

Improved Minimum Cuts and Maximum Flows in Undirected Planar Graphs

Giuseppe F. Italiano* Piotr Sankowski†

November 3, 2018

Abstract

In this paper we study minimum cut and maximum flow problems on planar graphs, both in static and in dynamic settings. First, we present an algorithm that given an undirected planar graph computes the minimum cut between any two given vertices in $O(n \log \log n)$ time. Second, we show how to achieve the same $O(n \log \log n)$ bound for the problem of computing maximum flows in undirected planar graphs. To the best of our knowledge, these are the first algorithms for those two problems that break the $O(n \log n)$ barrier, which has been standing for more than 25 years. Third, we present a fully dynamic algorithm that is able to maintain information about minimum cuts and maximum flows in a plane graph (i.e., a planar graph with a fixed embedding): our algorithm is able to insert edges, delete edges and answer min-cut and max-flow queries between any pair of vertices in $O(n^{2/3} \log^3 n)$ time per operation. This result is based on a new dynamic shortest path algorithm for planar graphs which may be of independent interest. We remark that this is the first known non-trivial algorithm for min-cut and max-flow problems in a dynamic setting.

*Dipartimento di Informatica, Sistemi e Produzione, University of Rome "Tor Vergata", italiano@disp.uniroma2.it. Work partially supported by the 7th Framework Programme of the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future - From Theory to Design") and by MIUR, the Italian Ministry of Education, University and Research, under Project AlgoDEEP.

†Institute of Informatics, University of Warsaw and Department of Computer and System Science, Sapienza University of Rome, sank@mimuw.edu.pl.

1 Introduction

Minimum cut and maximum flow problems have been at the heart of algorithmic research on graphs for over 50 years. Particular attention has been given to solving those problems on planar graphs, not only because they often admit faster algorithms than general graphs but also since planar graphs arise naturally in many applications. The pioneering work of Ford and Fulkerson [3, 4], which introduced the max-flow min-cut theorem, also contained an elegant algorithm for computing maximum flows in (s, t) -planar graphs (i.e., planar graphs where both the source s and the sink t lie on the same face). The algorithm was implemented to work in $O(n \log n)$ time by Itai and Shiloach [9]. Later, a simpler algorithm for the same problem was given by Hassin [6], who reduced the problem to single-source shortest path computations in the dual graph. The time required to compute single-source shortest paths in (s, t) -planar graphs was shown to be $O(n\sqrt{\log n})$ by Frederickson [5] and later improved to $O(n)$ by Henzinger *et al.* [8]. As a result, minimum cuts and maximum flows can be found in $O(n)$ time in (s, t) -planar graphs.

Itai and Shiloach [9] generalized their approach to the case of general planar (i.e., not only (s, t) -planar) graphs, by observing that the minimum cut separating vertices s and t in a planar graph G is related to the minimum cost cycle that separates faces f_s and f_t (corresponding to vertices s and t) in the dual graph. The resulting algorithm makes $O(n)$ calls to their original algorithm for (s, t) -planar graphs and thus runs in a total of $O(n^2 \log n)$ time. In the case of undirected planar graphs, Reif [13] improved this bound by describing how to find the minimum cost separating cycle with a divide-and-conquer approach using only $O(\log n)$ runs of the (s, t) -planar algorithm: this yields an $O(n \log^2 n)$ time to compute a minimum cut for undirected planar graphs. Later on, Frederickson [5] improved the running time of Reif's algorithm to $O(n \log n)$. The same result can be obtained by using more recent planar shortest path algorithms (see e.g., [8]). Hassin and Johnson [7] extended the minimum cut algorithm of Reif to compute a maximum flow in only $O(n \log n)$ additional time: this implies an undirected planar maximum flow algorithm that runs in $O(n \log n)$ time as well. In summary, the best bound known for computing minimum cuts and maximum flows in planar undirected graphs is $O(n \log n)$.

The first contribution of this paper is to improve to $O(n \log \log n)$ the time for computing minimum cuts in planar undirected graphs. To achieve this bound, we improve Reif's classical approach [13] with several novel ideas. To compute a minimum s - t cut in a planar graph G , we first identify a path π between face f_s (corresponding to vertex s) and face f_t (corresponding to vertex t) in the dual graph G_D . Next, we compute a new graph G_π as follows. We cut G_D along path π , so that the vertices and edges of π are duplicated and lie on the boundary of a new face: another copy of the same cut graph is embedded inside this face. We show that minimum separating cycles in G_D correspond to some kind of shortest paths in G_π . Applying a divide-and-conquer approach on the path π yields the same $O(n \log n)$ time bound as previously known algorithms [5, 7, 13]. However, our novel approach has the main advantage that it allows the use of *any* path π in the dual graph G_D , while previous algorithms were constrained to choose π as a shortest path. We will exploit the freedom implicit in the choice of this path π to produce a faster $O(n \log \log n)$ time algorithm, by using a suitably defined cluster decomposition of a planar graph, combined with the Dijkstra-like shortest path algorithm by Fakcharoenphol and Rao [2]. Our second contribution is to show that also maximum flows can be computed in undirected planar graphs within the same $O(n \log \log n)$ time bound. We remark that this is not an immediate consequence of our new minimum cut algorithm: indeed the approach of Hassin and Johnson [7] to extend minimum cut algorithms to the problem of computing maximum flows has a higher overhead of $O(n \log n)$. To get improved maximum flow algorithms, we have to appropriately modify the original technique of Hassin and Johnson [7]. To the best of our knowledge, the algorithms presented in this paper are

the first algorithms that break the $O(n \log n)$ long-standing barrier for minimum cut and maximum flow problems in undirected planar graphs.

As our third contribution, we present a fully dynamic algorithm that is able to maintain information about minimum cuts and maximum flows in a plane graph (i.e., a planar graph with a fixed embedding): our algorithm is able to insert edges, delete edges and answer min-cut and max-flow queries between any pair of vertices in $O(n^{2/3} \log^3 n)$ time per operation. This result is based on the techniques developed in this paper for the static minimum cut algorithm and on a new dynamic shortest path algorithm for planar graphs which may be of independent interest. We remark that this is the first known non-trivial algorithm for min-cut and max-flow problems in a dynamic setting.

2 Minimum Cuts in Planar Graphs

Let $G = (V, E, c)$ be a planar undirected graph where V is the vertex set, E is the edge set and $c : E \rightarrow \mathcal{R}^+$ is the edge capacity function. Let the planar graph G be given with a certain embedding. Using the topological incidence relationship between edges and faces of G , one can define the *dual graph* $G_D = (F, E_D, c_D)$ as follows. Each face of G gives rise to a vertex in F . Dual vertices f_1 and f_2 are connected by a dual undirected edge e_D whenever primal edge e is adjacent to the faces of G corresponding to f_1 and f_2 . The weight $c_D(e_D)$ of the dual edge e_D is equal to the weight $c(e)$ of the primal edge: $c_D(e_D)$ is referred to as the length of edge e_D . In other terms, the length of the dual edge e_D is equal to the capacity of the primal edge e . In the following, we refer to G as the *primal* graph and to G_D as its *dual*. Throughout the paper we will refer to vertices of the dual graph G_D interchangeably as (dual) vertices or faces. Note that G_D can be embedded in the plane by placing each dual vertex inside the corresponding face of G , and placing dual edges so that each one crosses only its corresponding primal edge. Thus, the dual graph is planar as well, although it might contain multiple edges and self loops. In simpler terms, the dual graph G_D of a planar embedded graph G is obtained by exchanging the roles of faces and vertices, and G and G_D are each other's dual. Figure 1 shows an embedded planar graph G and its dual G_D .

Let s and t be any two vertices of G (not necessarily on the same face). We consider the problem of finding a minimum cut in G between vertices s and t . Let C be a cycle of graph G : we define the *interior* of C , denoted by $\text{int}(C)$, to be the region inside C and including C in the planar embedding of the graph G . We can define the *exterior* $\text{ext}(C)$ of the cycle C in a similar fashion. A cycle C in G_D is said to be a *cut-cycle* if $\text{int}(C)$ contains exactly s but not t . The following lemma was proven by Johnson [10].

Lemma 1. *A minimum s - t cut in G has the same cost as a minimum cost cut-cycle of G_D .*

The lemma follows by the observation that for any cut-cycle C the faces of G_D inside $\text{int}(C)$ give a set of vertices S in G which defines a cut separating s and t . Note that Lemma 1 gives an equivalence between min-cuts in the primal graph G and minimum cost cut-cycles in the dual graph G_D . By using a divide-and-conquer approach, this equivalence can be turned into an efficient algorithm for finding flows in undirected planar graphs [13]. The resulting algorithm, combined with more recent results on shortest paths in planar graphs [8], is able to work in a total of $O(n \log n)$ time. However, this approach seems to inherently require $O(n \log n)$ time, and does not seem to leave margin for improvements. In the next section, we will present a completely different and more flexible approach, which will yield faster running times.

2.1 Computing Min-Cuts

Let f_s and f_t be arbitrary inner faces incident to s and to t respectively. Find any simple path π from f_s to f_t in G_D . The path π can be viewed as connecting special vertices in the dual graph

corresponding to s and t . Hence, any s - t cut needs to cross this path, because it splits s from t . Let π traverse dual vertices f_1, \dots, f_k , where $f_1 = f_s$ and $f_k = f_t$. Let us look at the path π as a horizontal line, with f_s on the left and f_t on the right (see Figure 2(b)). An edge $e_D \notin \pi$ in G_D such that e_D is incident to some face f_i , $1 \leq i \leq k$, can be viewed as connected to f_i from below or from above. We now define a new graph G'_π , by cutting G_D along path π , so that the vertices and edges of π are duplicated and lie on the boundary of a new face. This is done as follows. Let π' be a copy of π , traversing new vertices f'_1, f'_2, \dots, f'_k . Then G'_π is the graph obtained from G_D by reconnecting to f'_i edges entering f_i from above, $1 \leq i \leq k$ (see Figure 2(c)). Let G''_π be a copy of G'_π . Turn over the graph G''_π to make the face defined by π and π' to be the outer face (see Figure 2(d)). Now, identify vertices on the path π' (respectively π) in G''_π with the vertices on the path π (respectively π') in G'_π . Denote the resulting graph as G_π (see Figure 2(e)). Note that the obtained graph G_π is planar.

We define an f_i -cut-cycle to be a cut-cycle in G_D that includes face f_i and does not include any face f_j for $j > i$. The proof of the following lemma is immediate.

Lemma 2. *Let C_i be a minimum f_i -cut-cycle in G_D for $i = 1, \dots, k$. Then C_i with minimum cost is a minimum cut-cycle in G_D .*

A path ρ between f_i and f'_i in G_π is said to be f_i -separating if ρ contains neither f_j nor f'_j , for $j > i$. We say that a cycle C in G_D touches path π in face f_i if two edges on C incident to f_i go both up or both down, whereas we say that C crosses π in face f_i if one of these edges goes up, whereas the other goes down.

Lemma 3. *The cost of a minimum f_i -cut-cycle in G_D is equal to the length of a shortest f_i -separating path in G_π .*

Proof. Let C be some f_i -cut-cycle in G_D : we show that there must be some f_i -separating path ρ in G_π having the same cost as C . Note that the f_i -cut-cycle C must either cross or touch the path π in face f_i . First, assume that C crosses π in f_i . Note that in this case C has to cross the path π an even number of times (excluding f_i), as otherwise C would not separate s from t (see Figure 3(a)). We can go along C starting from f_i in the graph G_π and each time when C crosses π in G_D , we switch between G'_π and G''_π in G_π . Hence, due to parity of the number of crossings with π , this will produce a resulting path ρ in G_π which must end in f'_i (see Figure 3(b)). Second, assume that C touches π in f_i . Then C has to cross path π an odd number of times (see Figure 3(c)). Again if we trace C starting from f_i in G_π we will produce a path ρ in G_π ending up in f'_i (see Figure 3(d)). Moreover, since C is a f_i -cut-cycle in G_D , it cannot contain by definition any face f_j , for $j > i$: consequently, in either case the resulting path ρ in G_π will contain neither f_j nor f'_j , for $j > i$.

Conversely, let ρ be some f_i -separating path in G_π : we show that there must be some f_i -cut-cycle C in G_D having the same cost as ρ . First, assume that ρ enters f_i and f'_i using edges from the same graph, i.e., either G'_π or G''_π . If this is the case, then the two edges must be such that one is from above and the other is from below. By tracing ρ in G_D we obtain a cycle C that crosses π in f_i . This cycle does not need to be simple, but it has to cross π an odd number of times (including the crossing at f_i), as each time when C crosses π the path ρ must switch between G'_π and G''_π . Hence, C must be a cut-cycle. Second, assume that ρ enters f_i and f'_i using edges from different graphs, i.e., one edge from G'_π and the other edge from G''_π . If this is the case, then the two edges must be both from above or both from below. Consider first the case where ρ leaves f_i from above using an edge from G''_π , and enters f'_i from above using an edge from G'_π (see Figure 3(d)). This time we get a cut-cycle C touching π in f_i and intersecting π an odd number of times. The case where ρ leaves f_i from below using an edge from G'_π , and enters f'_i from below using an edge from G''_π is completely analogous (see Figure 3(f)). Once again, since ρ is an f_i -separating path in G_π , in any case the resulting cut cycle C will not contain f_j , for $j > i$. \square

Note that, having constructed the graph G_π , we can find the shortest f_i -separating path by simply running Dijkstra's algorithm on G_π where we remove all faces f_j and f'_j for $j > i$. By Lemmas 1, 2 and 3, the linear-time implementation of Dijkstra's algorithm known for planar graphs [8] implies an $O(n^2)$ time algorithm for computing minimum cuts in planar graphs. There is a more efficient way of computing minimum cuts that goes along the lines of Reif's recursive algorithm [13]. Before describing this approach, we need to prove some non-crossing properties of f_i -cut-cycles.

Lemma 4. *For $i < j$, let C_i be a minimum f_i -cut-cycle in G_D , and let C_j be a minimum f_j -cut-cycle in G_D . Then there exists a cut cycle $C \subseteq \text{int}(C_i) \cap \text{int}(C_j)$ in G_D such that $c(C) \leq c(C_i)$.*

Proof. If $C_i \subseteq \text{int}(C_j)$ then $C = C_i$ and the lemma follows trivially. On the other hand, it is impossible that $C_j \subseteq \text{int}(C_i)$ because in this case C_i would include some face f_k on π for $k \geq j > i$.

The only possibility left is that $C_i \not\subseteq \text{int}(C_j)$ and $C_j \not\subseteq \text{int}(C_i)$. In this case, there must exist a subpath p_i of C_i from f_a to f_b such that p_i intersects $\text{int}(C_j)$ only at f_a and f_b . Let p_j be the subpath of C_j going from f_a to f_b (see Figure 4). We claim that $c(p_j) \leq c(p_i)$. Indeed, suppose by contradiction that $c(p_j) > c(p_i)$, and let C'_j be the cycle obtained from C_j after replacing path p_j with p_i . Then the cycle C'_j is shorter than C_j . Moreover, since C_i does not include any face f_k for $k > i$, also C'_j cannot include any face f_k for $k > j > i$. As a consequence, the cycle C'_j is an f_j -cut-cycle in G_D , with $c(C'_j) < c(C_j)$, contradicting our assumption that C_j is a minimum f_j -cut-cycle in G_D .

Since $c(p_j) \leq c(p_i)$, replacing path p_i on C_i with the path p_j yields a cycle C'_i , with $c(C'_i) \leq c(C_i)$. As long as $C'_i \not\subseteq \text{int}(C_j)$ and $C_j \not\subseteq \text{int}(C'_i)$ we can repeat this procedure. At the end, we will obtain a cycle $C \subseteq \text{int}(C_i) \cap \text{int}(C_j)$, such that $c(C) \leq c(C_i)$. Moreover, the obtained cycle C will be a cut-cycle, as $\text{int}(C_i) \cap \text{int}(C_j)$ contains s . \square

Lemma 5. *For $i < j$, let C_i be a minimum f_i -cut-cycle in G_D , and let C_j be a minimum f_j -cut-cycle in G_D . Then, for some $i' \leq i$, there exists a minimum $f_{i'}$ -cut-cycle $C_{i'} \subseteq \text{int}(C_j)$ such that $c(C_{i'}) \leq c(C_i)$.*

Proof. Consider the cut-cycle C contained inside $\text{int}(C_i) \cap \text{int}(C_j)$ as given by Lemma 4, for which we know that $c(C) \leq c(C_i)$. As $C \subseteq \text{int}(C_i)$, C cannot contain any f_k such that $k > i$. Hence, it is an $f_{i'}$ -cut-cycle for some $i' \leq i$. The minimum $f_{i'}$ -cut-cycle $C_{i'}$ is shorter than C and consequently it is shorter than C_i . If $C_{i'} \subseteq \text{int}(C_j)$, the lemma follows. Otherwise, we can apply Lemma 4 to produce another minimum $f_{i''}$ -cut-cycle contained in $\text{int}(C_j)$ such that $c(C_{i''}) \leq c(C_i)$, for some $i'' \leq i' \leq i$. \square

The above lemma shows that each computed cut splits the graph into two parts, the interior and the exterior part of the cut, which can be handled separately. Hence, we can use a divide and conquer approach on the path π . We first find a minimum cut-cycle that contains the middle vertex on the path π . Then we recurse on both parts of the path, so there will be $O(\log n)$ levels of recursion in total. In this recursion we will compute minimum cost $s-t$ cuts for all vertices on π . By Lemma 3, we compute a minimum f_i -cut-cycle for some f_i by finding a shortest path in the planar graph G_π . Then we need to divide the graph into the inside and the outside of the cut. The vertices on the minimum cut need to be included into both parts, so we need to take care that the total size of the parts does not increase too much. This can be done in a standard way as described by Reif [13] or by Hassin and Johnson [7]. Their technique guarantees that on each level of the recursion the total size of the parts is bounded by $O(n)$. Hence, using the $O(n)$ -time algorithm [8] for shortest paths we get an $O(n \log n)$ time algorithm for finding minimum cuts in planar graphs. Although this approach yields the same time bounds as previously known algorithms [7, 13], it has the main advantage to allow the use of any path π in G_D , while previous algorithms were constrained to choose π as a shortest path. As shown in the next section, we can even allow the path π to be

non-simple. In Section 3 we will show how to exploit the freedom implicit in the choice of the path π to produce a faster $O(n \log \log n)$ time algorithm for finding minimum cuts in planar graphs.

2.2 Using Non-simple Paths

Let $\pi = (v_1, \dots, v_\ell)$ be a non-simple path and let v be a vertex appearing at least twice on the path π , i.e., $v = v_i = v_j$ for some $i < j$. We say that the path π is *self-crossing* in v if the edges incident to v on π appear in the following circular order $(v_{i-1}, v_i), (v_{j-1}, v_j), (v_i, v_{i+1}), (v_j, v_{j+1})$ in the embedding. We say that a path π is self-crossing if π is self-crossing in some vertex v . Otherwise we say that π is *non-crossing*. If a vertex v appears at least twice on a non-crossing path π than we say that π *touches itself* in v . In the previous section we assumed that the path π from f_s to f_t in G_D was simple, now we will show that we only need the weaker assumption that π is non-crossing.

In order to work with non-crossing paths we will modify the graph G_D to make the path π simple. Let $v = v_i$ be a face where π touches itself in G_D . Note that there is no other edge from π between the edges $(v_{i-1}, v_i), (v_i, v_{i+1})$ in the circular order around v given by the embedding. Take all edges E_v incident to v that are embedded between and including the edges $(v_{i-1}, v_i), (v_i, v_{i+1})$. Now, add a new face v' to G_D and make the edges E_v to be incident with v' instead of v . Finally connect v with v' using an undirected edge of length zero (see Figure 5). Let π be a non-crossing path: perform this vertex-splitting operation until π becomes a simple path. This produced a new graph $G_{D,\pi}$. Note that this transformation does not change the lengths of cut-cycles, so we get the following observation.

Corollary 6. *The lengths of minimum f_i -cut-cycles in G_D and $G_{D,\pi}$ are the same.*

As a result, if π is a non-simple non-crossing path, instead of running our algorithm on G_D , we can compute the graph $G_{D,\pi}$ and run our algorithm on $G_{D,\pi}$.

3 Cluster Partitions

Our algorithms are based on a particular cluster decomposition. We start by presenting the ideas behind this cluster decomposition and then show how this decomposition can be effectively exploited for computing min-cuts. From now on, we assume that we are given a graph for which the dual graph has vertex degree at most three. This is without loss of generality, since it can be obtained by triangulating the primal graph with zero capacity edges.

Let n be the number of vertices of G . We first define a *cluster partition* of G into edge clusters which will be used by our algorithm. In the cluster partition the graph is decomposed into a set of cluster \mathcal{P} such that each edge of G is included into exactly one cluster. Each cluster contains two types of vertices: *internal* vertices, and *border* vertices. An internal vertex is adjacent only to vertices in the same cluster, while a border vertex is adjacent to vertices in different clusters. A *hole* in a cluster is any face (including the external face) containing only boundary vertices. We denote by ∂P the set of border vertices of a cluster P . We define an *r-partition* of an n -node graph to be a cluster partition that contains at most $O(\frac{n}{r})$ clusters, each containing at most r vertices, $O(\sqrt{r})$ border vertices and a constant number of holes. The proof of the following lemma is in Appendix A. It is based on ideas of Frederickson [5], who constructed a similar partition without the bound on the number of holes.

Lemma 7. *An r-partition of an n -node planar graph can be computed in $O(n \log r + \frac{n}{\sqrt{r}} \log n)$ time.*

We use the *r*-partition to define a representation for shortest paths in a graph that has similar number of edges, but fewer vertices. In order to achieve this, we use the notion of dense distance

graphs. A *dense distance graph* for a cluster C is defined to be a complete graph over the border vertices of C where edge lengths correspond to shortest path distances in C . In order to compute dense distance graphs for all clusters we use Klein's algorithm [11], who have shown that after $O(n \log n)$ preprocessing time any distance from the external face can be computed in $O(\log n)$ time. The proof of the following lemma is given in Appendix B.

Lemma 8. *Given an r -partition \mathcal{P} we can compute a dense distance graph for all clusters in \mathcal{P} in $O(n \log r)$ time.*

The dense distance graphs can be used to speed up shortest path computations using Dijkstra's algorithm. It was shown by Fakcharoenphol and Rao ([2], Section 3.2.2) that a Dijkstra-like algorithm can be executed on a dense distance graph for a cluster P in $O(|\partial P| \log^2 |\partial P|)$ time. Having constructed the dense distance graphs, we can run Dijkstra in time almost proportional to the number of vertices (rather than to the number of edges, as in standard Dijkstra). We use this algorithm in graphs composed of dense distance graphs and a subset E' of edges of the original graph $G = (V, E)$:

Corollary 9. *Dijkstra can implemented in $O(|E'| \log |V| + \sum_i |\partial G_i| \log^2 |\partial G_i|)$ time on a graph composed of a set of dense distance graphs G_i and a set of edges E' over the vertex set V .*

Proof. In order to achieve this running time we use Fakcharoenphol and Rao [2] data-structure for each G_i . Moreover, minimum distance vertices from each G_i and all endpoints of edges in E' are kept in a global heap. \square

In general, clusters may contain holes although in typical cases, e.g., in grid graphs, the obtained clusters are holeless. In this section, in order to introduce the main ideas behind our algorithm, we restrict ourselves to *holeless r -partitions*, i.e., r -partitions where each cluster contains one hole (external face). We will show in Appendix D how to handle the general case.

Assume that we have computed the dense distance graphs for all clusters in a given r -partition \mathcal{P} of the dual graph G_D . Recall that in our min-cut algorithm we are free to choose any path π from f_s to f_t in G_D , as long as π is non-crossing. We choose π to minimize the number of clusters it crosses and to maximally use the dense representation of clusters.

We define a *skeleton graph* $G_{\mathcal{P}} = (\partial \mathcal{P}, E_{\mathcal{P}})$ to be a graph over the set of border vertices in \mathcal{P} . The edge set $E_{\mathcal{P}}$ is composed of infinite length edges connecting consecutive (in the order on the hole) border vertices on each hole. By our holeless assumption, all border vertices in each cluster lie on the external face of the cluster, so the graph $G_{\mathcal{P}}$ is connected. We define a *patched graph* to be $\overline{G} = G_D \cup G_{\mathcal{P}}$. Note that this graph is still planar and the shortest distances do not change after adding infinite length edges.

Define $\overline{G}^{s,t}$ to be the graph composed of: (1) two clusters P_s and P_t that include f_s and f_t respectively; (2) the dense distance graphs (represented by square matrices) for all other clusters; (3) the skeleton graph $G_{\mathcal{P}}$ (see Figure 6). Note that $\overline{G}^{s,t}$ contains: $O(2r + \frac{n}{r}\sqrt{r}) = O(r + \frac{n}{\sqrt{r}})$ vertices; $O(\frac{n}{r})$ dense distance graphs each over $O(\sqrt{r})$ vertices; at most $3r$ edges of G_D from P_s and P_t ; additional $O(\frac{n}{\sqrt{r}})$ edges from the skeleton graph $G_{\mathcal{P}}$. Using Corollary 9 to run Dijkstra's algorithm we get the following.

Corollary 10. *The shortest paths in $\overline{G}^{s,t}$ can be computed in $O((r + \frac{n}{\sqrt{r}}) \log^2 n)$ time.*

3.1 Recursive Division

Let b_s and b_t be any border vertices in clusters P_s and P_t respectively. We define π to be composed of: a simple path from f_s to b_s in P_s ; a simple path from b_s to b_t in $G_{\mathcal{P}}$; and a simple path

from b_t to f_t in P_t . Note that the construction of the graph $\overline{G}^{s,t}$ and consequently of $\overline{G}_\pi^{s,t}$ takes $O(n \log r + \frac{n}{\sqrt{r}} \log n)$ time by Lemmas 7 and 8.

Let C_i be some f_i -cut-cycle in G_D . After finding C_i , we need to recurse on the graphs $G_D \cap \text{int}(C_i)$ and $G_D \cap \text{ext}(C_i)$. These graphs cannot be computed explicitly. However, we will show how to determine $\overline{G}^{<i} := \overline{G_D \cap \text{int}(C_i)}^{s,t}$ and $\overline{G}^{i>} := \overline{G_D \cap \text{ext}(C_i)}^{s,t}$ using $\overline{G}^{s,t}$. Let P be a cluster in the partition other than P_s or P_t , and let G_P be its dense distance graph. For a set of vertices X we define $G_P \cap X$ to be the dense distance graph of $P \cap X$. The f_i -cut-cycle was found using $\overline{G}^{s,t}$ so parts of C_i that pass through P correspond to shortest paths. Hence, the shortest paths in $P \cap \text{int}(C_i)$ and $P \cap \text{ext}(C_i)$ cannot cross the cycle C_i . As a result, distances in $G_P \cap \text{int}(C_i)$ and $G_P \cap \text{ext}(C_i)$ between border vertices of P that are not separated by C_i are the same as in G_P . On the other hand, for border vertices that are separated by C_i the distances are infinite (see Figure 7). We define $\overline{G}^{s,t} \cap X$ to be the graph obtained by taking $G_P \cap X$ for every cluster in G . Note that we have the following.

Corollary 11. $\overline{G}^{<i} = \overline{G}^{s,t} \cap \text{int}(C_i)$ and $\overline{G}^{i>} = \overline{G}^{s,t} \cap \text{ext}(C_i)$.

Using these equalities the construction of $\overline{G}^{<i}$ and $\overline{G}^{i>}$ takes time proportional to the size of $\overline{G}^{s,t}$ only. Given a path $\pi = f_1, \dots, f_k$, the recursive algorithm for computing min-cuts works as follows. First, remove from $\overline{G}^{s,t}$ vertices with degree 2 by merging the two incident edges, and find an f_i -cut-cycle C_i for $i = \lfloor \frac{k}{2} \rfloor$. Next, construct graphs $\overline{G}^{<i}$ and $\overline{G}^{i>}$, and recursively find a minimum cut $C_{<i}$ in $\overline{G}^{<i}$ and a minimum cut $C_{i>}$ in $\overline{G}^{i>}$. Finally, return the smallest of the three cuts $C_{<i}$, C_i and $C_{i>}$.

To achieve our promised bounds, we need to show that the above algorithm works in sublinear time. In order to show that, we only need to prove that the total size of the graphs on each level in the recursion tree is small. Take a graph $\overline{G}^{s,t}$ and for each cluster P that contains more than one border vertex add a new vertex v_P and replace each dense distance graph by a star graph with v_P in a center. We call the resulting graph the *contracted graph* and denote it by $\overline{G}_c^{s,t}$. Obviously, the contracted graph has more vertices than the original graph: thus, we can bound the number of vertices in the original graph by considering only contracted graphs. The proof of this lemma is included in Appendix C.

Lemma 12. *The total number of vertices in contracted graphs on each level in the recursion tree is $O(r + \frac{n}{\sqrt{r}})$.*

By the above lemma and Corollary 9, running Dijkstra for each level takes $O((r + \frac{n}{\sqrt{r}}) \log^2 n)$ time in total. On each level the length of the path π is halved, so there are at most $\log n$ recursion levels. Hence we obtain the following theorem.

Theorem 13. *Let G be a flow network with the source s and the sink t . If G_D allows holeless r -partition, then the minimum cut between s and t can be computed in $O(n \log r + (r + \frac{n}{\sqrt{r}}) \log^3 n)$ time. By setting $r = \log^8 n$ we obtain an $O(n \log \log n)$ time algorithm.*

Theorem 13 holds also for general r -partitions within the same $O(n \log \log n)$ time bound. The modifications needed to make the above algorithm work in the general case are presented in Appendix D.

4 Computing Maximum Flows

The standard ways of computing maximum flow in near linear time assume that we already have computed its flow value f . It is given by the min-cut capacity and as we already know it can be computed in $O(n \log \log n)$ time. We will use the approach proposed by Hassin and Johnson [7],

but adopted to our case as we use a different family of cuts C_i . As argued in the following, their approach uses only very basic properties of the cut-cycles and can be directly applied to our case. Moreover, for the sake of brevity, we will assume that that we are given a holeless r -partition. The general case can be handled using ideas presented in Appendix D, and it will be included in the full version of this paper.

Let us define the graph \vec{G}_π to be the graph obtained from G'_π by adding directed edges of length $-f$ from f_i to f'_i for all $1 \leq i \leq k$. After Lemma 4.1 in Miller and Naor [12] we know the following.

Corollary 14. *The graph \vec{G}_π does not contain negative length cycles.*

Let r be such that C_r is the shortest of all C_i for $1 \leq i \leq k$. The above corollary assures that distance $\delta(v)$ from f'_r to a vertex v is well defined in \vec{G}_π . The next lemma follows by Theorem 1 in [7] or Section 5.1 in [12].

Lemma 15. *Let $e = (u, v)$ be the edge in G and let $e_D = (f_u, f_v)$ be the corresponding edge in G_D . The face f_u is defined to lie to the left when going from u to v . The function $f(u, v) := \delta(f_u) - \delta(f_v)$ defines the maximum flow in G .*

By this lemma, in order to construct the flow function we only need to compute distances from f'_r in \vec{G}_π . The cycles C_i for $i = 1, \dots, k$ divide \vec{G}_π into $k + 1$ subnetworks $\vec{G}_0, \dots, \vec{G}_k$ where for $1 \leq i \leq k - 1$, \vec{G}_i is the subnetwork bounded by and including C_i and C_{i+1} . The next lemma is a restatement of Lemma 2 from [7].

Lemma 16. *Let v be a vertex in N_i . Then if $i < r$ then there exists a shortest (f'_r, v) -path which is contained in $\bigcup_{j=0}^{r-1} \vec{G}_j$. Similarly, if $i \geq r$ then there exists a shortest (f'_r, v) -path which is contained in $\bigcup_{j=r}^k \vec{G}_j$.*

Lemma 16 implies that the computation of $\delta(v)$ for $v \in \bigcup_{j=0}^{r-1} \vec{G}_j$ can be restricted to $\bigcup_{j=0}^{r-1} \vec{G}_j$ only. We can restrict the computation for $v \in \bigcup_{j=r}^k \vec{G}_j$ in a similar fashion.

Similarly to [7] let us define a *normal* path to be a simple (f'_r, v) -path $\rho(v) = \rho_r \dots \rho_q \dots \rho_{2q-i}$ such that, for $j = r, \dots, q$, subpath ρ_j is in \vec{G}_j , and, for $j = q + 1, \dots, 2q - i$, subpath ρ_j is in \vec{G}_{2q-j} and uses no edges of negative length. We require as well that q is minimal. Hassin and Johnson [7] have shown that distances from f'_r in an n -vertex graph can be computed in $O(n \log n)$ time when for each vertex v there exists a shortest (f'_r, v) -path that is normal (Theorem 2 in [7]). Their computation is based on Dijkstra's algorithm and can be sped up using dense distance graphs and Corollary 9.

Lemma 17. *Distances from f'_r in the graph \vec{G}_π can be computed in $O(n \log r + (\frac{n}{\sqrt{r}} + r) \log^2 n)$ time when for each vertex v there exists a shortest (f'_r, v) -path that is normal.*

Proof. The algorithm works in two phases. First, we take \vec{G}_π and substitute each cluster P with its dense distance graph. The resulting graph has $O(\frac{n}{\sqrt{r}})$ vertices. The Dijkstra-like computation from [7] can be executed in $O((\frac{n}{\sqrt{r}} + r) \log^2 n)$ time on this graph. In this way we obtain distances $\delta(v)$ for border nodes v in all clusters in G_D . Second, for each cluster separately we run Hassin and Johnson's computation starting from border vertices only. The second phase works in $O(r \log r)$ time for each cluster, which yields $O(n \log r)$ time in total. \square

In order to use the above lemma we only need the following result, which can be proven in the same way as Lemma 3 from [7]. The proof in [7] uses only the fact that subpaths of cut cycles C_i are shortest paths and this holds in our case as well.

Lemma 18. *For any $i = r, \dots, k$, for every vertex v , in \vec{G}_i there exists a normal shortest path $\rho(v)$.*

Proof. Assume that a shortest path $\rho(v)$ intersects some cycle C_i . If the previous intersected cycle was C_i as well, then we can either short cut $\rho(v)$ using the part of C_i or short cut C_i using part of $\rho(v)$ (see Figure 8). This contradicts either the minimality of $\rho(v)$ or the minimality of C_i . Moreover, after $\rho(v)$ leaves C_i to go into \vec{G}_{i-1} it cannot use a negative edge any more, as otherwise it would cross itself. If it crosses itself then the resulting cycle could be removed as it cannot have a negative weight. \square

Combining Lemma 15, Lemma 17 and Lemma 18 we obtain the main result of this section.

Theorem 19. *The maximum flow in an undirected planar graph can be computed in $O(n \log r + (\frac{n}{\sqrt{r}} + r) \log^2 n)$ time. By setting $r = \log^8 n$ we obtain an $O(n \log \log n)$ time algorithm.*

5 Dynamic Shortest Paths and Max Flows in Planar Graphs

Most of the ideas presented in this section are not entirely new, but nevertheless combined together they are able to simplify and improve previously known approaches. Our dynamic algorithm builds upon the r -partition introduced in Section 3. We first show how to maintain a planar graph G_D with positive edge weights under an intermixed sequence of the following operations¹:

$insert_D(x, y, c)$	add to G_D an edge of length c between vertex x and vertex y , provided that the new edge preserves the planarity of G_D ;
$delete_D(x, y)$	delete from G_D the edge between vertex x and vertex y ;
$shortest_path_D(x, y)$	return a shortest path in G_D from vertex x to vertex y .

In our dynamic algorithm we maintain the r -partition of G_D together with dense distance graphs for all clusters in the partition. This information will be recomputed every \sqrt{r} operations. We now show how to handle the different operations. We start with operation *insert*: let (x, y) be the edge to be inserted, and let P_x (respectively P_y) be the cluster containing vertex x (respectively vertex y). If x and y are not already border vertices in clusters P_x and P_y , we make them border vertices in both clusters and add edge (x, y) arbitrarily either to cluster P_x or to cluster P_y . Next, we recompute the dense distance graphs of clusters P_x and P_y , as explained in Lemma 8. This requires overall time $O(r \log r)$. Note that an insert operation may increase by a constant the number of border vertices in at most two clusters and adds an edge to one cluster: since the partition into clusters is recomputed every \sqrt{r} operations, this will guarantee that throughout the sequence of operations each cluster will always have at most $O(r)$ edges and $O(\sqrt{r})$ border vertices. To delete edge (x, y) , we remove this edge from the only cluster containing it, and recompute the dense distance graph of this cluster. Once again, this can be carried out in time $O(r \log r)$. Amortizing the initialization over \sqrt{r} operations yields $O(\frac{n \log r + \frac{n}{\sqrt{r}} \log n}{\sqrt{r}} + r \log r)$ time per update.

In order to answer a *shortest_path*(x, y) query, we construct the graph $\bar{G}^{s,t}$ (as defined in Section 3). Note that the distance from s to t in G_D and $\bar{G}^{s,t}$ are equal. Hence, by Corollary 10 the shortest path from s to t can be computed in $O((r + \frac{n}{\sqrt{r}}) \log^2 n)$ time.

In order to minimize the update time we set $r = n^{2/3}$ and obtain the following theorem.

¹We have chosen to work with the dual graph G_D for consistency with the remaining parts of the paper.

Lemma 20. *Given a planar graph G with positive edge weights, we can insert edges (allowing changes of the embedding), delete edges and report shortest paths between any pair of vertices in $O(n^{2/3} \log^2 n)$ amortized time per operation.*²

We recall that we can check whether a new edge insertion violates planarity within the bounds of Lemma 20: indeed the algorithm of Eppstein *et al.* [1] is able to maintain a planar graph subject to edge insertions and deletions that preserve planarity, and allow to test whether a new edge would violate planarity in time $O(n^{1/2})$ per operation. Finally, we observe that not only we have improved slightly the running time over the $O(n^{2/3} \log^{7/3} n)$ time algorithm by Fakcharoenphol and Rao [2], but our algorithm is also more general. Indeed, the algorithm in [2] can only handle edge cost updates and it is not allowed to change the embedding of the graph. On the contrary, our algorithm can handle the full repertoire of updates (i.e., edge insertions and deletions) and it allows the embedding of the graph to change throughout the sequence of updates.

We now turn our attention to dynamic max-flow problems. In particular, given a planar graph $G = (V, E)$ we wish to perform an intermixed sequence of the following operations:

$\text{insert}(x, y, c)$	add to G an edge of capacity c between vertex x and vertex y , provided that the embedding of G does not change;
$\text{delete}(x, y)$	delete from G the edge between vertex x and vertex y ;
$\text{max_flow}(s, t)$	return the value of the maximum flow from vertex s to vertex t in G .

Note that *insert* operations are now more restricted than before, as they are not allowed to change the embedding of the graph. To answer *max_flow* queries in the primal graph G we need to maintain dynamically information about the distances in the dual graph G_D , with the bounds reported in Lemma 20. Unfortunately, things are more involved as a single edge change in the primal graph G may cause more complicated changes in the dual graph G_D . In particular, inserting a new edge into the primal graph G results in splitting into two a vertex in the dual graph G_D , whereas deleting an edge in the primal graph G implies joining two vertices of G_D into one. However, as edges are inserted into or deleted from the primal graph, vertices in the dual graph are split or joined according to the embedding of their edges. To handle efficiently vertex splits and joins in the dual graph, we do the following. Let f be a vertex of degree d in the dual graph: we maintain vertex f as a cycle C_f of d edges, each of cost 0. The actual edges originally incident to f , are made incident to one of the vertices in C_f in the exact order given by the embedding. It is now easy to see that in order to join two vertices f_1 and f_2 , we need to cut their cycles C_{f_1} and C_{f_2} , and join them accordingly. This can be implemented in a constant number of edge insertions and deletions. Similarly, we can support vertex splitting with a constant number of edge insertions and deletions. Additionally, for each cluster P , for each pair h, h' of holes we need to compute the dense distance graphs of $P_{\pi_h, h'}$ and the minimum-cuts between b_h and $b_{h'}$. However, following Corollary 24, this does not increase the running time of our dynamic algorithm. Note that this information is enough to construct the graphs $\overline{G}^{s,t}$ and $\overline{G}_\pi^{s,t}$ in $O(r + \frac{n}{\sqrt{r}})$ time. Moreover, by Theorem 13 and Theorem 26 the min-cut algorithm can be executed on these graphs in $O((r + \frac{n}{\sqrt{r}}) \log^3 n)$ time. Setting $r = n^{2/3}$ this yields immediately the main result of this section.

Lemma 21. *Given a planar graph G with positive capacities, each operation *insert*, *delete* and *max_flow* can be implemented in $O(n^{2/3} \log^3 n)$ amortized time.*³

²The same worst-case time bounds can be obtained using a global rebuilding technique.

³Again, the same worst-case time bounds can be obtained using a global rebuilding technique.

References

- [1] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification I. Planarity testing and minimum spanning trees. *Journal of Computer and System Sciences*, 52(1):3–27, February 1996.
- [2] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.
- [3] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, pages 399–404, 1956.
- [4] L. R. Ford and D. R. Fulkerson. *Flows in Network*. Princeton Univ. Press, 1962.
- [5] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.
- [6] R. Hassin. Maximum flows in (s,t) planar networks. *IPL*, page 107, 1981.
- [7] Refael Hassin and Donald B. Johnson. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM J. Comput.*, 14(3):612–624, 1985.
- [8] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.
- [9] Alon Itai and Yossi Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.
- [10] Donald B. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *J. ACM*, 34(4):950–967, 1987.
- [11] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *SODA '05: Proc. 16th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 146–155, 2005.
- [12] Gary L. Miller and Joseph Naor. Flow in planar graphs with multiple sources and sinks (extended abstract). In *FOCS*, pages 112–117. IEEE, 1989.
- [13] J. Reif. Minimum $s-t$ cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM Journal on Computing*, 12:71–81, 1983.

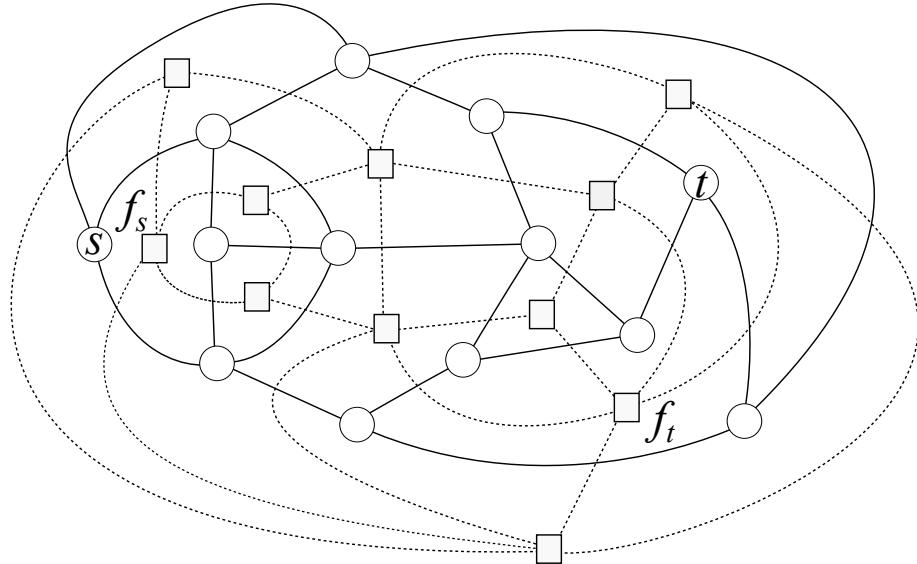


Figure 1: An embedded planar graph G and its dual graph G_D . Vertices of G are shown as circles, and edges of G are solid. Vertices of G_D are shown as gray squares, and edges of G_D are dashed. s and t are two vertices in G , and f_s and f_t are arbitrary inner faces incident respectively to s and t .

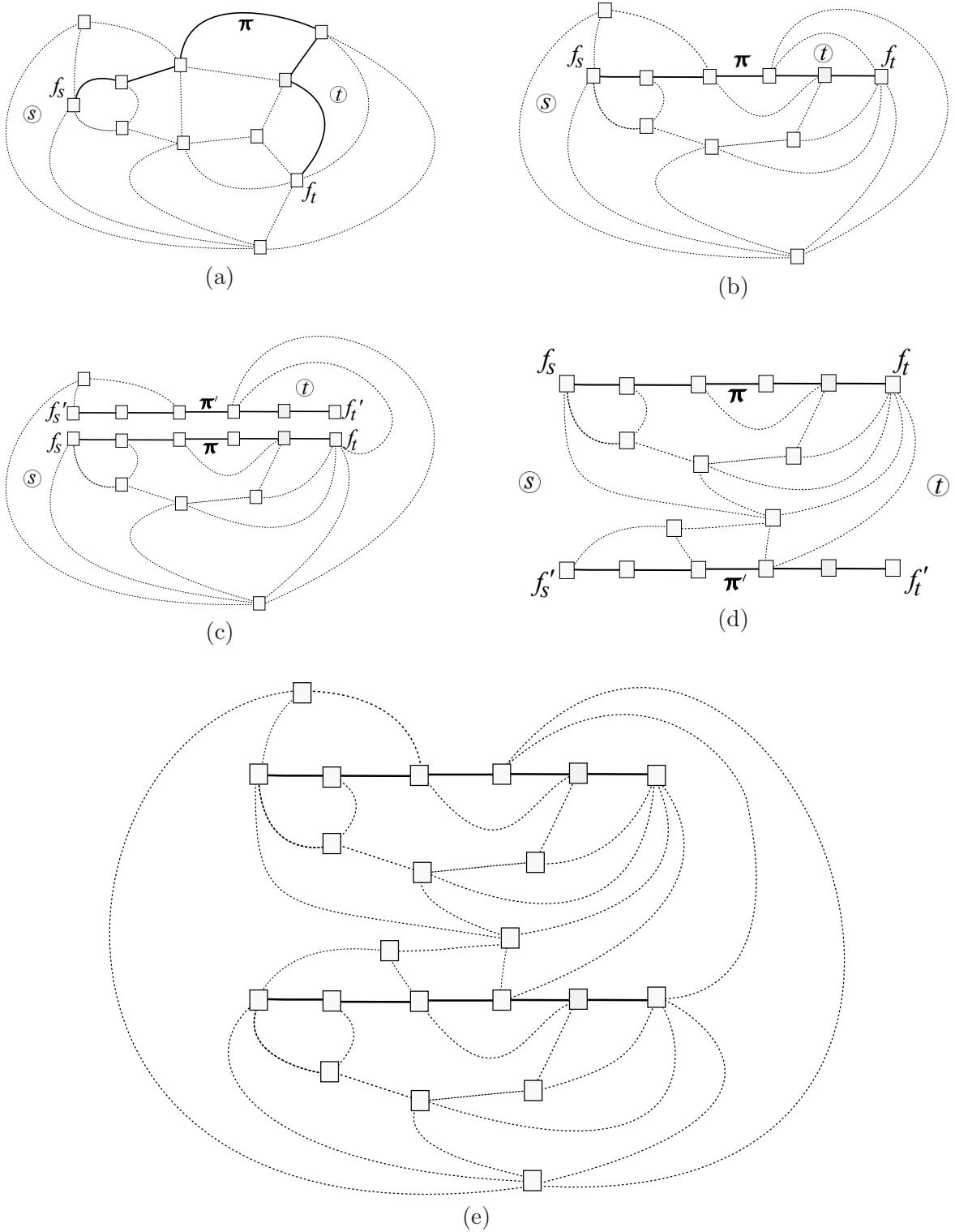


Figure 2: (a) The dual graph G_D of Figure 1: the path π from f_s to f_t is shown in bold. (b) The dual graph G_D embedded so that the path π is a horizontal line. (c) The graph G'_π . (d) The graph $G''_\pi = G'_\pi$, embedded so that the face defined by π and π' is the outer face. (e) The graph G_π obtained after identifying the path π' (respectively π) in G''_π with the path π (respectively π') in G'_π .

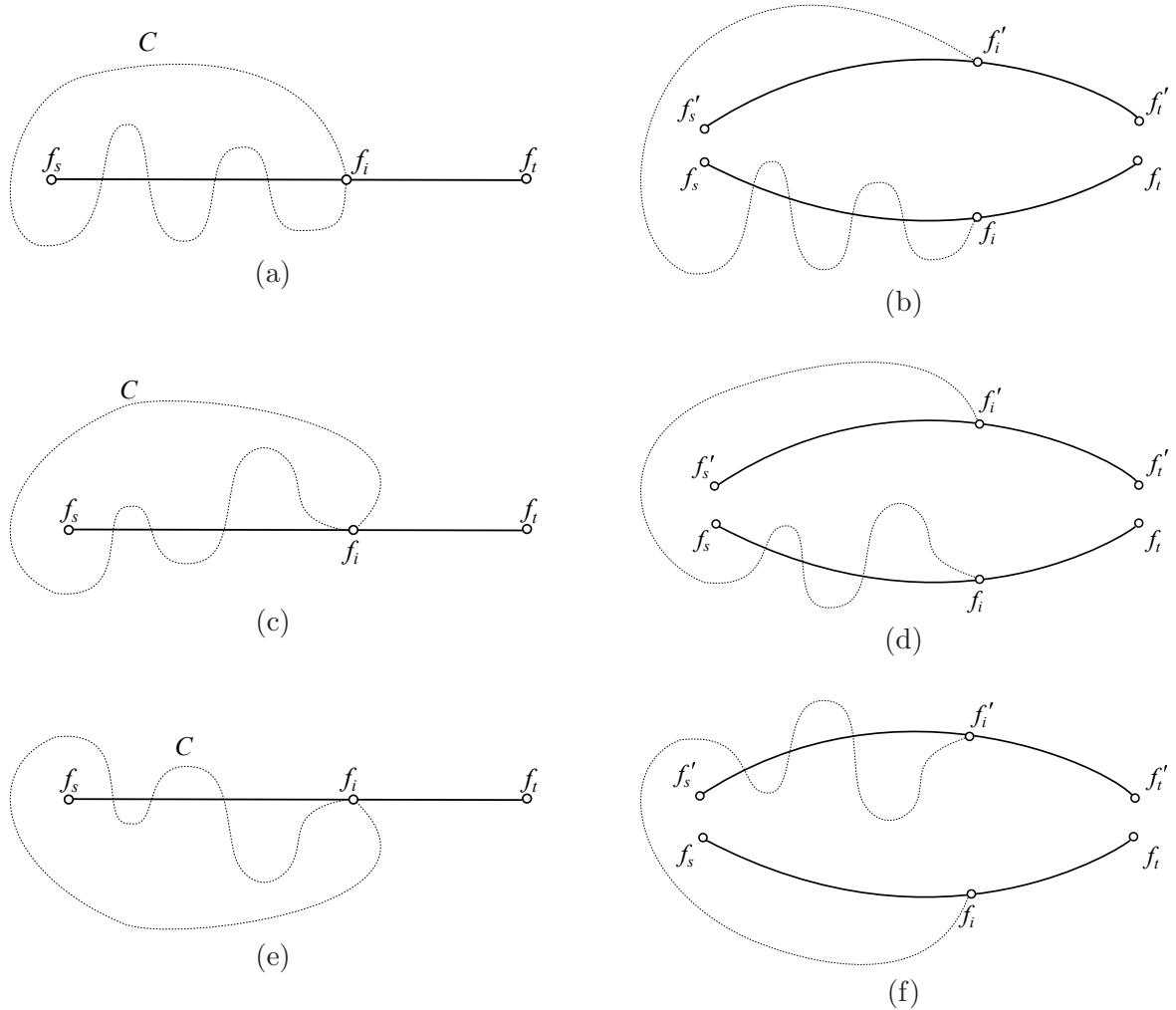


Figure 3: On the proof of Lemma 3. (a) The f_i -cut-cycle C crosses the path π in f_i in the graph G_D . (b) The corresponding f_i -separating path ρ in G_π . (c) The f_i -cut-cycle C touches from above the path π in f_i in the graph G_D . (d) The corresponding f_i -separating path ρ in G_π . (e) The f_i -cut-cycle C touches from below the path π in f_i in the graph G_D . (f) The corresponding f_i -separating path ρ in G_π .

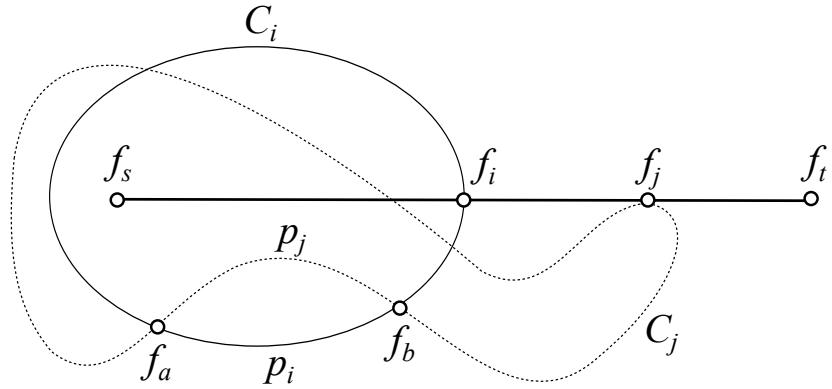


Figure 4: On the proof of Lemma 4.

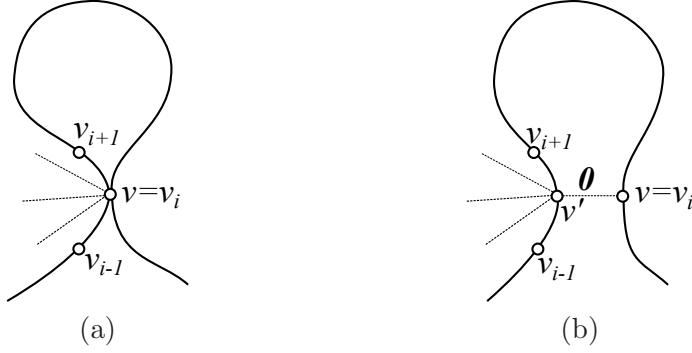


Figure 5: Dealing with non-simple paths. (a) A non-crossing path π of G_D that touches itself at face $v = v_i$. (b) The vertex splitting transformation on face v .

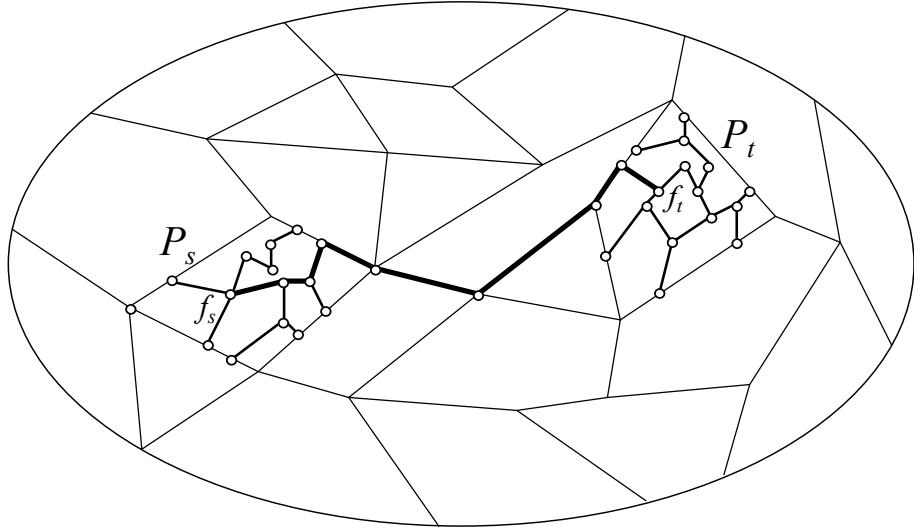


Figure 6: The graph $\overline{G}^{s,t}$ containing clusters P_s and P_t , the dense distance graphs for all other clusters, and the skeleton graph G_P . The path π from f_s to f_t is shown in bold.

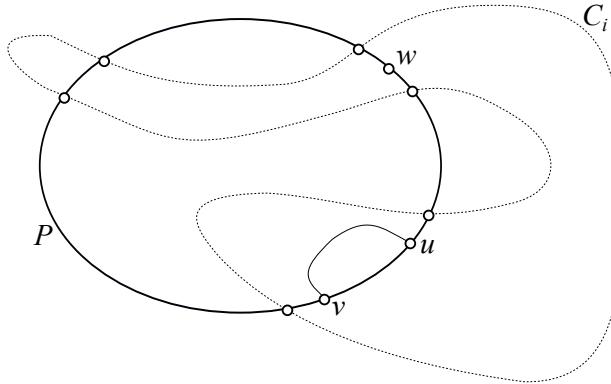
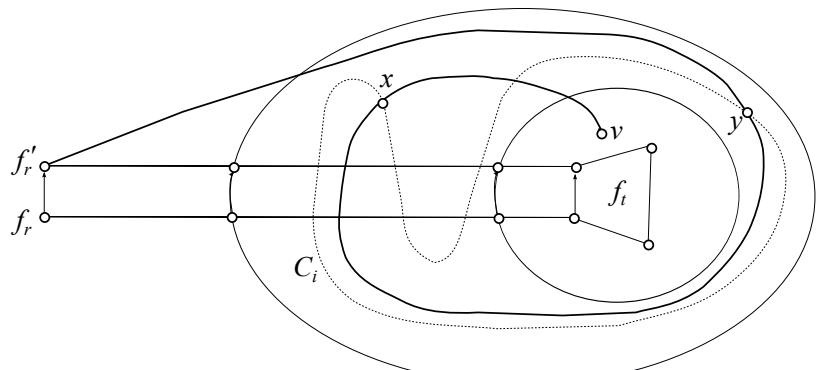
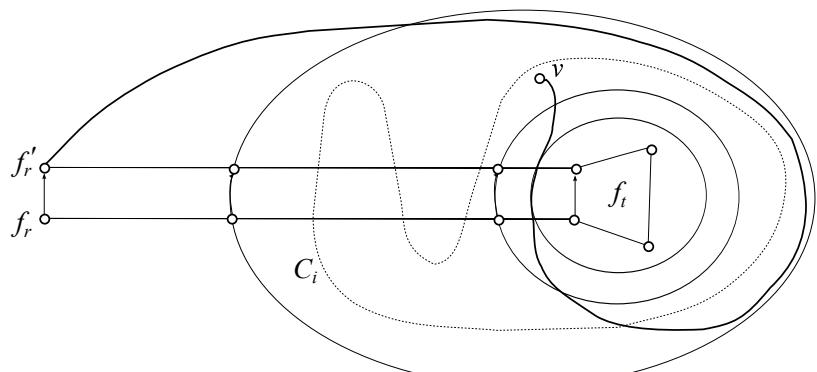


Figure 7: Computing $G_P \cap \text{int}(C_i)$. If two border vertices u and v are not separated by the f_i -cut-cycle C_i , then their distance in $G_P \cap \text{int}(C_i)$ is equal to their distance in G_P . On the other hand, if border vertices v and w are separated by the f_i -cut-cycle C_i , then their distance in $G_P \cap \text{int}(C_i)$ is infinite.



(a)



(b)

Figure 8: On the proof of Lemma 18. (a) The cycle C_i can be short cut using the subpath between x and y in $\rho(v)$. (b) A valid normal path.

A Computation of r -Partition: Proof of Lemma 7

In order to prove Lemma 7 we will combine Frederickson's [5] technique with the following implication of Section 3.1 and Section 5.1 from [2].

Corollary 22. *An r -partition of an n -node planar graph that already contains $O(\frac{n}{\sqrt{r}})$ border vertices can be computed in $O(n \log n)$ time.*

Proof. Actually, Fakcharoenphol and Rao [2] construct in the above time bound a recursive decomposition and in order to get an r -partition we only need to run their algorithm until the size of the clusters drops below $O(r)$.

Moreover, the recursive decomposition of Fakcharoenphol and Rao [2] assumes that border vertices are present in the decomposed graphs or clusters. Moreover, each time a cluster is split into smaller clusters, the border vertices are split into asymptotically equal parts. The $O(\frac{n}{\sqrt{r}})$ border vertices we start with will be distributed equally into $O(\frac{n}{r})$ clusters. Hence, each cluster in the obtained partition will have additional $O(\frac{n}{\sqrt{r}} \times \frac{r}{n}) = O(\sqrt{r})$ border vertices. \square

Now in order to find the r -partition quickly we use the following algorithm:

- Generate a spanning tree T of the graph;
- Find connected subtrees of T containing $O(\sqrt{r})$ vertices using a procedure from [5];
- Contract the graph on these subsets, to obtain a simple planar graph G_s with $O(\frac{n}{\sqrt{r}})$ vertices;
- Using Corollary 22, find an r -division in G_s with $O(\frac{n}{r^{3/2}})$ clusters of size $O(r)$;
- Expand G_s back to G . In G there are $O(n/r)$ clusters \mathcal{P}_1 of size $O(\sqrt{r})$ resulting from boundary vertices in G_s , and $O(\frac{n}{r^{3/2}})$ clusters \mathcal{P}_2 of size $O(r^{3/2})$ resulting from the interior vertices of G_s ;
- Apply Corollary 22 to find an r -division for clusters in \mathcal{P}_2 taking into account the border vertices already present in the clusters.

It is easy to see that the above procedure requires $O(n \log r + \frac{n}{\sqrt{r}} \log n)$ time. We only need to show that the result is a valid r -division. The clusters \mathcal{P}_1 correspond exactly to the connected subtrees of T of size $O(\sqrt{r})$. They cannot have more than $O(\sqrt{r})$ border vertices and all the border vertices lie on the external face, i.e., they contain one hole. Hence, clusters in \mathcal{P}_1 satisfy the properties of an r -partition.

Consider the clusters in \mathcal{P}_2 before we have applied Corollary 22. Each cluster is obtained by expanding a cluster in G_s that had \sqrt{r} border vertices. Consider the process of expanding to a subtree T a border vertex b_h lying on a hole h in a cluster P . Let h' be the hole obtained from h by removing b_h from P . The edges of T were not present in the cluster so the process of expanding T can be seen as gluing T at the side of h' . Note that by the connectivity of T no new hole is created and only the hole h' becomes "smaller". Moreover, not all of the vertices of T become border vertices of the expanded piece, as some of them do not lie on the side of the new hole (see Figure 9). Nevertheless as T contains $O(\sqrt{r})$ vertices at most $O(\sqrt{r})$ vertices can become border vertices. Hence, in total the pieces in \mathcal{P}_2 will have $O(r)$ border vertices. This number satisfies the assumption of Corollary 22, so pieces obtained by using it satisfy the assumption of r -partition. This completes the proof of Lemma 7.

B Computation of Dense Distance Graphs: Proof of Lemma 8

Proof. In order to compute dense distance graphs we use the following result by Klein [11].

Theorem 23 (Klein [11]). *Given an n -node planar graph with non-negative edge lengths, it takes $O(n \log n)$ time to construct a data structure that supports queries of the following form in $O(\log n)$*

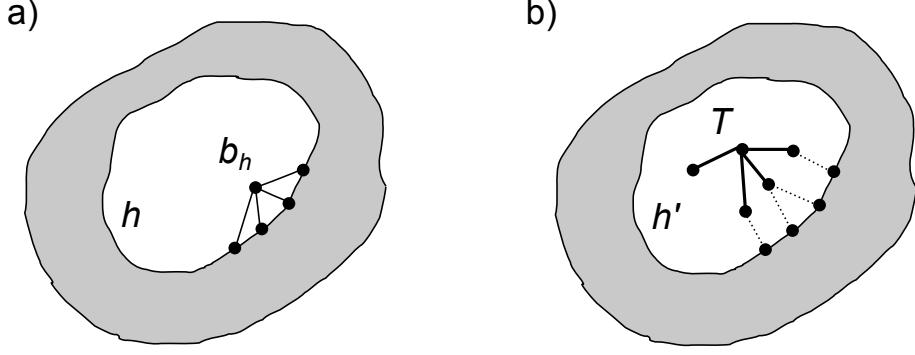


Figure 9: a) Before expanding the border vertex b_h in hole h ; b) after expanding b_h to T no new hole is created.

time: given a destination vertex t on the boundary of the external face, and given a start vertex s anywhere, find the distance from s to t .

A cluster C in the r -partition has r vertices, $O(\sqrt{r})$ border vertices and a constant number of holes. Border vertices in C lie on one of the holes. In order to compute distances from all border vertices to a given hole H we will apply Theorem 23. We simply find an embedding of the graph such that H becomes the external face, and query the distances from all border vertices to H . There are a constant number of holes in each cluster, and so Klein’s data structure will be used a constant number of times for each cluster. On the other hand, there are $O(r)$ pairs of border vertices in each cluster, so we will ask $O(r)$ queries. The time needed to process each cluster is hence $O(r \log r)$, which gives $\frac{n}{r} \times O(r \log r) = O(n \log r)$ time in total. \square

C Size of Contracted Graphs: Proof of Lemma 12

Proof. At the top level of the recursion we have one graph $\overline{G}_c^{s,t}$ that has $O(r + \frac{n}{\sqrt{r}})$ vertices and $O(r + \frac{n}{\sqrt{r}})$ faces. Now consider the case when the graph $\overline{G}_c^{s,t}$ is split along C_i into $\overline{G}^{<i}$ and $\overline{G}^{>i}$. Observe that the cycle C_i can be traced in $\overline{G}_c^{s,t}$ when we replace each edge in dense distance graphs for a cluster P by a two-edge path going through the center vertex v_P . In such a case we have $\overline{G}_c^{<i} = \overline{G}_c^{s,t} \cap \text{int}(C_i)$ and $\overline{G}_c^{>i} = \overline{G}_c^{s,t} \cap \text{ext}(C_i)$. In other words each time we recurse we split the graph $\overline{G}_c^{s,t}$ along some cycle and then remove degree-two vertices. Note that after splitting the graph $\overline{G}_c^{s,t}$ along the cycle C_i , the number of faces in the union of the resulting graphs increases by exactly one (see Figure 10). There are at most $O(r + \frac{n}{\sqrt{r}})$ recursive calls, so the union all contracted graphs on a given level in the recursive tree has $O(r + \frac{n}{\sqrt{r}})$ faces as well. Moreover, the vertices in this union graph have degree at least three, because all degree-two vertices are removed. Let v , e and f be the number of vertices, edges and respectively faces in the union graph. Now, by Euler’s formula the claim of the lemma follows: $v = 3v - 2v = 3v - 2(2+e-f) \leq 2e - 2(2+e-f) \leq 2f - 4 = O(r + \frac{n}{\sqrt{r}})$. \square

D General Clusters

When holes are present in the r -partition the skeleton graph G_P is not connected. Hence, the path connecting b_s and b_t cannot use border vertices only. We need to modify the algorithm to allow such paths. We do this as follows.

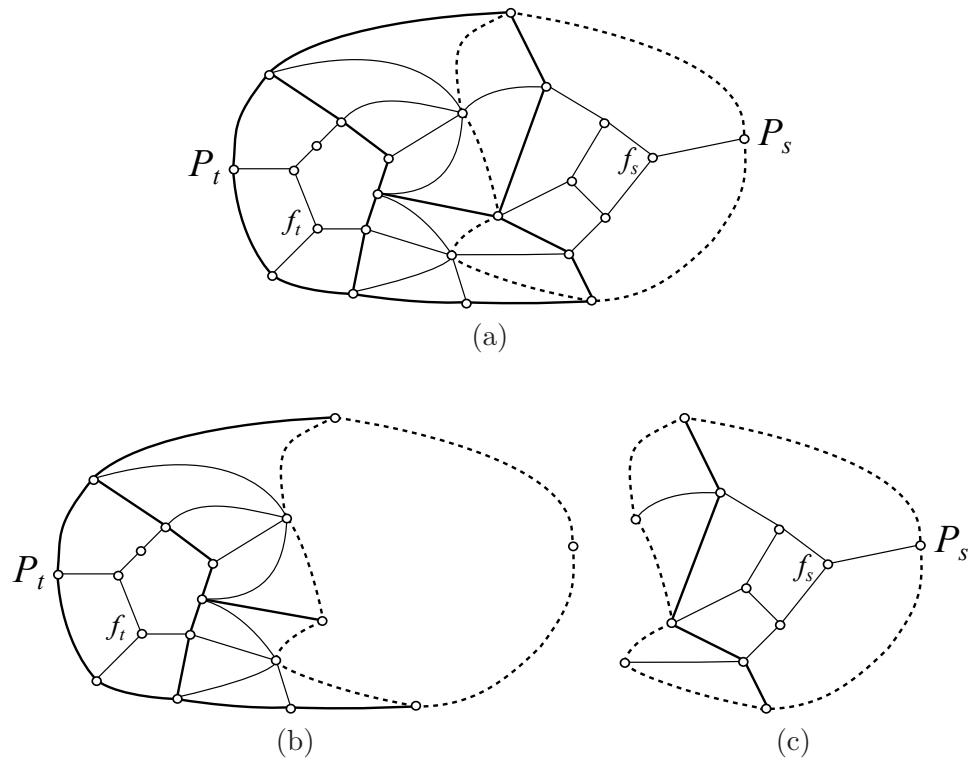


Figure 10: (a) The graph $\overline{G}_c^{s,t}$ and the cycle C_i . Cluster boundaries are shown in bold, and the cycle C_i is shown with dashed edges. (b) The graph $\overline{G}_c^{i>} = \overline{G}_c^{s,t} \cap \text{ext}(C_i)$. (c) The graph $\overline{G}_c^{i<} = \overline{G}_c^{s,t} \cap \text{int}(C_i)$. Note that the union of $\overline{G}_c^{i>}$ and $\overline{G}_c^{i<}$ contains one more face than $\overline{G}_c^{s,t}$.

For each hole h in P , we fix a border vertex b_h . For each pair of holes h, h' in P , we fix a path $\pi_{h,h'}$ that starts from b_h and ends in $b_{h'}$, goes through $b_{h''}$ for all holes h'' in P , and for all $b_{h''}$ on the path walks around the hole h'' passing through all its border vertices (see Figure 11). These paths are used to do some additional preprocessing for each cluster P in the partition. For each pair of holes h, h' in P we compute dense distance graph for $P_{\pi_{h,h'}}$, and find the minimum cut $C_{h,h'}$ between b_h and $b_{h'}$ in P .

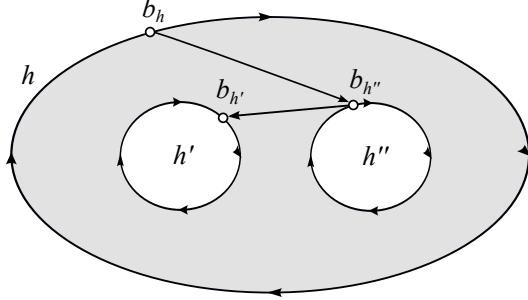


Figure 11: The path $\pi_{h,h'}$.

Corollary 24. *The additional preprocessing takes $O(n \log r)$ time.*

Proof. For each cluster P and each pair of holes the dense distance graph can be computed in the same manner as in Lemma 8. On the other hand, minimum-cuts can be found in $O(r \log r)$ time using the algorithm by Henzinger *et al.* [8]. Hence, over all clusters we need a total of $O(n \log r)$ time. \square

Now in order to connect f_s and f_t we will use paths $\pi_{h,h'}$ whenever we need to pass between two different holes h, h' in a piece P . Let \mathcal{P}_π be the set of all such pieces on π . The resulting path is no longer simple, but it will be non-crossing. As shown in Section 2.2, our min-cut algorithm can be executed on non-crossing paths as well. We can make the following observation (see Figure 12).

Corollary 25. *A minimum $s-t$ cut C either contains a vertex in $\partial\pi$ or is fully contained in one of the pieces \mathcal{P}_π .*

Proof. If the cycle C contains a vertex in $\partial\pi$, we are done. Assume that it does not contain any vertex from $\partial\pi$. In order to be a cut, it has to cross path π and it can do it so in one of $\pi_{h,h'}$. Then it has to be fully contained in the corresponding P as by the construction of $\pi_{h,h'}$ or border vertices of P lie on π . \square

Using Corollary 25, we can find the minimum cut in two phases. First, let C_i be the smallest of the cuts $C_{h,h'}$ in \mathcal{P}_π for $b_h, b'_h \in \pi$. Second, run the algorithm from previous section on a path $\partial\pi$ in $\overline{G}^{s,t}$ to find a cut C_b . Finally, return the smallest of the cuts C_i and C_b . The running time of the above algorithm is the same as in Theorem 13.

Theorem 26. *Let G be a flow network with source s and sink t . The minimum cut between s and t in G can be computed in $O(n \log r + (r + \frac{n}{\sqrt{r}}) \log^3 n)$ time. By setting $r = \log^8 n$, we obtain an $O(n \log \log n)$ time algorithm.*

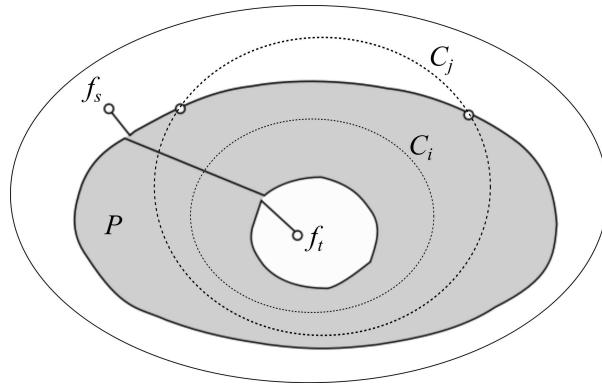


Figure 12: On illustrating Corollary 25. A cut can be either fully contained in one of the pieces \mathcal{P}_π (such as cut C_i) or it must contain one of the border vertices in $\partial\pi$ (such as cut C_j).