Simplifying Flow Networks

Therese C. Biedl*, Broňa Brejová**, and Tomáš Vinař***

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

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 [1]). 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.

The 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 (see [9] for an example of such algorithm). 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

* Supported by NSERC Research Grant.
** Supported by NSERC Research Grant OGP0046506 and ICR Doctoral Scholarship.
*** Supported by NSERC Research Grant OGP0046506.

M. Nielsen and B. Rovan (Eds.): MFCS 2000, LNCS 1893, pp. 192–201, 2000.

the actual capacities in the network. Thus, a change in capacities would not affect the usefulness of an edge.[1]

The above definition of a useful edge is hard to verify. As a first step, we therefore develop equivalent characterization which uses directed paths.[2]

Lemma 1. *The following conditions are equivalent:*

1. *Edge $e = (v, w)$ is useful.*
2. *There exists a simple path P from s to t that uses e.*
3. *There exist vertex-disjoint paths P_s from s to v and P_t from w to t.*

Proof Sketch. Assume that (1) holds. Let c be capacities such that any maximum flow contains e. By the flow decomposition theorem there exists a maximum flow f that can be decomposed into flows along simple paths from s to t. Since f uses e, (2) holds.

To prove that (2) implies (1) we set all capacities on P to 1, all other capacities to 0. The equivalence of (2) and (3) is obvious by adding/deleting edge e. □

This implies that testing whether an edge is useless is NP-complete, by reducing it to the problem of finding two disjoint paths between two given sources and sinks [3]. Thus we relax the problem 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.
- 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. For space reasons many details have been left out; a full-length version can be found in [2].

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.

[1] 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, and we will not pursue such an alternative definition.

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

- 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. 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.

Edges that are s-reachable can be easily detected using a directed search from s in $O(m + n)$ time (similarly we can also detect all t-reachable edges). However, the other introduced concepts cannot be handled so easily. We study for each of these concepts how to detect such edges in the following subsections.

2.1 *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. In particular, 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. This observation can be used to determine s-useless edges in $O(mn)$ time (perform n times a depth-first search starting in s, each time with one vertex removed from the graph). 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.

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. 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* (included in the DFS-tree), *back edges* (leading to an ancestor), *forward edges* (leading to a descendant that is not a child), and *cross edges* (leading to a vertex in a different subtree). By properties of a depth-first search, cross edges always lead to a vertex with a smaller DFS-number (see for example [1]).

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 inbetween. The precise definition is as follows (see Fig. 1a for an illustration).

Definition 2. *If u and v are two vertices, and u is an ancestor*[3] *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 $T(u, v) \cup \{v\}$.*

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

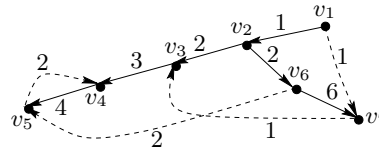

Fig. 1. The DFS-tree is shown with solid lines. Every edge e is labeled by *detour*(e).

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 $T(u, v)$.*[4] *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. 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.*

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.

The Second Traversal. We compute detours by scanning the vertices in reverse DFS-order. To compute *detour*(v) we need to compute *detour*(e) for all edges e incoming to v. The following lemma formulates the core observation used in the algorithm.

Lemma 2. *Let $e = (w, v)$ be an edge. If e is a tree edge or a forward edge then detour$(e) = num(w)$. If e is a cross edge or a backward edge and a is 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 Sketch. Let e be a tree edge or a forward edge. Obviously e is itself a detour and therefore $detour(e) = num(w)$.

Assume that e is a cross or backward edge. Let $d = \min_{u \in T(a,w]} detour(u)$, let u be a vertex in $T(a, w)$ that achieves the minimum, and let Q be a detour from v_d to u (see the left picture of Fig. 2). It is possible to show that v_d is an ancestor of a and that $Q - \{v_d\}$ does not contain a vertex b with $num(b) < num(u)$. The latter claim follows from the properties of DFS-tree. Let P be a path from v_d to v starting with Q from v_d to u, then following tree-edges from u to w, and ending with edge e. Since all vertices $b \in T(v_d, v]$ have $num(b) < num(w)$, path P is indeed a detour for e. This gives us $detour(e) \leq d$.

On the other hand, let P be the detour that ends with edge $e = (w, v)$ that starts in a vertex x such that $num(x) = detour(e)$ (see the right picture of Fig. 2). Observe that x is an ancestor of a. Let y be the first vertex on P that belongs to $T(a, w]$. Then the part of P from x to y is a detour for y and therefore $d \leq detour(y) \leq detour(e)$. □

[4] We allow $u = v$, in case of which any path from v to v is a detour.

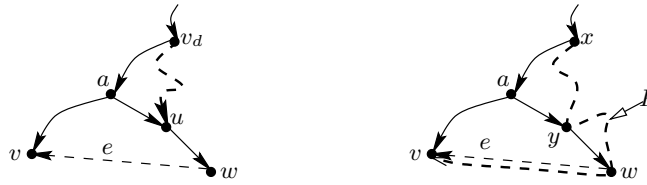

Fig. 2. Computation of *detour*(e) for a cross edge. The case of a back edge is similar.

Using this lemma, we can compute the detour-values during the second traversal. We use the dynamic trees (DT) data structure of [8], which we initialize to set of isolated vertices $\{v_1, v_2, \ldots, v_n\}$ in $O(n)$ time.

Now we process the vertices in reverse DFS-order (i.e., from v_n to v_1). After we have computed *detour*(v_j) $(j > 1)$, we update DT by linking v_j to its parent in T with an edge of weight *detour*(v_j). This can be done in $O(\log n)$ time [8]. Therefore DT contains in each step a subset of edges in T so that after processing v_j it contains all tree edges with at least one endpoint from $\{v_j, \ldots, v_n\}$. In particular, this means that each vertex v_i for $i < j$ is a root of a tree in DT.

To compute *detour*(v_j), we compute *detour*(e) for all incoming edges e of v_j and take the minimum. The value *detour*(e) is computed using Lemma 2. The value for tree and forward edges can be determined in straightforward way in $O(1)$ time. For back and cross edges we need to compute $\min_{u \in T(a,w]}$ *detour*(u), where a is the nearest common ancestor of v_j and w.

Observe that *num*$(a) \leq$ *num*(v_j) and therefore a is the root of the tree in DT containing w. Therefore *detour*(e) is the minimum value of an edge in DT on the path from w to the root of the tree containing w in DT (this root is a). Such a minimum can be computed in $O(\log n)$ time in dynamic trees [8]. Hence we can determine *detour*(e) in $O(\log n)$ time. Altogether, this traversal takes $O(m \log n)$ time.

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, whose proof is omitted for space reasons.

Lemma 3. *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).

To actually determine the s-useful edges, we use interval trees [6], where the endpoints of intervals are in $\{1, 2, \ldots, n\}$. The interval tree is initialized as an empty tree in $O(n)$ time. 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 $(\textit{detour}(u), \textit{num}(u))$ for all ancestors u of v. We add interval $(\textit{detour}(v), \textit{num}(v))$ into the tree. When we retreat from v, we delete $(\textit{detour}(v), \textit{num}(v))$ from the interval tree. Inserting and deleting an interval takes $O(\log n)$ time per vertex [6].

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 there is an interval (a, b) in the tree with $a < num(w) < b$. This can be tested in $O(\log n)$ time per back edge [6], and hence $O(m \log n)$ time total.

Theorem 1. *All s-useless edges can be found in $O(m \log n)$ time.*

2.2 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 4. *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, because 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. 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 (see Fig. 3).

We conjecture that $O(n)$ iterations always suffice, but this remains an open problem. One would expect that in practice the number of iterations is small, and hence this algorithm would terminate quickly. Another open problem is to develop a more efficient algorithm to find 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 [7] the problem is solvable in polynomial time in planar graphs. However, he did not study the precise time complexity of his algorithm, which seems to be high.

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 for fixed capacities and fixed planar embedding with t on the outerface, it is possible, in $O(n \log n)$ time, to modify graph so

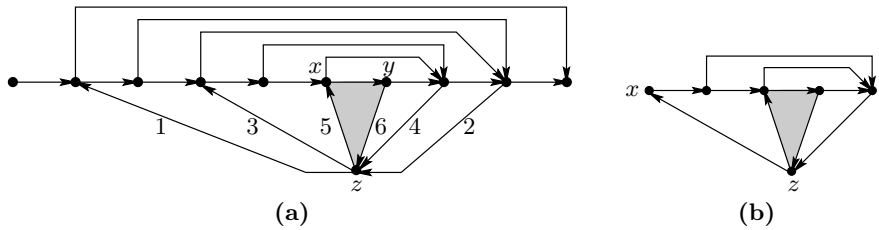

Fig. 3. Construction of a graph with $3k + 2$ vertices and $7k - 1$ edged that requires $\Omega(n)$ iterations. In the first iteration, edge 1 is detected as s-useless and deleted, which makes edge 2 t-useless. Deleting edge 2 makes edge 3 s-useless. This continues, and one can show that we need $2k = \Omega(n)$ 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.

that the maximum flow is not changed and all clockwise cycles are removed (see [5]). Therefore in general our algorithm finds for any planar network a planar network with the same maximum flow not containing useless edges.

The algorithm is unfortunately too slow to be relevant for maximum flow problems.[5] 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 5. *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 Sketch. Let x be the last vertex of P_s that belongs to P_t. Then $P_s[x, v] \cup (v, w) \cup P_t[w, x]$ forms the desired cycle. □

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.

Using planarity we will show that we need to examine only the "rightmost" 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 outgoing edge for the next step, take the counter-clockwise next edge after the edge from which we arrived.

The *rightmost cycle containing edge* $e = (v, w)$ consists of edge e and the path from w to v in the DFS-tree constructed by the right-first search starting in w with the first outgoing edge (in counter-clockwise order) after e.

[5] A maximum flow problem can be solved in subquadratic time for planar graphs. See for example the $O(n^{3/2} \log n)$ time algorithm by Johnson and Venkatesan [4].

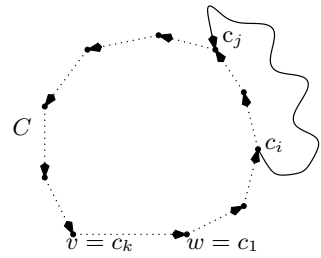

Fig. 4. A path from c_i to c_j, $i < j$ must either be inside C or contain other vertices of C. Otherwise C would not be the rightmost cycle containing edge (v, w).

Observe that the rightmost cycle containing e is not defined if and only if there exists no directed cycle containing e. However, in this case by Lemma 5 edge e is useful if and only if it is s-reachable 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 6. *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 Sketch. If there were a path that satisfies neither condition, then using it we could find a cycle containing e that is "more right" than C (see Fig. 4). □

We may assume that s has no incoming and t has no outgoing edges (such edges would be useless). 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*.

The following two lemmas show that we can determine whether e is useful from the markings as entrances and exits on C alone.

Lemma 7. *If there is an exit c_i and an entrance c_j with $1 \leq i < j \leq k$, then e is useful.*

Proof Sketch. 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, and using Lemma 6, one can argue that they are vertex-disjoint. See also Fig. 5a. □

Now we show that the reverse of Lemma 7 also holds.

Lemma 8. *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.*

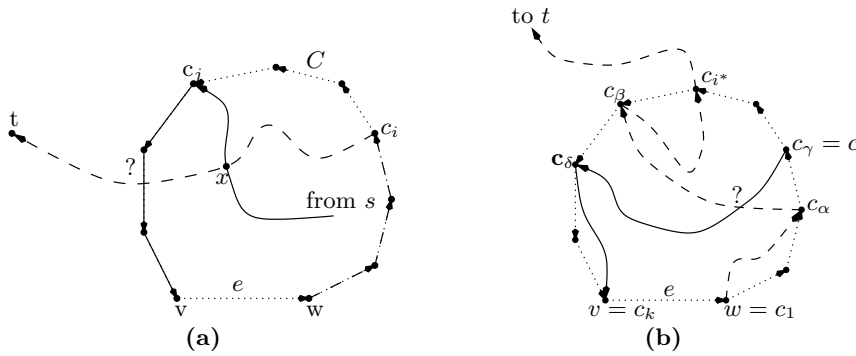

Fig. 5. **(a)** Paths P_s (solid) and P_t (dashed) are vertex disjoint, for if they intersect, it would have to be at a vertex x inside C, which contradicts planarity and the fact that t is outside C. **(b)** Illustration of the definition of $i^*, j^*, j, \alpha, \beta, \gamma$ and δ. Path P_s has solid, path P_t has dashed lines.

Proof Sketch. 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 $j^* < i^*$.

Let $j > 1$ be minimal such that $c_j \in P_s$. 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 of $P_t(v, c_\beta)$ that belongs to C. Similarly, let c_δ be the first vertex of P_s after c_j that belongs to C with $\delta \notin [\alpha, \beta]$. Let c_γ be the last vertex of $P_s(s, c_\delta)$ that belongs to C.

By careful observation it is possible to show that $j, \alpha, \beta, \gamma, \delta$ are well-defined and $\alpha < \gamma < \beta < \delta$ (see Fig. 5b).

Now $C \cup P_t[c_\alpha, c_\beta] \cup P_s[c_\gamma, c_\delta]$ forms a subdivision of K_4. But by Lemma 6, $P_t(c_\alpha, c_\beta)$ 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. □

Computing C and marking exits and entrances can be done with directed searches in $O(n + m)$ time, so the following result holds:

Theorem 2. *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.

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. Is there an efficient algorithm which works without this assumption?
- The problem of finding maximum flow in directed planar graphs in $O(n \log n)$ time seemed to be solved by [9]. However this algorithm needs s-or-t useless edges to be removed in preprocessing and the author does not provide a solution for this problem. Therefore we consider the following question still open: Is it possible to find a maximum flow in directed planar graphs in $O(n \log n)$ time?

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.
2. Therese Biedl, Broňa Brejová, and Tomáš Vinař. Simplifying flow networks. Technical Report CS-2000-07, Department of Computer Science, University of Waterloo, 2000.
3. Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, February 1980.
4. 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.
5. 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.
6. Franco P. Preparata and Micheal I. Shamos. *Computational Geometry: An Introduction*. Springer–Verlag, 1985.
7. Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal of Computing*, 23(4):780–788, August 1994.
8. 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.
9. 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.