Simplifying flow networks

Therese Biedl Broňa Brejová Tomáš Vinař [*]

March 1, 2000

Abstract

Maximum flow problems appear in many practical applications. In this paper, we study how to simplify a given directed flow network by finding edges that can be removed without changing the value of the maximum flow. We give a number of approaches which are increasingly more complex and more time-consuming, but in exchange they remove more and more edges from the network.

1 Background

The problem of finding a maximum flow in a network is widely studied in literature and has many practical applications (see for example [AMO93]). In particular we are given a network (G, s, t, c) where $G = (V, E)$ is a directed graph with n vertices and m edges, s and t are two vertices (called *source* and *sink* respectively) and $c : E \to R^+$ is a function that defines capacities of the edges. The problem is to find a maximum flow from s to t that satisfies capacity constraints on the edges.

Some graphs contain vertices and edges that are not relevant to a maximum flow from s to t. For example, vertices that are unreachable from s can be deleted from the graph without changing the value of maximum flow. In this article we study how to detect such useless vertices and edges.

Study of such a problem is motivated by two factors. First, some graphs may contain a considerable amount of useless edges and by removing them we decrease the size of the input for maximum flow algorithms. Also, the optimization of the network itself may be sometimes desired.

Second, some maximum flow algorithms may require that the network does not contain useless edges to achieve better time complexity. For example Weihe in [Wei97] presented an algorithm for computing a maximum flow in a directed planar network in $O(n \log n)$ time. Weihe's algorithm works only for a special subclass of planar graphs, and he claims that it is possible to transform any planar graph to a graph from this subclass without changing the value of a maximum flow. One step of this transformation is to remove what we call s-or-t-useless edges (see Section 2). However no algorithm for finding such edges is described

[*]Department of Computer Science, University of Waterloo, {biedl,bbrejova,tvinar}@uwaterloo.ca

in [Wei97] and as far as we know this problem was not investigated before. We implemented Weihe's algorithm, and found in testing more than 20000 randomly generated graphs that it seems to work even when not all s-or-t-useless edges have been removed (see [BV99]). This however has not been proved formally.

In this paper, we study how to remove such s-or-t-useless edges, and more generally, any edge that is not useful. The precise definition of a useful edge is as follows:

Definition 1 *We call an edge e* useful *if for some assignment of capacities, every maximum flow uses edge e. If e is not useful, then we call it* useless.

Note in particular that the definition of a useful edge depends only on the structure of the network and the placement of the source and sink, but not on the actual capacities in the network. Thus, a change in capacities would not affect the usefulness of an edge.

The above definition of a useful edge is hard to verify. As a first step, we therefore develop an equivalent definition which uses directed paths.[1]

Lemma 2 *An edge e is useful if and only if there exists a simple path P from s to t that uses e.*

Proof: Assume first that there exists a simple path P from s to t that uses e. Then let the capacities of the network to be 1 for every edge on P and 0 on all other edges. Clearly, the maximum flow in this network is 1, and any maximum flow must use the edges of path P, hence e.

Conversely, assume that e is useful, thus there exists some assignment of capacities such that e belongs to any maximum flow in the network. Let f be one such flow. By the Flow Decomposition Theorem (see for example [AMO93, p. 80]) f can be decomposed into a set of flows along simple paths from s to t and flows along simple directed cycles. Let f' be the flow that consists of only the flows along paths; this flow has the same value as f and therefore is also a maximum flow. So edge e is used by f' and must belong to one of the simple paths from s to t, which proves the claim. $\square$

It would seem a natural approach to extend the definition of useful edges in a way that takes capacities into account. However, such an extension is not straightforward. Let a *simple flow* be a flow that can be decomposed into a set of flows along simple paths from s to t (i.e., there are no directed cycles in its decomposition). A useful edge may be then defined as an edge that belongs to at least one maximum simple flow. However, we will not pursue this alternative definition.

The characterization of a useful edge in Lemma 2 can be used to give yet another characterization.

Lemma 3 *An edge $e = (v, w)$ is useful if and only if there exist vertex-disjoint paths P_s from s to v and P_t from w to t.*

[1] In this paper, we will never consider paths that are not directed, and will therefore drop "directed" from now on.

Proof: Assume e is useful, thus there exists a simple path P from s to t that uses e. We can break this path into two paths by removing e from it, and these are the two desired paths, which are vertex-disjoint because P is simple. Conversely, if we have the two paths P_s and P_t, then by adding e to them, we obtain a simple path from s to t containing e. □

With this observation, we can show that the problem of finding useless edges is hard in general. Namely, Fortune, Hopcroft and Wyllie showed that it is NP-hard to test, given four distinct vertices s_1, s_2, t_1, t_2 in a directed graph, whether there are two vertex-disjoint simple paths connecting s_1 with t_1 and s_2 with t_2, respectively [FHW80]. This, by the above characterization, corresponds exactly to our problem, except that we need to add an edge (t_1, s_2). Adding this edge cannot make the problem easier, because by vertex-disjointness condition none of the desired paths could use this edge anyway. Hence, testing whether a given edge is useful is NP-complete.

Since the problem is NP-complete, we simplify it in two ways:

- Find edges that are clearly useless, without guaranteeing that all useless edges will be found. In particular, in Section 2 we will give a number of conditions under which an edge is useless. For each of them we give an algorithm how to find all such edges. These algorithms are increasingly more complex.

- Restrict our attention to planar graphs without clockwise cycles. We present in Section 3 an $O(n)$ time algorithm to test whether a given edge is useless.

We conclude in Section 4 with open problems.

2 Finding some edges that are useless

As proved above, an edge (v, w) is useless if and only if there are two vertex-disjoint paths P_s from s to v and P_t from w to t. Unfortunately, it is NP-complete to test whether an edge is useless. We will therefore relax this characterization of "useless" in a variety of ways as follows:

- We say that an edge $e = (v, w)$ is *s-reachable* if there is a path from s to v, and *s-unreachable* otherwise.

  We say that an edge $e = (v, w)$ is *t-reachable* if there is a path from w to t, and *t-unreachable* otherwise.

- We say that an edge $e = (v, w)$ is *s-and-t-reachable* if it is both s-reachable and t-reachable, and *s-or-t-unreachable* otherwise.

- We say that an edge $e = (v, w)$ is *s-useful* if there is a *simple* path from s to w that ends in e, and *s-useless* otherwise. (Note that if e is s-reachable, then there always exists a path from s to w that ends in e, but it may not be simple because it may use w twice.)

  We say that an edge $e = (v, w)$ is *t-useful* if there is a simple path from v to t that begins with e, and *t-useless* otherwise.

- We say that an edge $e = (v, w)$ is *s-and-t-useful* if it is both s-useful and t-useful, and *s-or-t-useless* otherwise.

Notice that each edge that is "useless" according to some of these definitions, is clearly useless according to Definition 1. Unfortunately, even an s-and-t-useful edge is not necessarily useful. In Figure 1, we give an example of an edge e that is useless, but that is both s-useful and t-useful. One can easily verify that any paths P_s from s to v and P_t from w to t must cross.

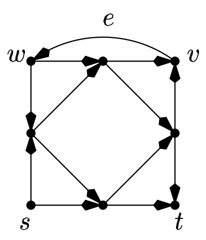

Figure 1: Edge e is s-useful and t-useful, but useless.

We study for each of these concepts how to detect such edges in the following subsections.

2.1 s-reachable edges

Recall that an edge $e = (v, w)$ is s-reachable if there exists a path from s to v. It is well-known (see e.g. [AMO93, p. 73]) that such edges can be detected with a directed search from s. If we reach vertex v during this traversal, then e is s-reachable, otherwise it is s-unreachable. Thus, we can find all s-unreachable edges in $O(m + n)$ time.

Once we have detected (and removed) s-unreachable edges, the graph is surely connected. Hence from now on we will assume that the input graph is connected.

2.2 s-and-t-reachable edges

Testing whether an edge is t-reachable is as easy as testing whether it is s-reachable: all we need to do is to perform a depth-first search, starting from t, in the *reverse graph*, i.e., the graph with all edge-directions reversed.

However, the question arises whether removing t-unreachable edges could ever make another edge s-unreachable, hence forcing us to start over. As we show now, this is not the case:

Lemma 4 *Let G be a network, let G' be the network that results from removing all s-unreachable edges in G, and let G'' be the network that results from removing all t-unreachable edges in G'. Then all edges in G'' are s-and-t-reachable.*[2]

[2]This lemma might seem quite trivial. However, as we will see later, the equivalent statement does *not* hold for s-and-t-useful edges, and hence this lemma is not completely obvious.

Proof: Let $e = (v, w)$ be an edge in G''. Since e was not removed when computing G' from G, it was s-reachable in G. So in G there exists a path P_s from s to v. All edges on P_s also belong to G', because these edges are all s-reachable in G by using path P_s.

Since e was not removed when computing G'' from G', it was t-reachable in G'. So in G' there exists a path P_t from w to t. All edges on P_t also belong to G'', because these edges are all t-reachable in G' by using path P_t. So P_t belongs to G'', which means that e is t-reachable in G''.

Let $e' = (v', w')$ be any edge in P_s. Since both P_s and P_t belong to G', we can combine them with (v, w) to obtain a path from w' to t in G'. Therefore, e' is t-reachable in G', and will also belong to G''. So P_s belongs to G'', which means that e is s-reachable in G''. $\square$

Hence, to obtain a graph for which all edges are s-and-t-reachable, we simply need to run two depth-first-searches, one from s in the input graph, and one from t in the reverse graph. This takes $O(m + n)$ time.

2.3 s-useful edges

Recall that an edge e is *s-useful* if there exists a simple path starting at s and using e, and *s-useless* otherwise. Note that not every s-reachable edge (v, w) is s-useful, because it might be that all paths from s to v must go through w. Figure 2 shows an example of an edge that is s-reachable but s-useless.

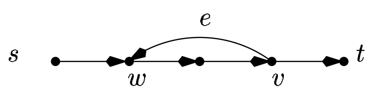

Figure 2: Edge e is s-reachable and t-reachable, but s-useless and t-useless.

On the other hand, this is the only way that an s-reachable edge could be s-useless. More precisely, an edge $e = (v, w)$ is s-useful if and only if there is a path from s to v that does not pass through w.

With this observation, it is very simple to determine s-useless edges: for every vertex w, remove w from the graph and perform a depth-first search from s. Afterwards we can check for every edge $e = (v, w)$ whether v is still reachable (and therefore whether edge e is s-reachable in the original graph). This algorithm takes $O(m)$ time per vertex, and hence $O(mn)$ time total.

We will improve on the time complexity of this simple algorithm, and present an algorithm to find all s-useless edges in $O(m \log n)$ time. To achieve this asymptotic bound, the algorithm uses two data structures: interval trees and dynamic trees. We first briefly review these two structures.

Interval trees An *interval tree* is a structure to store intervals whose end-points belong to some fixed set X of n points. Interval trees are used frequently in computational geometry and details of their implementation can be found for example in [PS85]. 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 (assuming that interval (a, b) was inserted before).
- **Covered(x)** returns true if there is an interval (a, b) in the tree with $a < x < b$.

Furthermore, an interval tree can be initialized in $O(n)$ time as an empty tree.

Dynamic trees A *dynamic tree* stores a collection of vertex-disjoint rooted trees on a fixed set of n vertices with weights on the edges. They were first introduced by Sleator and Tarjan [ST83], and have many applications in graph algorithms. The following operations are supported by dynamic trees in $O(\log n)$ time:[3]

- **Link(u,v,x)** inserts a new edge (u, v) with weight x. It is assumed that u was a root before. Vertex v now becomes the parent of u.
- **Change_value(u,x)** changes the weight of the edge from u to its parent to be x. It is assumed that u is not a root.
- **Min_value(u)** returns the minimum weight of an edge on the path from u to the root of the tree containing u. It is assumed that u is not a root.

Furthermore, a dynamic tree consisting of n isolated vertices can be initialized in $O(n)$ time.[4]

2.3.1 Overview of the algorithm

Our algorithm works by doing three traversals of the graph. The *first traversal* is simply a depth-first search starting in s. Edges not examined during this search are s-unreachable, hence s-useless, and can be removed immediately. We will not consider such edges in the following.

Let T be the depth-first search tree computed in the first traversal. Let $v_1, v_2, \ldots, v_n$ be the vertices in the order in which they were discovered during the depth-first search. Let the *DFS-number* $num(v)$ be such that $num(v_i) = i$.

A depth-first search of a directed graph divides the edges into the following categories.

- **Tree edges** are the edges included in the DFS-tree T.
- **Back edges** are the edges leading from a vertex to an ancestor.
- **Forward edges** are the edges leading from a vertex to a descendant that is not a child.

[3]Two versions of dynamic trees algorithms are presented in [ST83]; the simpler one achieving amortized complexity $O(\log n)$ per operation is sufficient for our purposes.

[4]The data structures presented in [ST83] support many more operations than are needed in our algorithm. Also, it is possible to modify our algorithm in such way that Change_value is not needed. However we were not able to find a simpler structure supporting Link and Min_value operations in $O(\log n)$ time.

- **Cross edges** are the edges leading from a vertex to a vertex in a different subtree. By properties of a depth-first search, such edges always lead to a vertex with a smaller DFS-number.

See Figure 3 for an illustration.

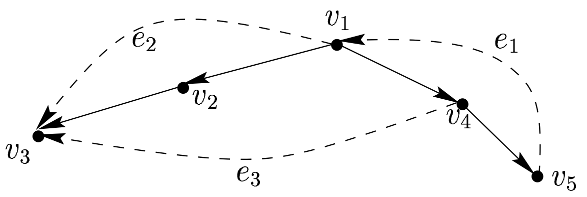

Figure 3: Types of edges created by a directed depth-first search. Tree edges are solid. Edge e_1 is a back edge, e_2 is a forward edge and e_3 is a cross edge.

In the *second traversal* we compute for each vertex v a value *detour*(v), which roughly describes from how far above v we can reach v without using vertices in the tree in between. The precise definition is as follows:

Definition 5

- *If u and v are two vertices, and u is an ancestor*[5] *of v in the DFS-tree, then denote by $T(u,v)$ the vertices in the path between u and v in the DFS-tree (excluding the endpoints), and let $T(u,v]$ be $T(u,v) \cup \{v\}$.*
- *Assume that vertex u is an ancestor of vertex v. A path from u to v will be called a* detour *if it does not use any vertices in $T(u,v)$. (We allow $u = v$, in case of which any path from v to v is a detour.)*
- *For an edge $e = (w,v)$, we denote by detour(e) the minimum value i for which there exists a detour from v_i to v that uses edge e as its last edge. (Note that a detour can start only in a vertex v_i that is an ancestor of v.)*
- *For a vertex v, we denote by detour(v) the minimum value i for which there exists a detour from v_i to v. In particular therefore, detour(v) is the minimum of detour(e) over all incoming edges e of v.*

This definition is illustrated in Figure 4.

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

[5] A vertex v is considered to be an ancestor of itself.

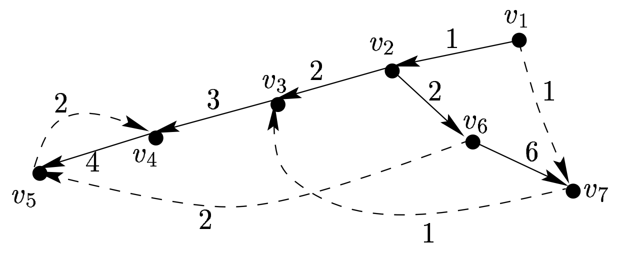

Figure 4: Detours in a graph. The DFS-tree is shown with solid lines. Every edge e is labeled by $detour(e)$.

2.3.2 The second traversal

We compute detours by scanning vertices in reverse DFS-order. Let us assume that we already know $detour(v_j)$ for all $j > num(v)$ and we want to compute $detour(v)$. In particular therefore, we want to compute $detour(e)$ for all incoming edges e of v. We distinguish cases by the type of e:

- $e = (w, v)$ **is a tree edge or a forward edge.** We must have $detour(e) \le num(w)$, because e itself is a path from w to v, and w is an ancestor of v. On the other hand, for any ancestor v_i of v with $i < num(w)$, any path from v_i to v using e must use vertex w which belongs to $T(v_i, v)$. Hence, $detour(e) = num(w)$.
- $e = (w, v)$ **is a cross edge or a back edge.** We give the formula for computing $detour(e)$ in the following lemma.

Lemma 6 *Let $e = (w, v)$ be a cross edge or a back edge, and let a be the nearest common ancestor of v and w ($a = v$ if e is a back edge). Then*

$$detour(e) = \min_{u \in T(a,w]} detour(u). \tag{1}$$

Proof: Let $d = \min_{u \in T(a,w]} detour(u)$. Since e is a cross edge or a back edge, w is not an ancestor of v, so $a \ne w$ and $T(a, w]$ contains at least one vertex. In particular, $T(a, w]$ contains a child z of a, which has $detour(z) \le num(a)$; therefore $d \le detour(z) \le num(a)$.

Let u be a vertex in $T(a, w]$ that achieves the minimum in Equation (1), so there exists a detour from v_d to u. By $d \le num(a)$, vertex v_d is an ancestor not only of u, but also of a, and therefore also of v. We can obtain a detour P from v_d to v using e as follows: start with the detour Q from v_d to u, then follow tree-edges from u to w, and finally follow edge e. See the left picture of Figure 5.

We claim that $Q - \{v_d\}$ does not contain a vertex b with $num(b) < num(u)$. Assume to the contrary that $Q - \{v_d\}$ contains such vertex and let b be the last such vertex on path $Q - \{v_d\}$. Let (b, b') be the next edge in Q, so $num(b') \ge num(u)$. By $num(u) > num(b)$, this implies that (b, b') is a tree edge or a forward edge, and in particular b' is a descendant of b.

Since b' is a descendant of b, all vertices x with $num(b) \leq num(x) \leq num(b')$ are also descendants of b. In particular therefore u is a descendant of b and b is an ancestor of u. But b cannot belong to $T(v_d, u)$, because Q is a detour from v_d to u. It also cannot be a proper ancestor of v_d, because otherwise the path Q restricted from b to u would be a better detour than Q. Finally by assumption $b \neq v_d$, so we have a contradiction.

This claim shows that none of $T(v_d, v)$ can belong to Q (all vertices in $T(v_d, v)$ have DFS-number smaller than $num(u)$), so P is indeed a detour for e and $detour(e) \leq d$.

On the other hand, let $detour(e) = num(x)$. Observe that $T(a, w] \cup (w, v)$ is a detour from a to v since w is not an ancestor of v; in particular therefore $num(x) \leq num(a)$. If $num(x) = num(a)$, then by $num(a) \geq d$ we have $detour(e) \geq d$ as desired; so assume $num(x) < num(a)$. Let P be the detour from x to v that ends with edge $e = (w, v)$. Then $P' = P - \{v\}$ avoids all vertices in $T(x, a] \subset T(x, v]$. Let y be the first vertex of P' in $T(a, w]$; this must exist since $w \in P'$. Then P' defines a path from x to y that avoids all vertices in $T(x, y) = T(x, a] \cup T(a, y)$. In particular therefore $detour(y) \leq num(x)$, and $d \leq detour(y) \leq detour(e)$ as desired. See the right picture of Figure 5. $\square$

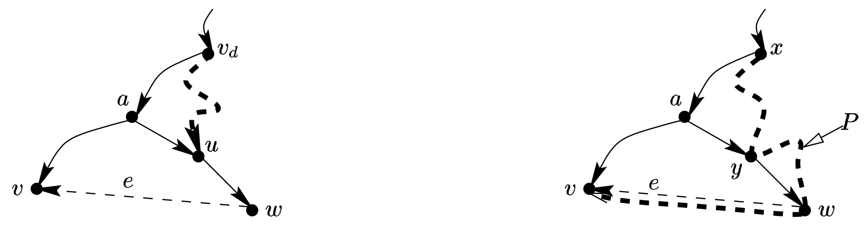

Figure 5: Computation of $detour(e)$ for a cross edge. The case of a back edge is similar.

Using this lemma, we can compute the detour-values during the second traversal. We use dynamic trees, which we first initialize to a set of n isolated vertices, and then insert the tree edges, with weight ∞. This takes $O(n \log n)$ time.

Now we process the vertices in reverse DFS-order (i.e., from v_n to v_1). During this computation, we will change the weight of the edges. More precisely, after v_j, $j > 1$, has been processed, we will set the weight of the edge connecting v_j to its parent to have weight $detour(v_j)$.

To compute $detour(v_i)$, we compute $detour(e)$ for all incoming edges e of v_i and take the minimum. We compute $detour(e)$ for $e = (w, v_i)$ as follows: If e is tree edge or forward edge then simply $detour(e) = num(w)$. Otherwise, we need to compute

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

where a is the least common ancestor of v and w. Note that by the properties of a depth-first search we have $num(a) \leq num(v_i) = i$ and $num(u) > num(v_i) = i$ for all $u \in T(a, w]$. Therefore all edges in $T(s, a)$ have weight ∞, while all edges in $T(a, w)$ reflect the detour-values. Thus $\min_{u \in T(a,w]} detour(u)$ is the same as the minimum along the path from w to s. In particular, we need not compute the least common ancestor to find the detour-value, but can find it with one call to Min_Value. Hence we can determine $detour(e)$ in $O(\log n)$ time,

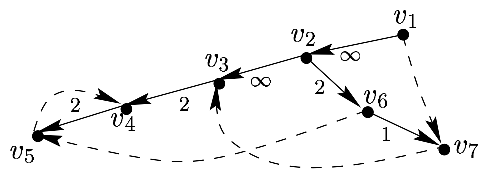

Figure 6: Example of the second traversal. The figure shows the values of the tree edges stored in dynamic trees structure at the time of processing node v_3. Note that Min_Value(v_7) returns 1 (as a minimum of $\infty, 2, 1$) and therefore for edge $e = (v_7, v_3)$ we have $detour(e) = 1$ and this will be also value of $detour(v_3)$.

and $detour(v_i)$ in $O(\deg(v_i)\log n)$ time. Finally, we need to update the weight of the edge from v_i to its parent, which takes $O(\log n)$ time.

Altogether we need $O(n)$ time for initialization of dynamic trees structure, $O(n \log n)$ time for initially inserting tree edges with infinite weight, and $O(\deg(v)\log n)$ time for each vertex v. Altogether, this traversal therefore takes $O(m \log n)$ time.

2.3.3 The third traversal

In the third traversal, we determine for each edge $e = (v, w)$ whether it is s-useful. If e is a tree edge, a forward edge or a cross edge, then the path from s to v in tree T followed by e is simple and e is s-useful. Thus we only consider back edges, and need the following lemma.

Lemma 7 *Let $e = (v, w)$ be a back edge. Then e is s-useful if and only if for some $u \in T(w, v]$ we have $detour(u) < num(w)$.*

Proof: Assume first that there is a vertex $u \in T(w, v]$ with $d = detour(u) < num(w)$. Then the detour P from v_d to u does not use $w \in T(v_d, u)$. Let P' be the path that consists of following tree edges from s to v_d, then P, and then tree edges from u to v. This path does not use w, hence there exists a simple path from s to v that does not use w, and e is s-useful.

Conversely, assume that e is s-useful, hence there exists a path P from s to v that does not use w. Let u be the first vertex on P that belongs to $T(w, v]$ (it might be v), and let v_d be the last vertex before u on P that is an ancestor of u (it might be s). Then $v_d \notin T(w, v]$ (by definition of u), and $v_d \neq w$ (because $w \notin P$), so v_d is an ancestor of w. Also, the part of P from v_d to u does not use any ancestor of u, and in particular no vertex of $T(v_d, u)$, so is a detour for u. We conclude that $detour(u) \leq d < num(w)$. □

To actually determine the s-useful edges, we use interval trees, where the endpoints of intervals are in $\{1, 2, \ldots, n\}$. We perform yet another depth-first search (with exactly the same order of visiting vertices). When reaching vertex v, the interval tree contains intervals of the form $(detour(u), num(u))$ for all ancestors u of v. We add interval $(detour(v), num(v))$ into the tree. When we retreat from v, we delete $(detour(v), num(v))$ from the interval tree. This takes $O(\log n)$ time per vertex.

Assume now we are currently processing vertex v. For each back edge (v, w) we want to check whether there is a vertex $x \in T(w, v]$ with $detour(x) < num(w)$. But this is the case if and only if $detour(x) < num(w) < num(x)$ and x is an ancestor of v. But the interval $(detour(x), num(x))$ is stored in the interval tree for all ancestors of v; thus edge $e = (v, w)$ is s-useful if and only if Covered($num(w)$) returns true. This takes $O(\log n)$ time per back edge, and hence $O(m \log n)$ time total.

Theorem 8 *All s-useless edges can be found in $O(m \log n)$ time.*

The complete algorithm is given in Figure 7.

```
Main Program
  1. Compute the DFS-tree $T$ and DFS-numbers
  2. Initialize a dynamic trees structure DT with the set of vertices $\{v_1, v_2, \ldots, v_n\}$
     For each $e = (v, w) \in T$ do
         DT.Link($w, v, \infty$)
     For $v_n, v_{n-1}, \ldots, v_1$ do
         Compute_detour($v_i$)
         DT.Change_value($v_i$, $detour(v_i)$)
  3. Initialize an 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 7: Pseudocode of algorithm for finding s-useless edges in $O(m \log n)$ time

2.4 s-and-t-useful edges

Recall that an edge e is s-or-t-useless if it is s-useless or t-useless. In the previous section we have shown how to find all s-useless edges in $O(m \log n)$ time. The algorithm can be adapted easily to find all t-useless edges, simply by reversing the direction of all edges in the graph and using vertex t instead of s. Therefore the following lemma holds.

Lemma 9 *All s-or-t-useless edges of a graph can be found in $O(m \log n)$ time.*

Surprisingly so, this lemma does not solve the problem how to obtain a graph without s-or-t-useless edges. As shown in Figure 8, removing all s-or-t-useless edges from graph G may introduce new s-or-t-useless edges. More precisely, removing t-useless edges may introduce new s-useless edges and vice versa.

To obtain a graph without s-or-t-useless edges we therefore need to repeat the procedure of detecting and removing s-or-t-useless edges until no such edges are found. The question

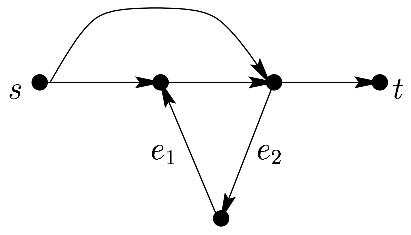

Figure 8: Edge e_1 is s-and-t-useful, but edge e_2 is t-useless. After removing e_2, edge e_1 becomes s-useless.

arises how many iterations are needed. Clearly, each iteration removes at least one edge, which gives $O(m)$ iterations and $O(m^2 \log n)$ time. Unfortunately, $\Omega(n)$ iterations may also be needed. Figure 9 shows a graph with $3k + 2$ vertices and $7k - 1$ edges that requires $2k$ iterations. Each iteration removes one edge.

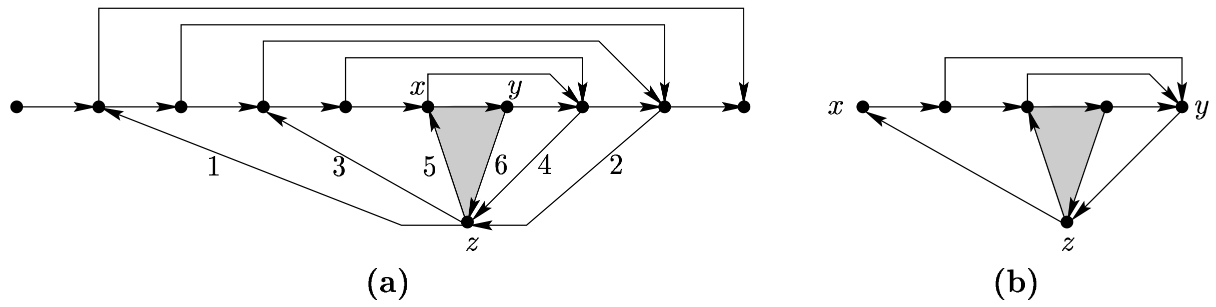

Figure 9: Construction of a graph with $3k + 2$ vertices and $7k - 1$ edges that requires $2k$ iterations. **(a)** The graph for $k = 3$. **(b)** The graph for higher k can be obtained by repeatedly replacing the shaded triangle with the depicted subgraph.

We conjecture that $O(n)$ iterations always suffice, but this remains an open problem. One would expect that in practice the number of iterations would be small, and hence this algorithm would terminate quickly. Another open problem is to develop an algorithm to find at once, without iterations, all edges that need to be removed to obtain a graph without s-or-t-useless edges.

3 Finding useless edges in planar graphs

Recall that an edge $e = (v, w)$ is *useful* if there are vertex-disjoint paths P_s from s to v and P_t from w to t. As mentioned in the introduction, it is NP-complete to test whether a given edge is useful in a general graph. By a result of Schrijver [Sch94] the problem is solvable in polynomial time in planar graphs. More precisely, Schrijver showed that the problem of finding k disjoint paths between k given pairs of sources and sinks in a planar directed graph is solvable in polynomial time. However, he did not study the precise time complexity of his algorithm, which seems to be high.

It should also be noted that if all sources and sinks of the k paths to be connected are on one face, then the problem can be solved in $O(n)$ time [RWW96]. However, in our problem the vertices s, v and t need not have any faces in common. Hence, none of the previous algorithms is applicable.

We present an algorithm finding in $O(n^2)$ time all useless edges in a planar graph. Our algorithm assumes that a fixed planar embedding is given such that t is on the outerface and the graph contains no clockwise cycles. Not all planar graphs have such an embedding, but after fixing arbitrary planar embedding with t on the outerface it is possible, in $O(n \log n)$ time, to modify graph so that maximum flow is not changed and all clockwise cycles are removed (see [KNK93]). Therefore in general our algorithm finds for any planar graph a planar graph with the same maximum flow not containing useless edges.[6]

This algorithm is unfortunately too slow to be relevant for maximum flow problems. Removing all useless edges takes $O(n^2)$ time, while a maximum flow problem can be solved in subquadratic time for planar graphs.[7] Still we believe that the problem is interesting enough in its own right to be worth studying.

The following notation will be useful later: Assume that $P = w_1, \ldots, w_k$ is a path. We denote by $P[w_i, w_j]$ $(i \leq j)$ the sub-path of P between vertices w_i and w_j. We denote by $P(w_i, w_j)$ $(i < j)$ the sub-path of P between vertices w_i and w_j excluding the endpoints. Note that this sub-path might consist of just one edge (if $i = j - 1$).

The crucial ingredient to our algorithm is the following observation, which holds even if the graph is not planar.

Lemma 10 *Let $e = (v, w)$ be an edge, let P_s be a simple path from s to v and let P_t be a simple path from w to t. If P_s and P_t have vertices in common, then there exists a simple directed cycle C that contains e such that some vertex $x \in C$ belongs to both P_s and P_t.*

Proof: Let x be the last vertex on P_s that also belongs to P_t, this exists by assumption. We claim that $C = P_s[x, v] \cup (v, w) \cup P_t[w, x]$ is the desired cycle. Clearly, C contains e and x. It is directed, because P_s is directed towards v and P_t is directed away from w. Finally, it is simple, because x was chosen as the last vertex of P_s that also belongs to P_t. $\square$

We will use the contrapositive of this observation: Assume that we have paths P_s and P_t, and they do not have a common vertex on any directed cycle containing e. Then P_s and P_t are vertex-disjoint.

Unfortunately, edge e may belong to many different directed cycles. But here is where planarity helps: we will show that we have to check only one directed cycle containing e, which is in some sense the "biggest" cycle containing e. To define this cycle, we use the concept of a *right-first search*: perform a depth-first search, and when choosing the next outgoing edge for the next step, take the counter-clockwise next edge after the edge from which we arrived.

[6]This is useful for flow algorithms that require input not containing useless edges or that might be simplified if it is known that such edges do not exist. One example of such algorithm is Weihe's algorithm [Wei97] that finds maximum flow in $O(n \log n)$ time, provided that the input graph is planar and fulfills three more assumptions, one of which is that it does not contain s-or-t-useless edges.

[7]See for example $O(n^{3/2} \log n)$ algorithm by Johnson and Venkatesan [JV82].

Definition 11 *The* rightmost cycle containing edge $e = (v, w)$ *is obtained as follows: perform a right-first search starting at w with the first outgoing edge (in counter-clockwise order) after e, and stop when we reach v for the first time. The rightmost cycle is then the path in the DFS-tree from w to v, together with edge e.*

Observe that the rightmost cycle containing e might not be defined. This happens if and only if there exists no directed cycle containing e. However, in this case by Lemma 10 edge e is useful if and only if it is s-and-t-reachable, which can be tested easily. So we assume for the remainder that the rightmost cycle C is well-defined, and it must be counter-clockwise because by assumption there are no clockwise cycles in the graph.

Enumerate C, starting at w, as $w = c_1, c_2, \ldots, c_k = v$. Cycle C defines a closed Jordan curve, and as such has an inside and an outside. We say that a vertex is *inside* (*outside*) C if it does not belong to C and is inside (outside) the Jordan curve defined by C. The following observation will be helpful later:

Lemma 12 *For any $1 \leq i < j \leq k$, any simple path P from c_i to c_j must either contain a vertex $\neq c_i, c_j$ in C or $P(c_i, c_j)$ must be inside C.*

Proof: Let P be a path between two vertices c_i, c_j with $i < j$. Let (c_i, x) be the first edge after c_i on P. If $x \in C$ then the lemma certainly holds, so assume $x \notin C$ and study the counter-clockwise order of edges around c_i. If this order is $(c_{i-1}, c_i), (c_i, c_{i+1}), (c_i, x)$, then (c_i, x) is inside C because C is counter-clockwise. If P ever reaches the outside of C, then by planarity it must contain some other vertex of C, which proves the claim. See the left picture of Figure 10.

If the order is $(c_{i-1}, c_i), (c_i, x), (c_i, c_{i+1})$, then during the right-first search, we explored (c_i, x) before exploring (c_i, c_{i+1}). Because we did not use (c_i, x) for C, this means that there is no path from x to w not using any of $c_1, \ldots, c_i$. Hence, path $P[x, c_j]$ must contain one of $c_1, \ldots, c_i$, and by simplicity of P, it must be one of $c_1, \ldots, c_{i-1}$. This proves the claim. See the right picture of Figure 10. $\square$

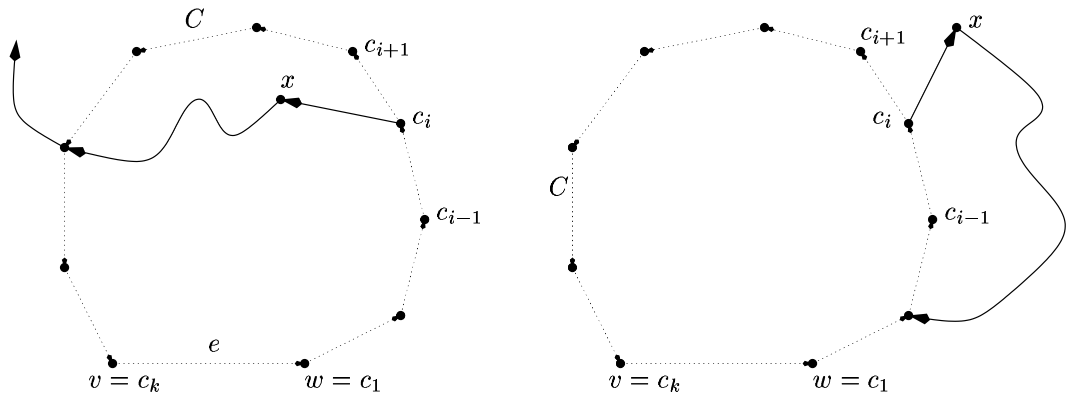

Figure 10: A path from c_i to c_j, $i < j$ must either be inside C or contain other vertices of C.

We may assume that s has no incoming and t has no outgoing edges (no such edge belongs to a simple path from s to t and therefore they are all useless and can be ignored). Therefore

$s, t \notin C$. On the other hand, both v and w belong to C. Hence, for any path from s to v there exists a first vertex $x \neq s$ on C; we mark vertex x as an *entrance*. Similarly, for any path from w to t there exists a last vertex $y \neq t$ on C; we mark y as an *exit*.

We show now that we can determine whether e is useful from the markings as entrances and exits on C alone.

Lemma 13 *If there is an exit c_i and an entrance c_j with $1 \leq i < j \leq k$, then e is useful.*

Proof: Let P'_s be a path from s to v that marked c_j as an entrance and let P'_t be a path from w to t that marked c_i as exit. Let $P_s = P'_s[s, c_j] \cup \{c_j, \ldots, c_k = v\}$ and let $P_t = \{w = c_1, \ldots, c_i\} \cup P'_t[c_i, t]$. Clearly, these are paths from s to v and from w to t; we only need to show that they are vertex-disjoint.

By definition of entrance and exit, $P'_s[s, c_j)$ and $P'_t(c_i, t]$ contain no vertices of C, so they are vertex-disjoint with $\{c_j, \ldots, c_k\}$ and $\{c_1, \ldots, c_i\}$, respectively. Also, by $i < j$ paths $\{c_1, \ldots, c_i\}$ and $\{c_j, \ldots, c_k\}$ are vertex-disjoint. So all we really need to show is that $P'_t(c_i, t]$ and $P'_s[s, c_j)$ are vertex-disjoint.

Assume to the contrary that they have vertices in common, and let x be the first vertex of $P'_t(c_i, t]$ that also belongs to $P'_s[s, c_j)$. See Figure 11. Let $P^* = P'_t(c_i, x] \cup P'_s[x, c_j)$. Then P^* is simple by choice of x. Also, it connects c_i and c_j, $i < j$, and it does not contain vertices of C. Hence, P^* and in particular therefore x is inside C by Lemma 12. But, vertex t is outside C because vertex t belongs to the outer-face and $t \notin C$. Hence $P'_t[x, t]$ connects a vertex inside C with a vertex outside C without containing vertices of C, a contradiction to planarity. $\square$

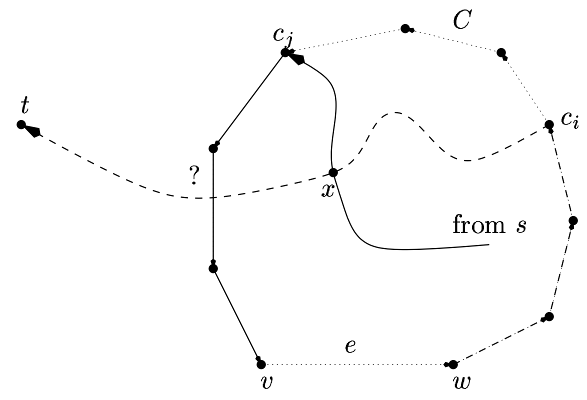

Figure 11: The remainders of the paths from s and to t are vertex-disjoint by planarity. Path P_s is marked with solid, path P_t is marked with dashed lines.

Now we show the reverse direction.

Lemma 14 *Let j_{max} be maximal such that $c_{j_{max}}$ is an entrance, and let i_{min} be minimal such that $c_{i_{min}}$ is an exit. If $j_{max} \leq i_{min}$, then e is useless.*

Proof: Assume to the contrary that e is useful, and let P_s and P_t be the vertex-disjoint paths from s to v and from w to t, respectively. Let c_{j^*} be the entrance defined by P_s, and let c_{i^*} be the exit defined by P_t. By assumption and vertex-disjointness, we have $j^* < i^*$.

Path P_t must use $c_1 = w$, so by vertex-disjointness $c_1 \notin P_s$. Let $j > 1$ be minimal such that $c_j \in P_s$; we have $1 < j \leq j^* < i^*$.

Path P_t uses the vertices c_1 and c_{i^*}, but not $c_j \in P_s$ by vertex-disjointness. Hence path P_t cannot "walk along" C to get from c_1 to c_{i^*} by $1 < j < i^*$, but must leave C somewhere and then get back to it. Let c_β be the first vertex of P_t that belongs to C and satisfies $j < \beta$, and let c_α be the last vertex before c_β (i.e., of $P_t(v, c_\beta)$) that belongs to C. See Figure 12. By definition of β we have $1 \leq \alpha < j < \beta$, and by definition of α path $P_t(c_\alpha, c_\beta)$ contains no vertices of C.

Path P_s uses the vertices c_j and $c_k = v$, but not c_β. Hence path P_s cannot "walk along" C to get from c_j to c_k by $j < \beta < k$, but must leave C somewhere and then get back to it at a vertex that is *not* between c_α and c_β. Let c_δ be the first vertex of P_s after c_j that belongs to C with $\delta \notin [\alpha, \beta]$. Actually, we know that $\delta > \beta$, because by $\alpha < j$ and definition of j P_s contains no vertex c_l with $l < \alpha$. Let c_γ be the last vertex of $P_s(s, c_\delta)$ that belongs to C. See Figure 12. By definition of β we have $\alpha < \gamma < \beta$, and by definition of γ path $P_s(c_\gamma, c_\delta)$ contains no vertices of C.

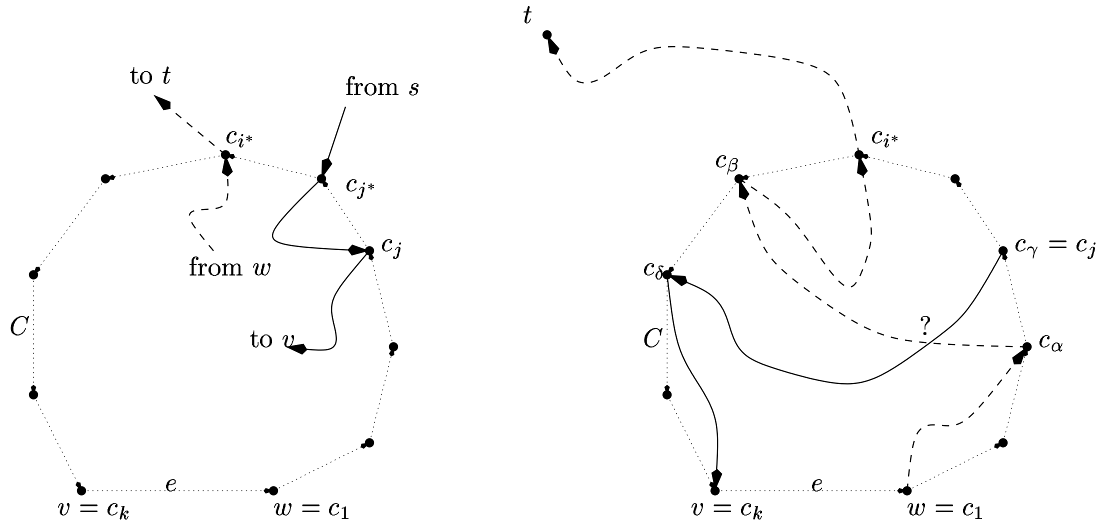

Figure 12: Illustration of the definition of $i^*, j^*, j, \alpha, \beta, \gamma$ and δ. Path P_s has solid, path P_t has dashed lines. We intentionally omitted s, as to show the independence of this proof with regards to whether s is inside or outside C.

Now $C \cup P_t[c_\alpha, c_\beta] \cup P_s[c_\gamma, c_\delta]$ forms a subdivision of K_4 by $\alpha < \gamma < \beta < \delta$. But by Lemma 12, $P_t(c_\alpha, c_\beta)$ and $P_s(c_\gamma, c_\delta)$ are both inside C because they contain no other vertex of C. Therefore, C is the outer-face of this subdivided K_4, which implies that K_4 is outer-planar. This is a contradiction. $\square$

Hence, if we are given a planar network with t on the outer-face and without clockwise cycles, then to find out whether $e = (v, w)$ is useful, we proceed as follows:

- Perform a right-first search from w, starting with the counter-clockwise first outgoing edge after e.

- When reaching v, stop and compute the rightmost cycle C.
- Perform a depth-first search from s. Every time a vertex on C is reached, mark it as entrance and retreat.
- Perform a depth-first search from t in the reversed graph. Every time a vertex on C is reached, mark it as exit and retreat.
- Scan the vertices of C in counter-clockwise order. If we find an exit strictly before an entrance, then e is useful, otherwise it is useless.

Clearly, this algorithm takes $O(m) = O(n)$ time.

Theorem 15 *Testing whether e is useless in a planar graph with t on the outerface and without clockwise cycles can be done in $O(n)$ time.*

4 Conclusion

In this paper, we studied how to simplify flow networks by detecting and deleting edges that are useless, i.e., that can be deleted without changing the maximum flow. Detecting all such edges is NP-complete. We first studied how to detect at least some useless edges. More precisely, we defined when an edge is s-useless, and showed how to find all s-useless edges in $O(m \log n)$ time. This improves on the simpler technique of removing all s-unreachable edges (which can be found in $O(m)$ time), because not all s-useless edges are s-unreachable.

We also studied other types of useless edges, in particular s-and-t-useless edges, and useless edges in planar graphs without clockwise cycles. While for both types we give algorithms to find such edges in polynomial time (more precisely, $O(m^2 \log n)$ and $O(n^2)$), these results are not completely satisfactory, because one would want to find such edges in time less than what is needed to compute a maximum flow per se. Thus, we leave the following open problems:

- We gave an example of a graph where computing s-or-t-useless edges takes $\Omega(n)$ rounds, hence our technique cannot be better than $O(mn \log n)$ time, which is too slow. Is there a "direct" approach to detect a subgraph without s-or-t-useless edges that has the same maximum flow?
- Can our insight into the structure of useless edges in planar graphs be used to detect *all* useless edges in a planar graph in time $O(n \log n)$ or even $O(n)$?
- Currently, our algorithm for useless edges in planar graphs works only if the planar graph has no clockwise cycles. While all clockwise cycles of a planar graph can be removed without changing the maximum flow, it would be more satisfactory to have an algorithm that does not rely on there being none. Does such an algorithm exist?

Finally, studying how to detect edges that are useless under a given set of capacities remains an open problem.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, 1993.

[BV99] Broňa Brejová and Tomáš Vinař. Weihe's algorithm for maximum flow in planar graph (project report), Fall 1999. University of Waterloo, Course CS760K.

[FHW80] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, February 1980.

[JV82] Donald B. Johnson and Shankar M. Venkatesan. Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time. In *Proceedings of the 20th Annual Allerton Conference on Communication,Control, and Computing*, pages 898–905, University of Illinois, Urbana-Champaign, 1982.

[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.

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

[RWW96] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. Linear-time algorithms for disjoint two-face paths problems in planar graphs. *International Journal of Foundations of Computer Science*, 7(2):95–110, June 1996.

[Sch94] Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal of Computing*, 23(4):780–788, August 1994.

[ST83] Daniel D. Sleator and Robert E. 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.