, 3.1 and 2.3*

We have used implementation of several data structures in LEDA library. Dynamic trees implementation which we are using (also coming from LEDA) implements operations on dynamic trees in $O(\log^2 n)$ time. Therefore our implementation has time complexity $O(n \log^2 n)$.

3.1.1 Original blow-up method and useless edges

In the original article the blow-up method replaced every vertex except s and t in G with counterclockwise cycle. The number of vertices in the cycle was the same as degree of original vertex and exactly one edge incident to the original vertex was connected to each of them (therefore every vertex is of degree 3 after blow-up).

Notice that it is possible to solve the problem of useless edges in the following way [Bie99a]. Consider $e = (v, w)$ is an edge for which there is no directed simple path starting in s containing e. However both of their ends (v and w) can be reached from s. Therefore there must be a directed path starting in s and ending in v containing w (see figure 3.2).

Now consider graph G' after blow-up transformation. In G' there is an edge e' corresponding to edge e. However in G' there exists simple directed path from s including e' and therefore (because similar argument holds also for directed path to t starting with e') edge e' is no longer useless.

However as it can be seen on figure 3.3, the problem with this solution is that **blow-up transformation introduces new useless edges into the graph** regardless of the fact whether there were any useless edges in the original graph or not. Therefore blow-up transformation as it is was described in Weihe's article **cannot be used in preprocessing** if we want to satisfy assumption 2.2.

3.2 Data structures and implementation details

In this section we introduce data structures which we have used in our implementation. In the first part we briefly describe structures provided by LEDA library. In the second part we discuss representation of network and in the third part we deal with back transformation.

In the implementation of the core of Weihe algorithm (augmenting part) we tried to follow pseudocode in the paper as much as possible (after necessary corrections).

3.2.1 Used LEDA data structures

Data structures for graphs

LEDA provides many variations of data structures for storing graphs. We describe only structures which we have used in our implementation. For further reference see [MNSU99].

Directed graphs. Directed graph consists of *nodes* (vertices) and *edges*. Both nodes and edges can have additional information stored with them. It is possible to create new nodes/edges using provided methods. Methods for obtaining obvious information (source of edge, target of edge, information assigned with node/edge) are provided.

For each node two *incidency lists* are maintained separately: list of incoming and list of outgoing edges[1]. Several macros are provided to *iterate* through all nodes/edges in the graph and incidency lists of any node.

Not all information needs to be stored directly in nodes or edges. It is possible also to create *node/edge arrays* which behave like an ordinary array with index space of node/edge pointers. These structures basically cannot handle insertions of new nodes/edges (unless you specify how much additional space for new nodes/edges you need to have). These arrays are extremely useful for storing auxiliary information. In particular we do not need to declare a special structure for each node/edge which stores temporary information needed in various phases of the computation. Instead we create auxiliary arrays as we need them.

Nodes/edges are created using methods of graph object. Therefore they are strongly connected to this object and cannot be reused in other graphs. It is quite natural in object-oriented point of view. However sometimes (especially in preprocessing phases of graph algorithms) we need to create slightly modified copy of a graph and it is convenient to think about nodes/edges in the new graph and in the original graph as the same objects. Since in structures mentioned above nodes/edges cannot be reused, we need to keep track of such relation ourselves.

Although structures for directed graphs provided by LEDA seem to be very simple they are extremely powerful and convenient to use.

Embeddings. Recall that incoming and outgoing edges for a node are stored in the two separate lists. However planar embeddings are usually stored by ordering edges incident to each vertex in counter-clockwise order. It is not straightforward to do it in these separate lists of incoming and outgoing edges.

Therefore LEDA introduces so called *bidirected graphs* for representing embedded undirected graphs. Each undirected edge is represented by two antiparallel directed edges (these two edges are called *reversal* of each other). In a bidirected graph it is possible to set/receive reversal edge for each edge.

An embedding in bidirected graph is defined by order of edges in outgoing incidency list of each node. Method *make_planar_map* is provided to find a planar embedding of a planar graph and order outgoing incidency lists according to this embedding (takes $O(n)$ time).

It is also possible to work with faces similarly as with nodes or edges. First it is necessary to call method *compute_faces* (takes $O(n)$ time; results are valid until graph is modified). Afterwards it is possible to use arrays and iterator macros similar to those available for edges and nodes. Since each edge is represented by two antiparallel directed edges there is exactly one face corresponding to each directed edge (a face on the left). To obtain face on the right it is possible to ask reversal edge for the information. Also additional macro iterating all edges adjacent to the face is available.

[1] It is possible to force edges to be stored in specific order in incidency lists

Other LEDA structures used

Dynamic trees. There is a dynamic trees structure implemented in LEDA library which we need for tree maintenance in Weihe's algorithm augmenting part. Implementation provided by LEDA takes $O(\log^2 n)$ amortized expected time per operation. It is possible to implement dynamic trees in $O(\log n)$ time per operation [ST83].

Priority queues. We use priority queues for Dijkstra algorithm implementation in clockwise cycle removal. LEDA provides implementation by Fibonacci heaps with $O(\log n)$ amortized time per operation.

Interface to dynamic trees and priority queues. For both these structures we have built separate interfaces working directly with structures we needed. The reason was to keep technical details (such as pointer casts) away from main code.

Lists. We are using this simple data structure in removal of vertices of degree 2.

3.2.2 Representation of networks and flows

Networks. We are using LEDA bidirected graph structure to store networks. For each network edge (u, v) we store two edges: (u, v) with capacity information and (v, u) with special *NOFLOW* value (which cannot be a capacity). These edges are set to be reversal edges of each other. We call edges of the network *flow edges* and their reversals *noflow edges.*

Flows. Flows are represented as edge arrays where information stored for noflow edges is unimportant.

3.2.3 Back transformation

Link structure. Have two graphs G and G'. We can establish a *link* between them which helps us to compute flow on G from given flow on G'. A link is a simple edge array on G where each element is record with the following structure:

plus	pointer to an edge in G'
minus	pointer to an edge in G'
circ	value

Suppose we have a flow f' on G'. Provided that we have link l between G and G' we compute flow f on G as follows ($f'[nil]$ is considered to be 0):

$$f[e] = f'[l[e].plus] - f'[l[e].minus] + l[e].circ$$

We call link *simple* if all minus fields are nil. Otherwise we call link *complicated.*

Transformations. The back transformation is thus very easy. Now we will provide details how to maintain link from original graph to preprocessed graph during preprocessing phase of the algorithm.

Assume we already performed several steps of preprocessing and we have modified graph G' and link l from original graph G to G'. Now we want to perform additional step on G' and to keep l up-to-date.

In most preprocessing phases[2] it works as follows:

[2] The only exception to this rule is blowing-up vertices where it is simpler to create modified graph directly instead of working on copy

1. **Create a copy G'' of G' with link l' established between them.** This is done by straightforward copying routine. Notice that l' needs to map only plus values to the copy of each edge in G''.

2. **Transform graph G'' maintaining l'.** This part is straightforward for most preprocessing transformations (generally by removing an edge we need to set corresponding link to nil; addition of an edge does not change anything). There are two exceptions to this:

   Clockwise cycles removal. We are adding new edges which have to be linked to proper minus fields. See section 2.1.1 for more information.

   Removal of vertices of degree 2. Here we simply do operations on graph G'' and maintain "inverse" links from G'' to G'. More precisely we maintain for each edge in G'' a list of corresponding edges in G'. Replacing a path with an edge corresponds to concatenating lists of corresponding edges in inverse link. At the end we compute link from inverse link by iterating through all lists.

3. **Update link l with l'.** After we have finished the preprocessing step we want to update l so that it now points to corresponding edges in G''. This is done by first following link from G to G' and then using values in l' to update it. Notice that there is only one transformation (namely clockwise cycles removal) which creates complicated links. Therefore we never need to update a complicated link with another complicated link.

Using this approach enables us to discard all intermediate graphs created during preprocessing and to keep only the original graph, the preprocessed graph and the link between them.

3.3 Usage and testing

3.3.1 Interface to the implementation

There are three reusable functions building an interface to our algorithm implementation. These functions allow to enter the computation in various stages.

int graph_flow(network &G, flow &F); underlying graph of a network should be directed planar graph (not our bidirected representation) with capacities set as an edge information; the function returns flow F or non-zero if error occurs.

int map_flow(network &G, flow &F); underlying graph of a network should be represented as it was described in section 3.2.2, edges should be ordered according to a combinatorial embedding; the function returns flow F or non-zero if error occurs.

void compute_flow(network &G, flow &F); underlying graph of a network should be represented as it was described in section 3.2.2, edges should be ordered according to a combinatorial embedding; moreover the graph should meet assumptions 2.1, 2.2 and 2.3; the function returns flow F; this function runs only the core (augmenting part) of Weihe's algorithm.

3.3.2 Testing

Base testing method. We have tested our implementation using following testing process:

1. **Generate random directed planar graph.** In this part we have used *random_planar_graph* generator provided in LEDA library. It works as follows.

(a) A maximal planar map with n nodes is iteratively constructed (for $n \leq 3$ the graph is given uniquely; for $n > 3$ the construction is used for $n-1$ first and afterwards additional node is put into a random face).

(b) All but m edges are deleted afterwards.

This algorithm is fast, however it is known that not all planar graphs can be generated in this way. LEDA provides other random planar graph generator which can be used in such testing (based on geometric construction).

2. **Generate capacities on edges.** According to given parameters several antiparallel edges are added and capacities on all edges are generated. Edges incident to s and t can be multiplied by given constant to obtain less trivial cuts in the resulting graph.

3. **Save resulting graph.** The generated graph is saved for the case of program failure. In such case we can rerun experiment with the same graph.

4. **Run implementation.** Afterwards our implementation of Weihe's algorithm is run giving us a flow in graph G.

5. **Check the flow.** Then resulting f is tested whether it represents a flow in graph G and whether this flow is maximum. We are using LEDA *CHECK_MAX_FLOW* to do this job.

6. **Iterate** these steps. If any error occurs stop the process.

Here is the sample of outputs of our testing routine. Testing was run under Linux operating system on 400 Mhz pentium with 32MB RAM (WOW! means that flow was checked and test passed):

```
$ ./test -n5000 -m1000 -c1,100 -r5
Graph: generate randomly
n=5000, m=10000
capacity: min=1 max=100
probability of "two way" edges: 50%
factor for multiplying edges incident to s and t: 10
number of tests: 5 (if -1, test forever)
backup file: BACKUP [LEDA]

Test number 1:
Return code: 0; Elapsed time: 16 s
Nodes after preprocessing: 35609
Found flow of value 306
WOW!
Test number 2:
Return code: 0; Elapsed time: 13 s
Nodes after preprocessing: 36276
Found flow of value 93
WOW!
Test number 3:
Return code: 0; Elapsed time: 18 s
Nodes after preprocessing: 35378
Found flow of value 211
WOW!
Test number 4:
Return code: 0; Elapsed time: 15 s
Nodes after preprocessing: 35421
```

```
Found flow of value 273
WOW!
Test number 5:
Return code: 0; Elapsed time: 12 s
Nodes after preprocessing: 35698
Found flow of value 174
WOW!
```

Testing results. We have tested the algorithm on about 20000 graphs with different parameters (n from 3 to 5000, m from 5 to 10000). We have not encountered any problem with our implementation. We were not able to test our implementation for larger graphs (memory requirements exceeded parameters of testing computer and extensive swapping decreased performance too much).

We were also watching for garbage which our implementation possibly could leave in a memory. For this we have run a test for about 10000 large graphs. The amount of memory allocated by program varied but there was no increase apparent with the number of graphs processed.

Preprocessed graphs. Graphs after preprocessing are quite large. Usually for every edge (including antiparallel edges) more than 2 nodes in the preprocessed graph are generated (each of degree 3). Also during preprocessing phase we usually need to store two graphs of this size at the same time. It causes large memory requirements.

Debugging. Using our testing routine we have discovered several bugs in our implementation. Following LEDA features were helpful in debugging:

- **Checked operations.** Preconditions of many operations with LEDA structures are checked (for example references to non-existing objects, etc.) and any misusing is reported immediately to the error output and execution is terminated.
- **Good identifier naming conventions.** Most of LEDA identifiers are reasonably named. Therefore call stack dump after the crash of the implementation tells lot about nature of the error.
- **CHECK_MAX_FLOW.** This works as an assertion in the program, and if given f which is not maximum flow, it terminates with reasonable error message (which describes why given f did not pass).

3.4 Animation

We have also implemented a simple animation of the core of Weihe's algorithm. In this section we describe several details on this animation. Animation tools are to be used as follows:

1. Take a graph satisfying all assumptions and run core of Weihe's algorithm on it. This produces log file containing all details on algorithm progress necessary for animation.
2. Generate straight-line planar drawing of the graph.
3. Animation program takes input graph, log file and coordinates of vertices and run animation on it.

Animation shows maintenance of tree T, augmenting paths used in each step and current value of flow on each edge. In figures 3.4 and 3.5 you can see main steps of animation for given graph.

Log file approach allows us to separate implementation details of animation and the algorithm implementation itself. Therefore the algorithm implementation can still be used for purposes other than animation.

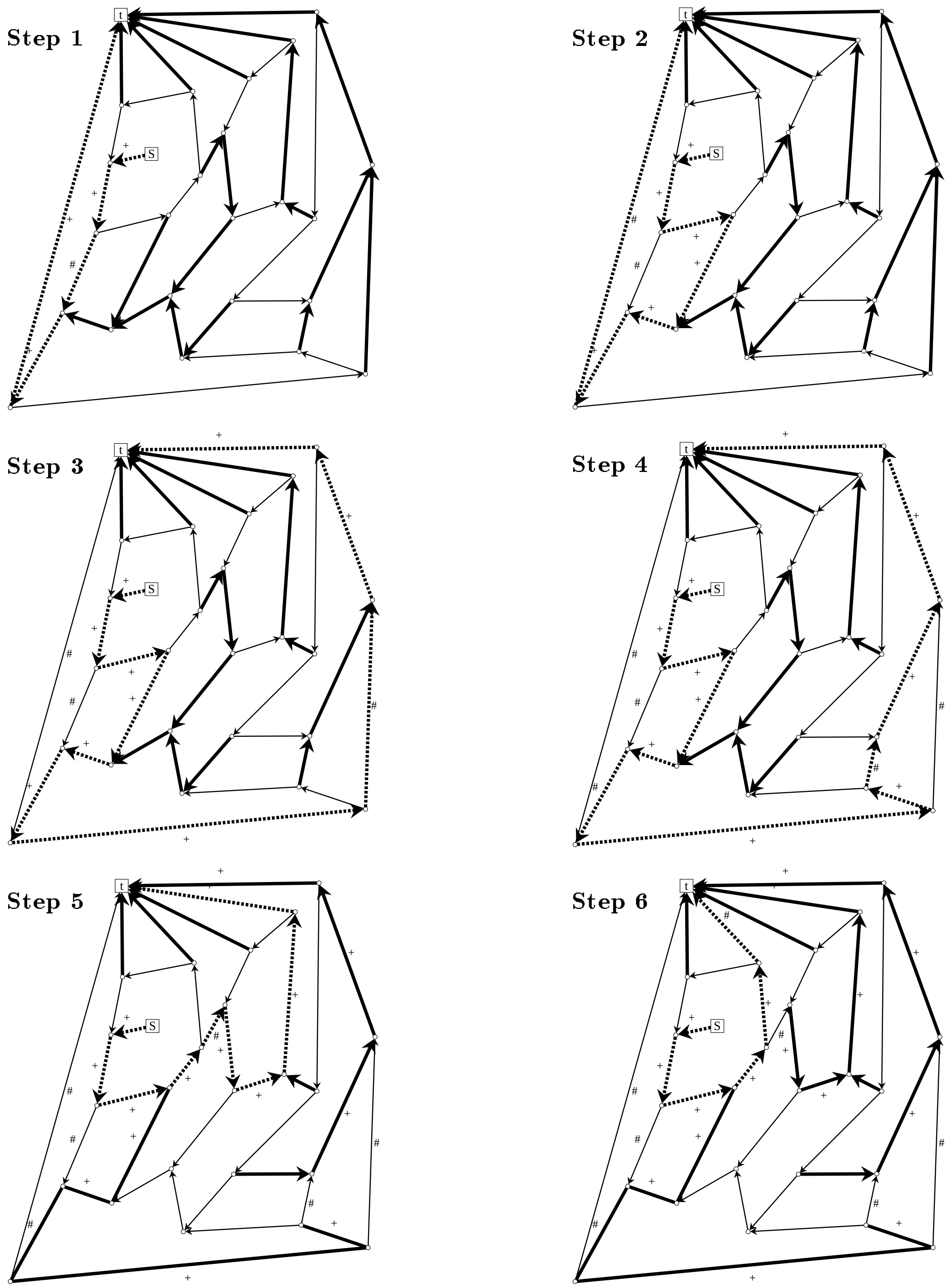


Figure 3.4: The example of how algorithm works. Boldface lines represent edges in the tree (edges without arrows are edges which are entered into the tree in the backward direction). Dashed lines represent augmenting paths. Labels indicate amount of flow on each edge (**+** - some flow on edge, **#** - full capacity is used). Notice that after step 4 some vertices became isolated.

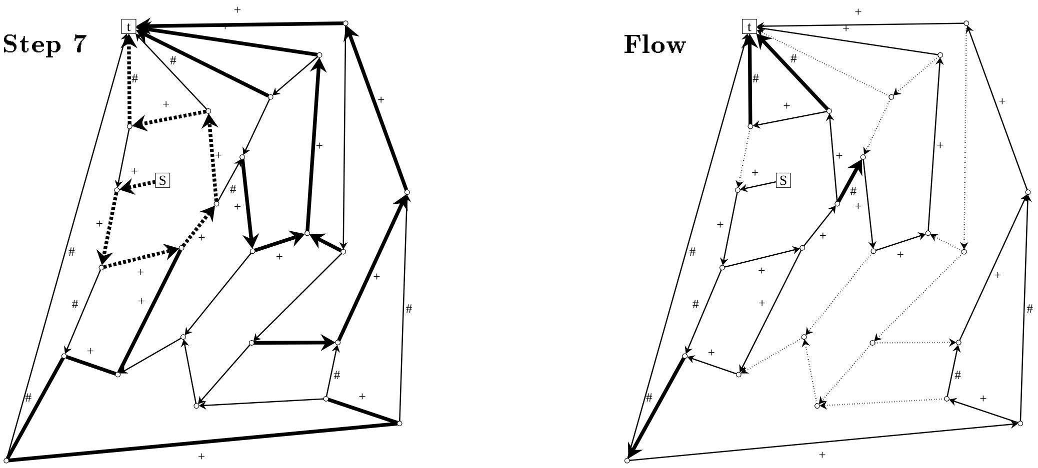


Figure 3.5: The example of how algorithm works continued. On the last figure it is possible to see maximum flow with minimum cut highlighted.

Implementation. LEDA *GraphWin* was used as an user interface. This part of LEDA is easy to use and provides relatively friendly user interface without much programming necessary. However we did not have not so good experience with LEDA as before. The main reason for this is probably that we have used research (experimental) parts of LEDA that are not yet well developed, documented and reliable.

Graph drawings. LEDA provides several (experimental) routines for graph drawings. Generally there were two problems with drawings:

- Drawings are usually ugly (this is problem of algorithms used not the problem of LEDA).
- There is no "legal" way how to force LEDA drawing routines to choose given face as outerface (fortunately, if graph is already embedded routines follow this embedding).

To solve the second problem we have made additional assumptions on how the particular routine works and rearranged list of vertices in such way that it takes outer face as we need it (however it is not necessary compatible with future versions of LEDA which can contain other implementations of the routine). To solve the first problem we allow user to rearrange graph drawing.

Chapter 4

Project history

The previous parts of the report describe the results of our project from various points of view. Here we would like to show how we got those results and what we were actually doing all that time.

Step 1 — LEDA. Shortly after starting to work on our project we decided to use data structures for graphs implemented in LEDA library. However it took use quite long to find a working version of LEDA. Graduate machines at the university had only old versions often without documentation and a new version was to big to fit into our user space quotas. We tried to install one at home but it refused to work with our version of a compiler. Finally we managed to get a working version after reinstalling our machine with a different installation of a Linux... All this took more than one week. But we succeeded.

Step 2 — study. Meanwhile we read Weihe's paper to see what we were going to implement. It turned out that in order to do preprocessing we would need to read also another paper — [KNK93]. This paper establishes an algebraic structure of lattice in the set of circulations in a planar graph and things needed in our preprocessing were hidden on various places. The result of our attempt to find all what was needed and to summarize it can be found in section 2.1.1.

Step 3 — useless edges. Still in the phase of reading papers and thinking about them we decided to work out details of detecting useless edges (see Assumpion 2.2). We thought it would be some easy excercise with depth first search but we could not find a solution. A visit to Therese Biedl also did not help. So we send an e-mail to Karsten Weihe asking him how he meant it. It turned out that he just did not realized that it might be a problem. However he sent us some algorithm which finds useless edges (see page 41). Meanwhile we also found an algorithm but it was very complicated (it can be found in section 2.1.2). When we presented these things to Therese Biedl she came up with a nice and easy-to-implement idea how to avoid this problem. Her solution was then actually implemented in our project (see section 3.1.1).

Step 4 — implementing augmenting. After we have got a working version of LEDA we started to implement the core part of Weihe's algorithm. At the beginning we needed to study LEDA manual a lot and we found out that there are several different classes for representing graphs. The one that seems to be natural choice (parametrized planar maps) turned out to be not very usefull. But after we have decided on right data structures (which are described in section 3) it was quite straightforward to rewrite pseudocode from [Wei97] to the program. After a while we became familiar with all necessary LEDA methods and everything went well.

Step 5 — finding the first errors. When we finished coding the core of the algorithm we tried to run it on some simple examples. Of course, it did not work. There were some our bugs, but not all the bugs were ours. We have discovered that the pseudocode in [Wei97] contains several errors, one of them in exceptional1, which is the most tricky part of the algorithm. Therefore we needed to study the informal description of the algorithm very carefuly to find out correct solution of these problems (see section A).

Step 6 — preprocessing. When we finished implementing the core of Weihe's algorithm we implemented preprocessing. It turned out that this was much more difficult. There was no nice pseudocode one could follow. We had to think about a way how to keep information needed for back transformation. We had to make copies of a graph and to keep links between copies of vertices and edges. Lots of pointers everywhere. We wrote a procedure for removing vertices of degree 2 which was the longest of all preprocessing steps and we found out that it does not work and must be reworked from the beginning. Together preprocessing had twice as long code as the core of the algorithm and it took us quite a long time to do it.

Step 7 — testing. So after writing entire program we started to test it on some random graphs. At the beginning we discovered some bugs but afterwards everything was working and we were satisfied with the result of our work.

Step 8 — animation. In order to enjoy our program even more we wrote a simple animation and we were now able to see right on our screen how the tree changes. Everything was so great.

Step 9 — another error. As the presentation of the project was approaching we prepared our slides. And as we were looking on a figure showing the trick with avoiding detection of useless edges we suddenly realized that it does not work. It was only one day before the presentation so we decided to left it as it was and to show the error at the end of talk as a surprise.

After we calmed down a little bit we found out that the problem was not in avoiding removing useless edges but in blowing vertices (see Figure 3.3) because it can create new useless edges. Then we discovered modification of blowing vertices that does not create new useless edges (see section 2.1.3) but this modification cannot be used for avoiding useless edges.

So we were in a situation where we were able in principle to correct our implementation but it would be quite complicated (it would be necessary to add a new procedure for removing useless edges and also to change substantially the procedure for blowing up vertices). Due to the time constrains we decided to leave it as it is.

Step 10 — yet another error. When we were writing this report we wanted to describe Weihe's solution to the problem of removing useless edges in chapter Algorithms. However as we were preparing the proof we found out that it is not that easy to prove it, and than we even found a counter-example showing that the algorithm does not work. So we have simply moved the algorithm from chapter Algorithms to chapter Errors...

At the beginning we had three possible approaches to removing useless edges. Our approach was most complicated and also least efficient. But it turned out that after getting rid of all other algorithms this is the only choice that seems to work (at least nobody found an error in it yet).

Does it work or not? Our implementation of preprocessing can leave some useless edges in the input graph. However the algorithm found the maximum flow correctly on so many inputs so it is not quite clear whether it is really needed to remove such edges. It seems from our correspondence that even Karsten Weihe is not quite sure about it.

Achievements. In the project we have achieved its main goal to implement Weihe's algorithm (although the preprocessing is not absolutely satisfactory). We have also found several errors or omittions in the paper. Finally, we have designed an algorithm for finding useless edges and summarized all information needed to preprocessing.

What we have learned. We have learned to use LEDA which was a good experience. After initial problems with installation we found our work with LEDA quite pleasant. Basic graph structures are usefull and nicely designed. However more advanced features such as animation tools were not documented and did not work very reliably. Anyway, if we would start from scratch now, we would use LEDA again.

The second big lesson from this project and entire cs760k course is that figures can be very decisive and planar graphs are an area where it is easy to make an error and difficult to find it.

Dreams. Our implementation of preprocessing produces graphs that may contain useless edges. This problem could be solved in several ways. One possibility is to implement the algorithms described in this report. Another way is to prove that it is possible to weaken the assumption 2.2 so that our preprocessing satisfies it. It would be also nice to understand Weihe's proof of correctness so that it will be then possible to say whether the assumption can be weakened. Or to test the program more extensivelly using some better methods of generating graphs and to discover a case where it fails.

Appendix A

Errors in [Wei97]

In this chapter we summarize errors and missing parts found in Weihe's paper [Wei97] during the work on our project. Usually we also include suggestions for improvement. The left column contains a short citation from the paper indicating location of an error, the right column describes why it is an error and how it can be solved. The last error mentioned is not from the paper but it is an error in the algorithm provided in [Wei99].

Page: 455
This algorithm determines a certain *circulation* in G, (...) The algorithm in [KNK93] requires $O(n \log n)$ time. We refer to [KNK93] for further details.

This citation is slightly misleading. There is one algorithm working in $O(n \log n)$ time described in this paper, and it could be probably used in preprocessing. However this algorithm is more general that is needed and can be replaced by computing shortest paths using Dijkstra algorithm. This approach is mentioned in [KNK93] but not explicitly as an algorithm. We guess that the author did not refer to this approach since he is aware of the fact that the shortest paths can be computed in $O(n)$ time in planar graphs.
Description of computing the circulation using shortest paths can be found in section 2.1.1.

◇ ◇ ◇

Page: 455
Assumption 2.2. (...) To satisfy this assumption, simply remove all arcs that violate it.

It is easy to remove such arcs (called useless edges in this report), however it is difficult to find them in a graph. For a possible solution see section 2.1.2.

◇ ◇ ◇

Page: 455
Next remove all arcs leaving t, as to ensure the assumption about t.

No edge leaving t does belong to a simple directed path ending in t. It can belong only to a simple directed cycle. Therefore all edges leaving t are useless. It is not necessary to remove them explicitly because they are removed together with all other useless edges.

◇ ◇ ◇

Page: 455-456
Thus we replace (u, v) and (v, w) by (u, w), and the capacity of (u, w) equals the minimum of the capacities of (u, v) and (v, w). (Parallel arcs are merged afterwards, and their capacities are added up.)

There are two possible interpretations of this description of removing vertices of degree 2. The first one is that we first replace all paths by edges and we merge parallel edges afterwards. It is possible to implement merging of all parallel edges in a graph in $O(n)$ time. However this approach is not correct because merging parallel edges can create new vertices of degree two and we would need to repeat the process again and again which would cause $O(n^2)$ complexity.
The second approach might be to merge the parallel edges immediately after creating each new edge. But having a vertex of a high degree, how can one possibly even to detect a parallel edge in $O(1)$ time? Maybe it would be possible to use some data structures and to implement it in $O(n \log n)$ but surely it is not straightforward.
We propose the solution to leave parallel edges in a graph and to use blowing vertices for all vertices, even for vertices of degree 3 (see section 2.1.3).

◇ ◇ ◇

Page: 456
After we "blow up" each vertex $v \in V \setminus \{s, t\}$ with a degree greater than 3 to a counterclockwise cycle as shown in Fig. 1. (...)
Obviously, the modified instance is completely equivalent to the original one, and Assumptions 2.1 and 2.2 are still valid after these modifications.

This is not true because when we expand each vertex to a counterclockwise cycle we can introduce new useless edges (see Figure 3.3).
We propose a modification of blowing up vertices which can be found in section 2.1.3.

◇ ◇ ◇

Page: 457
FUNCTION saturated (v : Vertex; a : Arc) : Boolean;
IF v is the tail of a
    THEN return $f[a] = c_a$
    ELSE return $f[a] = 0$;

Note that function saturated(v, a) is called also for edges that are currently in the tree and their flow is not updated in array f. Therefore it is necessary to call F.value for such edges.

◇ ◇ ◇

Page: 457
FUNCTION exceptional_1 (v, w: Vertex) : Boolean;
return $(w, v) \in A$ AND $f_{(w,v)} = 0$;

First, it is unclear what $f_{(w,v)}$ does refer to. It should find value of the flow from w to v and therefore it cannot be a simply test in an array f, it must be accompanied by a call of F.value where appropriate.
The second problem is that the condition covers even the cases that are not exceptional. In particular we need to test whether $(w, v) = a_2^v$ and whether w is a child of v in the tree.

◇ ◇ ◇

Page: 457
PROCEDURE link_to_t_or_isolate (v : Vertex);
(...)
4. IF exceptional_1 (v, w) THEN
   (a) link_to_t_or_isolate (w);
   (b) link_to_t_or_isolate (w);
(...)

Before calling link_to_t_or_isolate (w) we need to call remove$(a^w_{out_going[w]})$.

◇ ◇ ◇

Page: 460
Our scan strategy implies that each such arc is non-active in direction out of C except for (possibly) the unique arc incident to C at v and except (...)

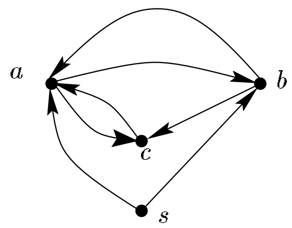
This is a small typo where "incident to C at v" should be probably replaced by "incident to C at w".

◇ ◇ ◇

Algorithm for removing edges from [Wei99].[a]
Useless edges can be found in planar graph in $O(n \log n)$ time as follows. Choose outerface to be a face incident to s. Using depth-first search starting from s construct left-most and right-most DFS trees. The right-most tree can be found so that we start from s with an edge that has outerface on its right side and in each vertex we process edges in counterclockwise order starting with the edge immediately following edge that was used to discover that vertex. Left-most tree is found analogically.
Any edge, that is not an back edge in at least one of the two trees, belongs to some simple directed path from s. Edges that are back edges in both trees are useless (do not belong to a simple directed path from s). Analogical process is then repeated for t.

[a]Here we do not provide original Weihe's wording but instead our short formulation of the algorithm.

Idea of Weihe's proof is to take edge (v, w) which is back edge in both trees and to argue that any path from s to v must go to the right of the $s - v$ path in the left-most tree and to the left of the $s - v$ in the right-most tree. However these two paths have a vertex w in common and therefore every path from s to v must go through w.
This proof is however not correct. See for example the following graph:



The right-most tree is path s–b–a–c and the left-most tree is path $s - a - b - c$. Edge (c, a) is a back-edge in both trees, but it also belongs to simple directed path $s - b - c - a$.

Bibliography

[Amm96] Leendert Ammeraal. *Algorithms and data structures in C++*. Wiley, New York, 1996.

[Bie99a] Therese Biedl. Personal communication. 1999.

[Bie99b] Therese Biedl et al. Advanced graph algorithms. http://magnolia.math.uwaterloo.ca/~biedl/teach/cs760K, Fall 1999. Class Notes, University of Waterloo, Course CS760K.

[FF62] L. R. Ford and D. R. Fulkerson. *Flow in Networks*. Princeton University Press, 1962.

[Has81] Refael Hassin. Maximum flow in (s,t) planar networks. *Information Processing Letters*, 13(3):107, 13 December 1981.

[Hu69] Te Chiang Hu. *Integer Programming and Network Flows*. Addison-Wesley, 1969.

[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, Montréal, Québec, Canada, 23–25 May 1994.

[LED99] LEDA: library of efficient data structures and algorithms, 1999. http://www.mpi-sb.mpg.de/LEDA.

[MN00] Kurt Mehlhorn and Stefan Nähler. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, February 2000. to be published.

[MNSU99] Kurt Mehlhorn, Stefan Näher, Michael Seel, and Christian Uhrig. The LEDA user manual, version 3.8, 1999.

[PS85] Franco P. Preparata and Micheal Ian Shamos. *Computational Geometry: An Introduction*. Springer–Verlag, 1985.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[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.

[Wei99] Karsten Weihe. Personal e-mail correspondence. 1999.