Improved Algorithms for Min Cut and Max Flow in Planar Graphs

Robin Münstermann

August 8, 2018

Abstract

This paper deals with the problem of min cut and max flow computation in planar graphs.

There are many algorithms for the min cut and max flow problem in general graphs, but these problems can be solved even faster in planar graphs than in general graphs. We will have a closer look on computing a min cut and a max flow in undirected planar graphs. This paper will start by providing an overview over some definitions, which then will be used to introduce Reif's algorithm as well as an improved version, the so called faster min *st*-cut algorithm. Reif's algorithm finds a min *st*-cut by finding a min *st*-separating cycle in the dual graph, because solving this problem is equivalent. The faster min *st*-cut algorithm improves Reif's algorithm by using a two phase approach, firstly running a coarse version of Reif's algorithm and secondly solving the min *st*-cut problem exactly. With this algorithm a min *st*-cut can be calculated in $O(n \log\log n)$ and then a max *st*-flow can be computed in $O(n)$ starting from this. Afterwards an algorithm for min cut and max flow in directed planar graphs and also the multiple-sink, multiple-source problem in planar graphs is presented briefly.

Contents

1 Introduction

In planar graphs many problems can be computed faster than in general graphs. The same holds true for computing a min cut and a max flow in planar Graphs. Min cut and max flow problems on planar graphs are appearing naturally in many practical problems and also theoretical problems can be solved faster with the algorithms presented in this paper [7].
When computing a max st-flow in a directed graph $G = (V, E)$, every edge e has a capacity c_e. In a st-flow every edge e is assigned a value f_e, so that $0 \leq f_e \leq c_e$ holds true for every edge $e \in E$ and also for every vertex $v \in V \backslash \{s, t\}$ the total incoming flow $\sum_{w \in V \backslash \{v\}} f_{(w,v)}$ equals the total outgoing flow $\sum_{w \in V \backslash \{v\}} f_{(v,w)}$. A max st-flow maximizes the total outgoing flow minus the incoming flow of vertex s [1]. This problem is defined analogously in undirected graphs. The min st-cut problem can be solved by finding a set of edges, which divides G into two graphs with s in one graph and t in the other one, while the total capacity of these edges is equal or less than the capacity of any other set of edges dividing G into two graphs as described. To determine the value of a max flow, also a min st-cut can be computed, because a min st-cut has the same value as a max st-flow.
Finding a min cut and a max flow is an important topic for graph algorithms. Ford and Fulkerson, Edmonds and Karp and many others gave algorithms to solve this problem on general graphs. Ford and Fulkerson also presented an algorithm for max st-flows in (s, t)-planar graphs, graphs where s and t are on the same face. This was later improved to a runtime of $O(n)$. Itai and Shiloach generalized this algorithm to planar graphs and they gave an algorithm taking $O(n^2 \log n)$ [6]. Reif's algorithm improved this even further, which is presented in this paper.
The approach of the presented algorithms is to find the min st-cut first and with an algorithm of Hassin and Johnson the max st-flow can be computed in $O(n)$.
Firstly some definitions are given to understand the algorithms, which are presented afterwards. The first algorithm presented is Reif's algorithm, which can solve the min st-cut problem. The second algorithm, faster min st-cut algorithm, speeds up Reif's algorithm by using two phases. In the first phase a coarse version of Reif's algorithm is used. Only a subset of the vertices of a given Graph G is considered, what leads to a faster algorithm, which does not calculate the min st-cut, but divides G in order to use these parts of G in the second phase to calculate the exact min st-cut, while having the advantage that only a part of G needs to be considered in each computation. Afterwards it is shown, that with a given min st-cut solving the max st-flow problem in G takes $O(n)$. At the end other algorithms for max flow problems in planar graphs are briefly presented.

2 Definitions

2.1 Face

If a planar graph is drawn on a paper or in the plane without any edges crossing each other, the edges and vertices create regions. These regions are called faces [10]. Also there will be at least one region, which is not bounded by any vertices and edges. This region is also a face and it is called outer face (see Figure 1).

2.2 Piece

Given a graph G a piece P of G is defined as the subgraph of G induced by a subset of E [7]. All vertices of P, which are not incident to a vertex not in P, are called interior vertices. Vertices which are incident to a vertex not in P are called boundary vertices. Pieces of G

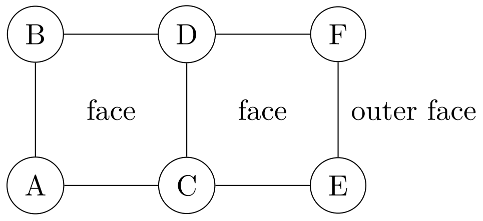

Figure 1: Example for faces in a planar graph.

will be found using r-division. The r-division does not have normal, disjunct subgraphs as output, but pieces which share their boundary vertices. For example a boundary vertex v of piece a can also be a boundary of another piece b.

2.3 r-Division

To find pieces in a graph a modified algorithm of Frederickson is used. Frederickson gave an algorithm, which divides a given graph G into $O(n/r)$ pieces for any parameter $r \in (0, n)$ [3]. The output of this algorithms is pieces with each having $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices. Fredericksons algorithm has $O(n \log r + (n/\sqrt{r}) \log n)$ running time. The parameter r will later be chosen so, that r is suitable for the runtime of the faster min st-cut algorithm. But a stronger result is needed for the faster min st-cut algorithm. This is given in theorem 2.1. Therefore a hole is defined as a bounded face of P, which is not a face of G. A hole can exist, if a piece contains another piece inside of it. This will increase the complexity of the algorithm, so we do not want to find pieces with many holes. When r-division is mentioned it is defined as followed.

Theorem 2.1. *For a plane n-vertex graph, an r-division in which each piece has $O(1)$ holes can be found in $O(n \log r + (n/\sqrt{r}) \log n)$.*

The goal of the r-division is to divide G and using dense distance graphs to speed up the first phase of the faster min st-cut algorithm.

2.4 Algorithm of Frederickson

The algorithm of Frederickson divides G into $O(n/r)$ pieces. Only the idea of the algorithm will be explained. The algorithm is divided into two phases: The first one takes $O(n/\sqrt{r})$ and the second phase $O(n \log r)$. This gives us a runtime of $O(n \log r + (n/\sqrt{r}) \log n)$.
In the first phase we find a spanning forest in G with trees of size $\Theta(\sqrt{r})$. Then we contract these trees to a graph G' and try to find pieces in G' recursively. This gives us $O(n/r^{3/2})$ pieces, therefore a second phase transforms G' back to G and divides pieces more, so that finally there are only $O(n/r)$ pieces. For a detailed description of the algorithm see [3].
This algorithm gives us a r-division in $O(n \log r + (n/\sqrt{r}) \log n)$, but does not ensure that there are only $O(1)$ holes. We need a constant number of holes to achieve our runtime of $O(n \log \log n)$ in the faster min st-cut algorithm. Hence the recursive algorithm is a bit modified by using Miller's cycle separator theorem [9]. This theorem ensures, that if a piece P is split into two subpieces, where the number of holes of one piece P' has increased, we split this piece again, so that the number of each of the subpieces of P' have again maximum the number of holes of P by using an idea of Fakcharoenphol and Rao [7]. This modification of the algorithm of Frederickson ensures $O(1)$ holes for each pieces resulting from the r-division, so that finding shortest paths can be prepared for each piece so the faster min st-cut algorithm is able to have its fast runtime.

2.5 Dense Distance Graph

Given these pieces, which can be computed with the algorithm of Frederickson, preprocessing of the pieces can be done, in order to compute shortest paths fast. Given a piece P from every boundary vertex to every other boundary vertex all shortest paths are computed. This

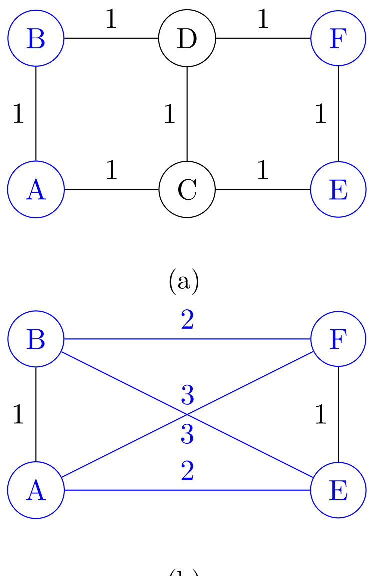

Figure 2: (a) Piece with A, B, F, E boundary vertices. In (b) the resulting dense distance graph is shown.

preprocessing results in a dense distance graph. This graph is a graph with weight of an edge (v, w) equal to the shortest path from a boundary vertex v to a boundary vertex w (see Figure 2).
Klein's algorithm can compute a dense distanse graph for one piece P in $O(r \log r)$. By given r-division dense distance graphs of $O(n/r)$ pieces have to be calculated. Therefore this step takes in total $O(n/r) \cdot O(r \log(r)) = O(n \log(r))$ time [7].

2.6 Fast Dijkstra

If the graph is divided in dense distance graphs, what is obtained by the r-division and Klein's algorithm, shortest path can be computed faster, because we do not have to take every vertex of G into account, but only the boundary vertices. Let G be a graph consisting of dense distance graphs only and b be the total number of boundary vertices of G. A shortest path computed with fast Dijkstra, which is essentially Dijkstra considering only boundary vertices, takes $O(b \log^2(n))$ time. Because of the r-division and Klein's algorithm, we can transform a given graph, where the min *st*-cut problem should be solved in, in a graph with the structure of G and then b equals $O(\sqrt{r})$. So for our problem fast Dijkstra runs in $O(\sqrt{r} \log^2 n)$.

2.7 Dual Graph

Both algorithms, Reif's algorithm and the faster min st-cut algorithm are not operating on the given graph G, but they first transform the given graph into a dual graph G'. For the dual graph we first determine all faces of G. Then all the faces become vertices of the dual graph. Every face is separated from another face by one or more edges of the primal graph and for every edge of these edges a edge is added to the dual graph with vertices corresponding to faces separated by the edge of the primal graph (see also figure 3) [7]. If a edge of the primal graph has a weight, then this weight will also be the weight of the corresponding edge in the dual graph. Moreover if from the dual graph again the dual graph is computed, the result is the primal graph again, so we can always revert this operation and figure out which edges of the primal and dual graph and which faces and vertices of the primal and dual graph belong together.

3 Reif's Algorithm

Reif's algorithm solves the min st-cut problem for a given Graph G' by computing a min st-separating cycle of the dual Graph G of G'. A min st-separating cycle is a simple cycle, which contains a face s or t inside of the cycle and the other face is outside of the cycle, so that s and t will be divided or separated by this cycle [7]. Now the min st-cut will be found by using

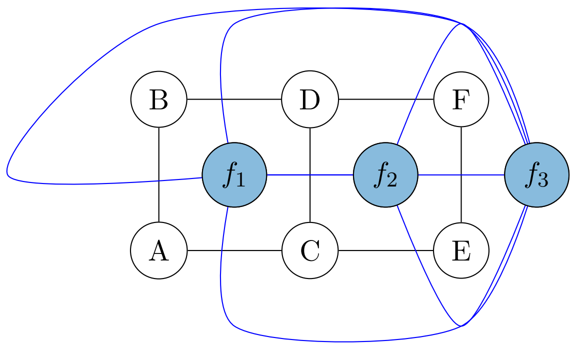

Figure 3: Dual graph of the graph given in figure 1. The vertices f_1, f_2, f_3 are the faces of the primal graph.

the following Lemma, which was proven by Itai and Shiloach [6]:

Lemma 3.1. *A min st-separating cycle of a dual graph G defines a min st-cut of the corresponding primal graph G'.*

This works basically because a min *st*-cut defines the bottleneck of the max *st*-flow. We find this bottleneck by calculating a cycle in the dual graph, which includes edges with the least weight possible, while dividing s and t.
From this follows the idea of Reif's algorithm, which works on the dual graph G to compute the min *st*-separating cycle. The min *st*-separating cycle has to include at least one vertex from a path from s to t. The idea is to take the shortest path as this path. Then we compute all min *st*-separating cycles, which include each a vertex of this path, so that we find the min *st*-separating cycle of those cycles. But first we modify the graph in a way, that we can search for shortest paths instead of min *st*-separating cycles, which include each one vertex of the path from s to t, to speed up the algorithm.
First we compute the shortest path π from a vertex p_1 of face s to a vertex $p_{|\pi|}$ of face t. Then an incision along π is made: All edges E_r outgoing on the right of π are deleted. Now an copy of π called π' is inserted and E_r is inserted back in, but instead of $p_i \in \pi$ the corresponding node ${p_i}' \in \pi'$ is set. Next a shortest path ϱ from $p_{|\pi|/2}$ to $p_{|\pi'|/2}$ is computed. The result of these operations can look like it is shown in Figure 4. The incision and the shortest path divide the graph into two subgraphs, where we search recursively for a shortest path. We always take $p_i \in \pi$, which divides the subgraph in the middle, resulting in shortest path from $p_i \in \pi$ to the corresponding $p_i' \in \pi'$ for every $p_i \in \pi$. The goal of this algorithm is to find the min *st*-separating cycle, which has to include at least one $p_i \in \pi$ in order to separate faces s and t. We already computed the shortest path for all $p_i \in \pi$, so we can just pick the shortest path of all these computed shortest path, which all define a min cycle including one p_i and separating s and t and we get a min *st*-separating cycle. With this cycle found also the min *st*-cut is found.
The runtime will only be covered briefly here, because in the faster min *st*-cut algorithm the runtime will be covered in detail and this algorithm has a faster runtime. Reif's algorithm runs in $O(n \log^2 n)$ by using normal Dijkstra to find a shortest path. If a faster shortest path algorithm, like for example Frederickson's algorithm is used, the runtime is improved to $O(n \log n)$ [3]. For 25 years this was the fastest algorithm for this problem, but the faster min *st*-cut algorithm improves this even more.

4 Faster Min *st*-Cut Algorithm

The faster min *st*-cut algorithm of a given graph G' computes, like Reif's algorithm, a min *st*-cut by computing a min *st*-separating cycle of the dual graph G of G', but only takes $O(n \log \log n)$ by using two phases.
The first phase roughly computes the shortest paths for those $p_i \in \pi$, which are boundary ver-

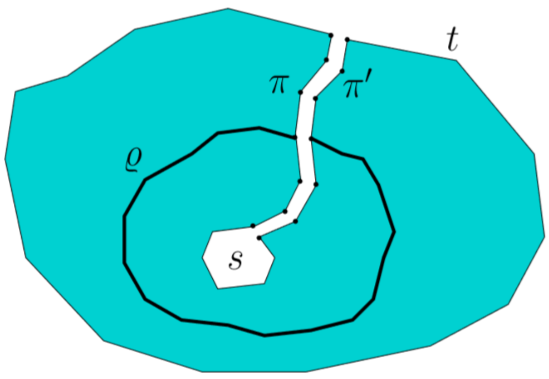

Figure 4: Example how a Graph can look like after the third step of Reif's algorithm.

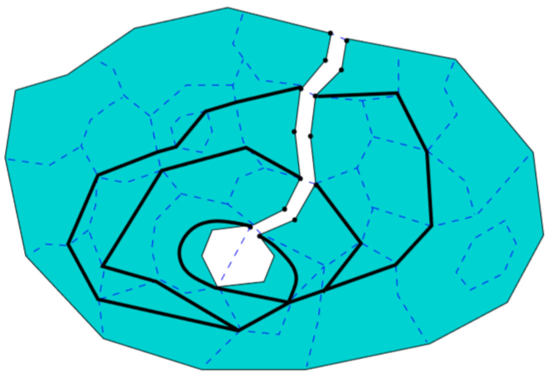

Figure 5: Illustration of a graph after the first phase of the faster min st-cut algorithm.

tices, while taking advantage of dense distance graphs and only takes boundary vertices into account. Just Reif's algorithm is executed on a graph, which only consist of boundary vertices and the edges of dense distance graphs. This leads to a faster runtime, because this phase works on a smaller graph. In figure 6 the result of the first phase is illustrated. The in the first phase found st-separating cycles split G into subgraphs for the second phase, where the not yet computed cycles are calculated. Each subgraph can be taken separately into account, because a st-separating cycle with start and end vertex p in a subgraph H, which just uses vertices of H cannot be longer than another st-separating cycle with the same start and end vertex p, which can use all vertices of G. The shortest paths for the remaining, in the first phase not considered, interior vertices can be calculated by using this characteristic.
In the second phase we do this exact calculation of st-separating cycles and therefore we need to compute a shortest path for interior vertices of every piece P on a subpath π' of π on a subgraph H induced by the shortest paths found in the first phase.

Lemma 4.1. *Let π' be a subpath of π of length $O(\log^c n)$ with constant c. The subproblem, finding shortest paths from every $p \in \pi'$ to the corresponding vertex of the incision on a subgraph H can be solved in $O(|H| \log\log n)$.*

Proof. A piece has $O(r)$ vertices, because the r-division has this as a property. If we choose $r = \log^c n$ we get $|\pi'| = O(r) = O(\log^c n)$, because if a piece has vertices limited by $O(\log^c n)$, a path inside a piece cannot be longer than all vertices of this piece contained once. Recursion depth of Reif's algorithm is $O(\log(\log^c n)) = O(\log\log n)$. So using a shortest path algorithm [5] we get a total runtime of $O(|H| \log\log n)$. □

Now we will have a closer look on both phases and why problems, where vertices are considered more than once, do not affect the runtime of $O(n \log\log n)$.

4.1 First Phase

Given a graph G' firstly the dual graph G of G' is computed. Then an r-division is made on this graph and on the produced pieces dense distance graphs are calculated.
In this first phase Reif's algorithm is executed not on a G, but a modified Graph only consisting of boundary vertices found with r-division

and edges resulting from dense distance graphs on the pieces. Or in other words we are using fast Dijkstra to compute shortest path for the boundary vertices of π.

It takes $O(n)$ to create the dual graph G of G'. For the r-division $r = \log^6 n$ is chosen to attain a total runtime of this algorithm of $O(n \log\log n)$. Following Theorem 2.1 computing the r-division takes $O(n \log r + (n/\sqrt{r}) \log n) = O(6n \log\log n + (n/log^6 n) \log n) = O(n \log\log n)$. Having a division of G in pieces, there is also a problem called cutting pieces open. This happens when an incision is made in a piece. That means if π is cutting through a piece P, for example $v \in P$, $v \in \pi$ and v is interior vertex of P, there will be a boundary vertex $x \in \pi$ and a boundary vertex $x' \in \pi$, which are divided by the incision. Then we just choose x and x' to be still a part of P. They belong to the same piece P. Although these nodes are doubled during the incision, this ensures a total number of $O(\sqrt{r})$ boundary vertices in each piece. Dense distance graphs can be computed in $O(n \log(r))$ as it was shown before. With the chosen r, this yields in a runtime of $O(n \log r) = O(n \log\log^6 n) = O(n \log\log n)$. Fast Dijkstra can now be applied on the graph taking $O((n/\sqrt{r}) \log^2 n) = O(n/\log n)$ time. Recursion depth is $O(n \log n)$, what leads to a runtime of $O(\log n(n/\log n)) = O(n)$ for computing shortest path from every boundary vertex $p \in \pi$ to the corresponding boundary vertex $p' \in \pi'$.

Now given shortest paths, these paths can share vertices and we have to ensure, that these shared vertices are not too many. In Figure 6 this problem is illustrated. ϱ_1 and ϱ_3 are sharing some nodes, but if every vertex of G is considered too often our runtime will increase.

Although shortest path are not unique, we can assume that there will be an entry vertex q_i and an exit vertex q'_i, where ϱ_1 and ϱ_3 share the same path and therefore some same vertices. If now starting from $p_3 \in \pi$ the shortest path reaches q_i, we start searching q'_i from $p'_3 \in \pi'$. All vertices in ϱ_3 between q_i and q'_i will now be substituted by an edge π_1 from q_i to q'_i with weight corresponding to the path between q_i to q'_i. Also there will be a path from p'_3 to p_3, which maximal has one entry and one exit vertex on ϱ_1. ϱ_3 can share nodes also with ϱ_2, but cannot share nodes with any other shortest path without the nodes being included in ϱ_1 or in ϱ_3, because the adjacent shortest path limit the area for shortest paths. If ϱ_3 shares nodes with ϱ_2, we handle this the same way with a new edge π_2. Now we can assume that we have at most 4 entry and exit nodes. All other nodes $\varrho_3 \backslash (\pi'_1 \cup \pi'_2)$ are only contained in ϱ_3 and are not included in any other shortest path. It follows, that the size of a subgraph induced by a shortest path is limited by a constant multiplied with the number of boundary vertices plus the number of dividing the graph in subgraphs through shortest paths.

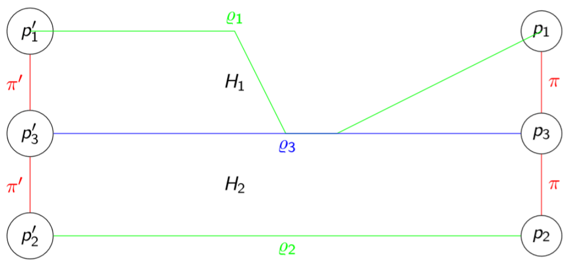

Figure 6: Problem of shortest path sharing nodes.

4.2 Second Phase

In the second phase we take the divided graph by the shortest paths of the first phase and by using Lemma 4.1 we can compute the remaining shortest path of the interior vertices of π and π' in $O(n \log\log n)$ time, because we chose our $c = 6$ while choosing $r = \log^6 n$.

When talking about subgraphs induced by

shortest path of the first step, we are talking about an implicit representation of shortest path by using preprocessed paths of dense distance graphs. Transforming an implicit representation in an explicit representation takes $O(n)$ time, because a planar graph G has constant degree and Klein's algorithm can compute the explicit representation in linear time depending on the length of the path [8].
Nevertheless we have to face a similar problem as in the first phase: Paths should not share too many boundary vertices in order to achieve an $O(n \log \log n)$ runtime. A shortest path ϱ_2 from $p_2 \in \pi$ to $p_2' \in \pi'$ can be limited again by the adjacent shortest paths ϱ_1 and ϱ_3, but we cannot handle this by using an edge as in phase one again, because we want to have the shortest paths as a result. Therefore another approach is chosen. If ϱ_2 and ϱ_1 share the same subpath, we have again one entry v_1 and one exit point v_2. A path can be represented as a concatenation of different paths. Let ϱ_{v_1,v_2} be the path between v_1 and v_2, ϱ_{p_2,v_1} a shortest path between p_2 and v_1, $\varrho_{v_2,p_2'}$ a shortest path between v_2 and p_2'. Then ϱ_2 can be represented as a concatenation of ϱ_{p_2,v_1}, ϱ_{v_1,v_2} and $\varrho_{v_2,p_2'}$. ϱ_1 can be represented analogously. Maximum number of doubled considered vertices, the entry and exit points of the adjacent shortest paths, is four for each shortest path. With this method with can ensure a runtime of $O(n)$ for explicit representation.
With this algorithm we can solve the min st-cut problem in undirected planar graphs in $O(n \log \log n)$ time.

5 Faster Max st-Flow

With the faster min st-cut algorithm we can compute the min st-cut in $O(n \log \log n)$. Knowing the value of the min st-cut respectively the value of the max st-flow, we can now compute a max st-flow in $O(n)$. This is done with an algorithm of Hassin and Johnson [4] using a faster shortest path algorithm of [5]. Therefore first the faster min st-cut algorithm can be used and then the algorithm of Hassin and Johnson to solve the max st-flow problem in undirected planar graphs in $O(n \log \log n)$.

6 Outlook

After explaining the algorithms for solving the min cut and max flow problem in undirected planar graphs in detail, a brief outlook to other max flow and min cut problems will be given.

6.1 Max st-Flow In Directed Planar Graphs

When looking at this problem in directed planar graphs, it is not possible to use Reif's algorithm, because Reif's algorithms uses heavily the graph-characteristic of undirected edges. Therefore Glencora Borradaile and Phillip Klein presented an algorithm for max st-flow [1], which takes $O(n \log n)$. This algorithm works in two steps. The first step is a preprocessing step, where single-source shortest-path distances in the dual are computed. In the second step the algorithm iterative saturates the leftmost path from source s to sink t of the residual graph. This algorithm, presented in 2009, was the fastest algorithm for this problem.

6.2 Multiple-Source Multiple-Sink Max Flow in Directed Planar Graphs

The problem computing a max flow in a graph with multiple sources and multiple sinks in a directed planar graph G given a algorithm for single-source single-sink max flow on directed planar graphs is not trivial like in general graphs. In general graphs we can solve this problem by manipulating G. We can introduce a super-source and connect this vertex to all previous existing sources, while they are becoming normal vertices and proceed analogously with the sink vertices. For more than twenty years this method was also the fastest methods for this problem in planar graphs, but actually this method possibly destroys the given planarity of G.

While manipulating G would lead to a runtime of $O(n^2 \log n)$, Borradaile et al presented a faster algorithm for this specific problem by using the planarity of G. This algorithm has a runtime of $O(n \log^3 n)$ [2]. To achieve this runtime a very complex algorithm was created, so that only the used methods will be mentioned. In the presented faster min st-cut algorithm a divide and conquer strategy is used, which is also used for this problem. Also both algorithms are operating on dense distance graphs to speed up the computation. But the algorithm for multiple-sink multiple-source uses also a lot of other techniques like multiple source shortest path or pseudoflows. In the end this algorithm takes $O(n \log^3 n)$ and is the first algorithm for this problem, which can take advantage of the planar characteristic of G and therefore allows, amongst other problems, some computer vision problems to be solved faster.

7 Conclusion

Reif's algorithm can solve the min st-cut problem in planar undirected graphs. This algorithm uses that a min st-separating cycle on the dual graph can be found, in order to find a min st-cut. Then on the dual graph a shortest path π from face s to face t can be computed and the shortest path from every vertex of π to a copy of itself can be calculated. Because of that, we get all st-separating cycles, which are minimal while including a vertex $p \in \pi$ and a minimum of these can be chosen to get the min st-cut.

This idea was developed further resulting in the faster min st-cut algorithm. This algorithm uses a two phase approach and does some preprocessing to achieve a runtime of $O(n \log \log n)$. A given dual graph is divided in $O(n/r)$ pieces, what allows dense distance graphs to be created. Then the first phase takes place. Basically Reif's algorithm is executed on a graph consisting only of boundary vertices and edges of the dense distance graphs, what allows this phase to be faster than the original algorithm. Afterwards in the second phase the remaining shortest path for the interior vertices can be computed. Because of the first phase, the graph is divided by shortest path of the boundary vertices, what speeds up the second phase, since again not many vertices have to be taken into account for this computation.

The max st-flow problem in undirected planar graphs can be solved by using this algorithm and an $O(n)$ algorithm of Hassin and Johnson, resulting in a total runtime of also $O(n \log \log n)$.

At the end of this paper an algorithm for max st-flow in directed planar graphs was presented with a runtime of $O(n \log n)$. Furthermore an outlook on the, in planar graphs non-trivial, multiple-source multiple-sink problem in directed planar graphs was given. An algorithm, which is the first one for this problem in planar graphs, can solve this problem in $O(n \log^3 n)$.

References

[1] G. Borradaile and P. Klein. An o(n log n) algorithm for maximum st-flow in a directed planar graph. *J. ACM*, 56(2):9:1–9:30, Apr. 2009.

[2] G. Borradaile, P. N. Klein, S. Mozes, Y. Nussbaum, and C. Wulff-Nilsen.

Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time.

[3] G. N. Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[4] R. Hassin and D. B. Johnson. An $\mathrm{O}(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM Journal on Computing*, 14(3):612–624, 1985.

[5] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1):3–23, 1997.

[6] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.

[7] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. pages 313–322, 2011.

[8] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155. Society for Industrial and Applied Mathematics, 2005.

[9] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and system Sciences*, 32(3):265–279, 1986.

[10] R. Trudeau. *Introduction to Graph Theory.* Dover Books on Mathematics. Dover Publications, 2013.