1986

Fast Algorithms for Shortest Paths in Planar Graphs, with Applications

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

Report Number:
84-486

TR # 486

FAST ALGORITHMS FOR SHORTEST PATHS IN PLANAR GRAPHS, WITH APPLICATIONS*

Greg N. Frederickson
April 1986

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

* This research was supported in part by the National Science Foundation under Grants MCS-8201083 and DCR-8320124.

Abstract. Graph decomposition and data structures techniques are presented that exploit the structure of planar graphs to yield faster algorithms for a number of shortest path and related problems. Improved algorithms are presented for the single source problem, the all pairs problem, the problem of finding a minimum cut in an undirected graph, and testing the feasibility of a multicommodity flow when all sources and sinks are on the same face. The algorithm for the single source takes $O(n\sqrt{\log n})$ time in an n-vertex graph, an improvement from $O(n \log n)$. The algorithm for all pairs takes $O(n^2)$ time, an improvement from $O(n^2 \log n)$. The algorithm for minimum cut takes $O(n \log n)$ time, an improvement from $O(n(\log n)^2)$. As a consequence, an algorithm for maximum flow is similarly improved.

Key words and phrases. all pairs shortest paths, decision trees, heaps, maximum flow, minimum cut, multicommodity flow, planar graph, planar separator, single source shortest paths.

1. Introduction

Algorithms that are efficient on sparse graphs will do well on planar graphs, since the number of edges in an n-vertex planar graph is $O(n)$. But in what ways can the planarity of a graph be exploited, beyond taking advantage merely of the sparsity? We examine this question by considering the problem of finding shortest paths in planar graphs. We present algorithms faster than those previously known for solving the single source shortest paths, the all pairs shortest paths, and several related problems. The work highlights the effect that planarity has on issues of not only planar graph decomposition but also data structures.

Finding shortest paths is a fundamental and well-studied problem in applied graph theory [AHU, DP]. For the single source problem on a directed or undirected graph with nonnegative edge costs, Dijkstra's algorithm takes $O(n^2)$ time [D]. An implementation of Dijkstra's algorithm that uses a heap improves this to $O(n \log n)$ time for any graph with $O(n)$ edges [J]. Our algorithm for the single source problem in a planar graph uses $O(n\sqrt{\log n})$ time. For the all pairs problem on a undirected graph with nonnegative edge costs, the best previous algorithm runs the heap version of Dijkstra's algorithm n times, each time with a different vertex as the source, and thus uses $O(n^2 \log n)$ time altogether on a planar graph. Using results of [LRT] and [EK, J, Ne, Ni], this strategy can be extended to handle directed graphs with negative cost edges but no negative cycles in the same time bound. We present an algorithm for both variants of the all pairs problem that uses $O(n^2)$ time. This result is optimal, if the output is required to be in the straightforward form.

The best previous algorithms for finding an s-t minimum cut [R] and an s-t maximum flow [HJ] in an undirected planar graph use a number of shortest path computations. We show how to adapt our shortest path techniques to improve these algorithms from $O(n(\log n)^2)$ to $O(n \log n)$ time. Algorithms for determining feasibility in a multicommodity flow problem in which all terminals are on the same face of a planar graph have been presented in [Hs2, MNS]. We show how to adapt our techniques to yield a faster feasibility test.

Our algorithms are based on Dijkstra's single source algorithm, which we now describe briefly. Dijkstra's algorithm performs a search of the graph that proceeds in iterations. For each vertex v whose shortest distance $d(v)$ from the source s is not known, the vertex is termed *open,* and the currently known shortest distance $\rho(v)$ from the source to v is maintained. Initially, all vertices are open, $\rho(s) = 0$, and $\rho(v) = \infty$ for all other vertices v. On each iteration, the open vertex v with minimum $\rho(v)$ is closed, and the shortest distances $\rho(w)$ are updated for all w such that there is an edge (v,w). The distances $\rho(v)$ can be maintained in a heap, giving $O(\log n)$ time per update. Since there are $O(n)$ updates in handling a planar graph, the total time is $O(n \log n)$.

We improve on Dijkstra's algorithm by conducting the iterative search on a carefully selected subset of the vertices. This means that a *preprocessing phase* is necessary to identify this subset of vertices. This phase must also determine shortest paths between all pairs of vertices in this subset, where intermediate vertices on these paths are not in the subset. The *search phase* consists of two parts. During the *main thrust* of the search phase, when a vertex v in the special subset is closed, $\rho(w)$ must be updated for all vertices w in the subset such that a path of the above type exists from v to w. At the end of

the main thrust, the shortest distances are known from the source to each vertex in the subset. The *mop-up* portion of the search phase then determines the shortest distance to every remaining vertex.

For the main thrust of the search phase to be efficient, the vertices in the subset must separate the graph into a number of regions of convenient size and favorable adjacency properties. A planar separator algorithm [LT1] can be used in separating a planar graph into regions. However, a straightforward use of this algorithm is not adequate to generate regions with appropriate characteristics. We thus contribute interesting results for planar graph decomposition.

A second idea that makes the main thrust of the search phase efficient involves the design of appropriate data structures. When a vertex v is closed, many distances $\rho(w)$ may need to be updated in the heap. It turns out that the total number of updates performed during the search phase is in worst case at least proportional to the number performed in Dijkstra's algorithm. Fortunately however, the updates involve the same region. We present a heap whose organization is based on the adjacency of regions and thus allows a batch of related updates to be handled more efficiently.

Our algorithms for all pairs shortest paths and minimum cut in a planar graph rely in addition on performing extensive preprocessing of the graph, that allows for a number of very fast searches. The preprocessing constructs decision trees to identify the portions of a shortest path tree within each of many very small regions. The decision trees implement a divide-and-conquer approach that splits a region based on shortest paths to a set of separator vertices.

The paper is organized as follows. In sections 2 and 3 we show how to divide the graph into regions and identify the special subset of vertices on which to perform the main thrust. In section 4 we present an efficient data structure for performing the main thrust. Section 5 gives the single source algorithm. Sections 6 and 7 describe one-time preprocessing and the decision trees that allow for very efficient searches. The improvements to finding minimum cuts, maximum flows, and multicommodity flows are discussed in sections 8 and 9.

A preliminary version of this paper appeared in [F2].

2. Regions and boundary vertices

Our shortest path algorithms make use of a division of the planar graph into regions. A *region* will contain two types of vertices, boundary vertices and interior vertices. An *interior vertex* is contained in exactly one region, and is adjacent only to vertices contained in that region, while a *boundary vertex* is shared among at least two regions. An example of a graph which has been divided into regions is shown in Figure 1. The boundary vertices are shown in bold, and the regions are circled. To generate appropriate regions, we make use a linear-time algorithm of Lipton and Tarjan that finds a planar separator [LT1]. Let the vertices have nonnegative vertex costs summing to no more than one. The separator algorithm partitions the vertices of G into three sets A, B, and C such that no edge joins a vertex in A with a vertex in B, neither A nor B has total cost exceeding 2/3, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices.

Given a parameter r, we first discuss how to generate $\Theta(n/r)$ regions with $O(r)$

vertices each, and $O(n/\sqrt{r})$ boundary vertices in total. Initially, G consists of one region with all vertices interior. Apply the separator algorithm to the graph with all vertex weights equal to $1/n$, yielding sets A, B and C. Infer two regions with vertex sets $A_1 \subseteq A \cup C$ and $A_2 \subseteq B \cup C$, of sizes $\alpha n + O(\sqrt{n})$ and $(1-\alpha)n + O(\sqrt{n})$, where $1/3 \leq \alpha \leq 2/3$. Recursively apply the procedure on the subgraph induced by any region with more than r vertices. The total time required will be $O(n \log(n/r))$. This approach is similar to that described in [LT2].

Lemma 1. An n-vertex planar graph can be divided into $\Theta(n/r)$ regions with no more than r vertices each, and $O(n/\sqrt{r})$ boundary vertices in total.

Proof. Let the above method be applied to a graph. For any boundary vertex v, let $b(v)$ be one less than the number of regions that contain v in the division. Let $B(n, r)$ be the total of $b(v)$ over all boundary vertices v. Thus $B(n, r)$ is the sum of the number of boundary vertices v weighted by the count $b(v)$. From the above discussion, we have the following recurrence:

$$B(n, r) \leq c\sqrt{n} + B(\alpha n + O(\sqrt{n}), r) + B((1-\alpha)n + O(\sqrt{n}), r) \qquad \text{for } n > r$$

$$B(n, r) = 0 \qquad \text{for } n \leq r$$

where $c = 2\sqrt{2}$ and $1/3 \leq \alpha \leq 2/3$. Then we claim that

$$B(n, r) \leq cn/\sqrt{r} - d\sqrt{n}$$

for some constant d. The claim can be shown by induction. □

An *r-division* is a division into $\Theta(n/r)$ regions of $O(r)$ vertices each and $O(\sqrt{r})$ boundary vertices each. An r-division may be obtained from the preceding division in the following way. While there is a region that has more than $c\sqrt{r}$ boundary vertices, for

some constant c, apply the planar separator algorithm to the region, with the n' boundary vertices each having weight $1/n'$, and interior vertices having weight zero. Infer the two resulting regions as before.

Lemma 2. A planar graph of n vertices can be divided into an r-division in $O(n \log n)$ time.

Proof. We claim that the above method generates the desired division. Consider a division before regions are split to enforce the requirement of $O(\sqrt{r})$ boundary vertices per region. Let t_i be the number of regions with i boundary vertices. From the proof of the previous lemma, note that $\sum_i it_i = \sum_{v \in V_B}(b(v)+1)$, where V_B is the set of boundary vertices. It follows that $\sum_{v \in V_B}(b(v)+1) < 2B(n, r)$, which is $O(n/\sqrt{r})$. For a region with i boundary vertices, where $i > c\sqrt{r}$, at most $di/(c\sqrt{r})$ splits need be done, for some constants c and d. This will result in at most $1+di/(c\sqrt{r})$ regions, and at most $c\sqrt{r}$ new boundary vertices per split. Thus the number of new boundary vertices is at most

$$\sum_i (c\sqrt{r})(di/(c\sqrt{r}))t_i \leq d \sum_i it_i = O(n/\sqrt{r})$$

The number of new regions will be at most

$$\sum_i (di/(c\sqrt{r}))t_i = O(n/r). \qquad \square$$

It is convenient to have no boundary vertex be in more than some constant number of regions. Thus before any other preprocessing, we transform the initial planar graph G_0 into planar graph G with no vertex having degree greater than 3. A well-known transformation in graph theory [Hr, p. 132] may be used to generate G from G_0. Consider a planar embedding of G_0. For each vertex v of degree $d > 3$, where

$w_0, \cdots, w_{d-1}$ is a cyclic ordering of the vertices adjacent to v in the planar embedding, replace v with new vertices $v_0, \cdots, v_{d-1}$. Add edges $\{(v_i, v_{(i+1) mod\ d}) \mid i=0, \cdots, d-1\}$, each of distance 0, and replace the edges $\{(w_i, v) \mid i=0, \cdots, d-1\}$ with $\{(w_i, v_i) \mid i=0, \cdots, d-1\}$, of corresponding distances. From a corollary of Euler's formula [Hr], the number of vertices in the resulting graph will be $n \leq 6n_0-12$, where n_0 is the number of vertices in G_0.

3. Suitable graph divisions

To facilitate efficient execution of heap updates in the search phase of our algorithm, it is helpful to have each region share boundary vertices with relatively few other regions. Two problems arise with the division as generated in the previous section. First, a boundary vertex may be in a large number of regions, even though its degree is limited to three. Second, the regions generated by the previous method are not necessarily connected, so that the effect of locality provided by the planarity of the graph may be lost. A *suitable r-division* of a planar graph is an r-division such that

1. each boundary vertex is contained in at most three regions, and

2. any region that is not connected consists of connected components, all of which share boundary vertices with exactly the same set of either one or two connected regions.

The following strategy ensures that no boundary vertex is in more than three regions. Consider an application of the planar separator algorithm, which yields sets A, B and C. Let C' be the set of vertices in C not adjacent to any vertex in $A \cup B$, and let $C'' = C - C'$. Identify the connected components $A_1, A_2, \cdots, A_q$ in $A \cup B \cup C'$.

Any vertex v in C'' adjacent to a vertex in A_i and not adjacent to a vertex in A_j for $j \neq i$, can be removed from C'' and inserted it into A_i. We term the resulting subgraphs *connected subgraphs*. Connected subgraph i will have interior vertices A_i and boundary vertices in C'' that are adjacent to some vertex in A_i. The time to find and augment the connected subgraphs will be linear. A boundary vertex will be in at most three connected subgraphs, since the degree is at most three, and a vertex is included as a boundary vertex of a connected subgraph only if it is adjacent to an interior vertex in the connected subgraph.

More than $\Theta(n/r)$ connected subgraphs can result when the above strategy is applied recursively to each connected subgraph with more than r vertices. To generate $\Theta(n/r)$ regions from the set of connected sugraphs, do the following in a greedy fashion. Initialize each region to be a connected subgraph. While there are two regions that share a common boundary vertex and each has no more than $r/2$ vertices, union the regions together. It is still possible that there may be more than $\Theta(n/r)$ regions. While there are two regions that each have no more than $r/2$ vertices, and are adjacent to the same set of either one or two regions, combine them. This procedure can be performed in linear time.

The resulting division will have each region being either connected or the union of connected subgraphs which share boundary vertices with the same set of regions. Examples of regions that are unions of connected subgraphs are shown in Figure 2. Each component of region C is adjacent only to region A, and each component of region D is adjacent to both region A and region B.

Of course, it is also required that no region have more than $c\sqrt{r}$ boundary vertices, for some constant c. However this constraint is not a problem. Before initializing the regions to be the connected subgraphs, apply the separator strategy to any connected subgraph with more than $c\sqrt{r}$ boundary vertices. The regions are then initialized to be the resulting subgraphs, and combined in a fashion similar to that described above. But now a region can be combined with another if it has at most $r/2$ vertices and at most $c\sqrt{r}/2$ boundary vertices.

Theorem 1. A planar graph of n vertices can be divided into a suitable r-division in $O(n \log n)$ time.

Proof. It is clear that the above strategy generates regions with at most r vertices and $c\sqrt{r}$ boundary vertices. We must establish that there will be $\Theta(n/r)$ regions in the division. Consider a graph, called the *region graph,* in which there is a node for each region in the division, and an edge between two nodes if the corresponding regions share a boundary. The region graph is not explicitly constructed, but rather is a device for counting regions in the division. Call a node *small* if it represents a region that has at most $r/2$ vertices and at most $c\sqrt{r}/2$ boundary vertices, and *normal* otherwise. Since there are $O(n)$ vertices and $O(n/\sqrt{r})$ boundary vertices in total, there can be no more than $O(n/r)$ normal nodes.

From the foregoing procedure, the following properties hold. Each small node is adjacent only to normal nodes. There can be no more than one small node of degree 1 adjacent to each normal node. There can be no more than one small node of degree 2 adjacent to any pair of normal nodes. Thus the number of small nodes of degree 2 is no

greater than the number of edges in the reduced graph with each such small node and its incident edges replaced by one edge. Hence the number of small nodes of degree 2 is in worst case proportional to the number of normal nodes. Consider the reduced graph with small nodes of degree one and two removed, and consider a planar embedding. Add as many edges as possible between normal nodes, and then delete the edges incident on small nodes. There will be at most one of these small nodes per planar face, of which there can be at most $O(n/r)$. Thus there are at most $O(n/r)$ small nodes of all degrees in the reduced graph.

The time bound follows, since the time to apply the separator strategy recursively is $O(n \log n)$, and the time to perform the combining is $O(n)$. □

4. A topology-based heap and the batched update operation

We now show how to organize a heap based on a suitable r-division. Partition the boundary vertices into *boundary sets*, which are maximal subsets such that every member of a set is shared by exactly the same set of regions. The boundary sets corresponding to the boundary vertices in Figure 1 are circled in Figure 3a. Every boundary set contained in exactly two regions will correspond to an edge in the region graph discussed in the proof of Theorem 1. As for boundary sets contained in three regions, no region can contain more such boundary sets than there are edges incident on its node in the region graph. Since the region graph is planar, and there are $\Theta(n/r)$ nodes in it, there are $\Theta(n/r)$ edges. Thus there are $\Theta(n/r)$ boundary sets.

The boundary sets can be identified as follows. Assume that the regions are

indexed from 1 up to the number of regions. For each boundary vertex, generate an ordered list of the either two or three regions that the vertex is in. Now sort the set of lists, using a lexicographic sort. All boundary vertices in the same boundary set will have their lists appear in consecutive order as a result of the sort. The time to identify the sets will be proportional to the number of boundary vertices.

A *topology-based heap on boundary vertices* is a heap represented by a balanced tree in which the values associated with the boundary vertices appear in the leaves, with values from any one boundary set being in consecutive leaves. A topology-based heap corresponding to the boundary sets of Figure 3a is shown in Figure 3b. The sets of leaves that are circled correspond to the boundary sets. Note that the boundary vertices can be associated with the leaves in precisely the order generated by the previous lexicographic sort. Thus a topology-based heap can be set up in $O(n/\sqrt{r})$ time.

A *batched update* is an operation that updates in the topology-based heap the values associated with all vertices of a boundary set. In the search phase of our algorithm, whenever some vertex is closed, each region containing the vertex will have a batched update performed for each of its boundary sets. To perform a batched update, do the following. Modify the values at all leaves corresponding to vertices in the boundary set. Then proceed to modify the ancestors of these leaves level by level moving upward: first parents of the leaves, then grandparents, etc., until the value at the root is modified.

The number of nodes in the heap that are modified by a batched update is less than 2 log n plus twice the size of the boundary set. This is established by the following reasoning. The number of interior nodes, all of whose leaf descendants correspond to

vertices in the boundary set, is less than the size of the boundary set. The only other interior nodes that are examined have some leaf descendants that correspond to vertices in the boundary set, and some that do not. There are at most two such interior nodes per level.

Theorem 2. Let an n-vertex planar graph be divided into a suitable r-division. Using the associated topology-based heap, the main thrust of our algorithm will perform a set of batched update operations which cost $O(n + (n/\sqrt{r})\log n)$.

Proof. Since there are at most $O(\sqrt{r})$ boundary vertices per region, and each boundary set is in at most 3 regions, each boundary set can have a batched update performed on it at most $O(\sqrt{r})$ times. Thus the total work involved in all batched updates for any one boundary set is proportional to $\sqrt{r}\log n$ plus $\sqrt{r}$ times the cardinality of the boundary set. Since there are $\Theta(n/r)$ boundary sets, the total cost of the first term over all boundary sets is $O((n/\sqrt{r})\log n)$. Since there are $O(n/\sqrt{r})$ boundary vertices, the total cost of the second term over all boundary sets is $O(n)$. □

We now describe how our strategy would work, if we are already given a suitable r-division of a graph, for a parameter r which we shall specify later. To perform the main thrust, we need, for each region, the shortest paths between every pair of boundary vertices. These paths can be found by performing, for each boundary vertex of a region, Dijkstra's algorithm within the region. Since there are $O(n/\sqrt{r})$ boundary vertices, and each run of Dijkstra's algorithm will use $O(r \log r)$ time, the time required to find these paths will be $O(n\sqrt{r} \log r)$.

Given this preprocessing, the main thrust of the algorithm follows our previous description, using the topology-based heap, and performing batched update operations in it. At termination of the main thrust, shortest paths from the source to each boundary vertex will be known. Shortest distances to other vertices may be found by performing mop-up in each region separately. Dijkstra's algorithm can be used in each region, starting the search from the set of boundary vertices, labeled with the shortest distances to them. We choose $r = (\log n)/(\log\log n)$, so as to balance the times for the main thrust and the preprocessing.

Lemma 3. Let an n-vertex planar graph be divided into a suitable r-division, where $r = (\log n)/(\log\log n)$. A single source shortest path tree can be found in $O(n\sqrt{\log n}\sqrt{\log\log n})$ time.

Proof. As claimed in Theorem 2, the time to perform the main thrust will be $O(n + (n/\sqrt{r})\log n)$. The mop-up will take $O(r \log r)$ time for each of $\Theta(n/r)$ regions, yielding $O(n \log r)$ time altogether. The result follows since $r = (\log n)/(\log\log n)$. □

5. A Fast Algorithm for the Single Source Problem

In this section we give an algorithm for finding a single source shortest path tree in a planar graph in $O(n\sqrt{\log n})$ time. To achieve this bound, we must show how to find a suitable r-division in $o(n \log n)$ time. We must also improve on the performance of the algorithm in the last section, as described in Lemma 3. We first discuss the latter problem.

A time-critical part of the computation is in the preprocessing, in which for each region the shortest paths between every pair of boundary vertices are found. Instead of using Dijkstra's algorithm to perform these searches within regions, we shall use our search strategy. This will of course require that suitable divisions of each region be found, so that the search can be performed.

We now describe our algorithm. First find a suitable r_1-division of the graph, where $r_1 = \log n$. (We shall describe later how to do this in $o(n \log n)$ time.) Call each region in this division a *level 1 region*. Then for each level 1 region find a suitable r_2-division, where $r_2 = (\text{loglog } n)^2$. Call each region in these divisions a *level 2 region*. When finding this division, start with each level 1 boundary vertex automatically being a level 2 boundary vertex. This will not cause more than $\Theta(n/\sqrt{r_2})$ boundary vertices of level 2 regions to be created. For each level 2 region, find the shortest paths between every pair of its boundary vertices. Dijkstra's algorithm should be used for this task, with the source being each boundary vertex of the region in turn. Then for each level 1 region, find the shortest paths between every pair of its boundary vertices. The main thrust of our search phase, described earlier, should be used. This concludes the preprocessing. Having found, for each level 1 region, the shortest distances between its boundary vertices, we then perform the search phase on the graph. The main thrust will yield a shortest path tree encoding the shortest paths to each level 1 boundary vertex in the graph. The mop-up phase can then be performed by labeling boundary vertices with the shortest distances to them, and then using Dijkstra's algorithm within each region.

We analyze the time for this algorithm, exclusive of the time to find the r_1-

division. The time required to find an r_2-division of a region of size r_1 is $O(r_1 \log r_1)$. Summing over the $\Theta(n/r_1)$ regions gives a total time of $O(n \text{ loglog } n)$. Using Dijkstra's algorithm, the time to find a shortest path tree in a region of size at most r_2 is $O(r_2 \log r_2)$. Finding these shortest path trees for each of $O(n/\sqrt{r_2})$ level 2 boundary vertices will use $O(n \text{ loglog } n \text{ logloglog } n)$ time. Using our main thrust, the time to find a shortest path tree in a region of size at most r_1 is $O(r_1+(r_1/\sqrt{r_2})\log r_1)$, or $O(r_1)$. Note that no mop-up is necessary, since each level 1 boundary vertex was automatically designated a level 2 boundary vertex. Thus the total time for finding these shortest path trees for each of the $O(n/\sqrt{r_1})$ level 1 boundary vertices is $O(n\sqrt{r_1})$, or $O(n\sqrt{\log n})$. By Theorem 2, the main thrust within the graph will take time $O(n\sqrt{\log n})$. The mop-up for each of $\Theta(n/r_1)$ level 1 regions will take $O(r_1 \log r_1)$ time per region, or $O(n \text{ loglog } n)$ time total. Thus the total time for everything except finding the r_1-division will be $O(n\sqrt{\log n})$.

We now show how to generate a suitable r-division quickly. Find a spanning tree of the graph using depth-first search. Generate connected sets of $\Theta(\sqrt{r})$ vertices in a bottom-up fashion, using a procedure *FINDCLUSTERS* from [F1]. Now shrink the graph on these sets, yielding a planar graph G_s with $\Theta(n/\sqrt{r})$ vertices. Apply the planar separator strategy from section 3 to graph G_s to yield $\Theta(n/r^{3/2})$ regions of cardinality $O(r)$. Expand G_s back to G. In G there will be $O(n/r)$ regions of size $O(\sqrt{r})$ resulting from boundary vertices in G_s, plus $\Theta(n/r^{3/2})$ regions of cardinality $O(r^{3/2})$ resulting from the interior vertices of G_s. Infer boundary vertices and slightly expanded regions that share these vertices. Apply our procedures from section 3 on the regions to generate

a suitable r-division.

Lemma 4. A planar graph of n vertices may be divided into a suitable r-division in $O(n \log r + (n/\sqrt{r})\log n)$ time.

Proof. The connected sets and the graph G_s can be identified in $O(n)$ time. The time to generate a division of G_s will be $O(n(\log n)/\sqrt{r})$. The time to expand G_s back to G will be $O(n)$. The time to split regions of size $O(r^{3/2})$ will be $O(n \log r)$. □

Theorem 3. A single source shortest paths problem on an n-vertex planar graph with nonnegative edge costs can be solved in $O(n\sqrt{\log n})$ time. □

Our single source shortest path algorithm can be used to speed up an algorithm [Hs1, IS] for finding a maximum flow in an $s-t$ planar network. The dominating term in the running time of this algorithm is the time to solve one single source problem. Substituting our single source shortest path algorithm yields the following.

Corollary 1. A maximum flow in an $s-t$ planar network can be found in $O(n\sqrt{\log n})$ time. □

6. Decision trees for regions

As indicated in the introduction, there are several problems that can be solved by solving a number of single source problems. For such a problem it is not necessary to balance the preprocessing time against the search time of one shortest path computation, since the preprocessing need be done only once, while the search will be done many

times. In the next section, using the results of this section, we show how to realize $O(n)$ search time at a one-time expense of $O(n \log n)$ preprocessing time.

A part of the search will use decision trees, one for each of many small regions. For each of these regions, its decision tree will take as input the shortest distances from some source vertex in the graph to each of the boundary vertices of the region (plus some other information to be described subsequently). A leaf in the decision tree will identify all edges inside the region that are contained in a shortest path tree of the graph rooted at the source vertex. The height of the decision tree for any region, and thus the time to search it, will be proportional to the size of the region. The time to build the decision tree will be quite large in comparison to the size of the region. However, this technique will be applied in the next section to regions of very small size as a function of n.

We assume initially that the region is *nice*, i.e., that there is a planar embedding of the region such that all boundary vertices of the region are on the exterior face, and that a shortest path from the source to any boundary vertex is exterior to the region. In general, a region will not be nice. In the case when the region is not nice we show later in this section how to set up a decision tree that will identify a subset of the boundary vertices that are on an exterior face in some embedding, so that shortest paths can still be computed.

The decision tree that we construct for nice regions will work for nice regions that are in a *prepared form*. Each prepared region will consist of a source vertex, a set of vertices V^b, and a set of other vertices V^u. For each vertex v_j in V^b there is an edge from the source to v_j, and for each vertex v_k in V^u there is no edge from the source to v_k.

Every edge except those incident on the source has an edge cost supplied when the decision tree is built. For each vertex v_j in V^b, the cost of the edge from the source to v_j will be supplied as input when the decision tree algorithm is run. The input will satisfy the constraint that for each v_j in V^b, the edge from the source to v_j will represent a shortest path from the source to v_j that is exterior to the region.

We first discuss how to transform a nice region H into its prepared form. The boundary vertices of H will comprise the set V^b, and the interior vertices will comprise the set V^u. Introduce a source vertex and an edge from the source to each vertex v_j in V^b. The region from the lower right in the graph of Figure 1 is shown in Figure 4a. The corresponding region in prepared form, along with costs on its edges, is shown in Figure 4b.

We next discuss how shortest distances and shortest paths from the source to every vertex in H can be found efficiently. Let $m = |V^u|$ and $b = |V^b|$. If $m = 0$, then shortest paths are already known from the source to each vertex in H, and no comparisons are needed in its decision tree. If $b = 1$, let $V^b = \{v_1\}$. A shortest distance to each vertex x will be $d(source, x) = d(source, v_1) + d(v_1, x)$. A shortest path from the source to x must go from the source to v_1, and then follow a shortest path from v_1 to x. Since a shortest path from v_1 to x does not depend on the input to the decision tree, again no comparisons are needed in the decision tree for H.

If $b = 2$, let $V^b = \{v_1, v_2\}$. The shortest distance from the source to each vertex x will be the smaller of $d(source, v_1) + d(v_1, x)$ and $d(source, v_2) + d(v_2, x)$. Note that distances $d(v_1, x)$ and $d(v_2, x)$, and the shortest paths realizing these distances, do

not depend on the inputs to the decision tree, and can be computed when the decision tree is built. Thus for each vertex x of V^u, only one comparison is needed in the decision tree to determine which of v_1 and v_2 is on the shortest path from the source to x. This means that m comparisons involving input values are sufficient to determine the edges in H contained in a shortest path tree rooted at the source.

If $m>0$ and $b>2$, region H can be handled as follows. By the planar separator theorem of [LT1], there is a separator V^s of size at most $2\sqrt{2}\sqrt{m}$ for the subgraph induced by V^u. Let $s = |V^s|$. For each vertex v_k in V^s, a shortest distance from the source to v_k can be found by finding the smallest value $d(source, v_j) + d(v_j, v_k)$ over all vertices v_j in V^b. We assume that a tie between two vertices v_j and $v_{j'}$ in V^b is broken by choosing v_j if $j < j'$. The shortest path from the source to v_k will consist of the edge from the source to the vertex v_j realizing the minimum, followed by the shortest path from v_j to v_k. Since the latter shortest paths do not depend on the input to the decision tree, only $(b-1)s$ comparisons involving input values are necessary to find shortest paths from the source to all vertices in V^s. These $(b-1)s$ comparisons would be performed in the top $(b-1)s$ levels of the decision tree for the region, and would determine the set of shortest paths to all vertices in V^s. The remaining levels can be determined as follows.

Any set of paths realizing shortest distances to vertices in V^s divides H into some number of subregions, say t. Note that any vertex x in V^u which is on a shortest path from the source to some v_k in V^s has a shortest path from the source that is a prefix of the shortest path to v_k. Let C be the set of vertices in V^u that are are on a shortest path

from the source to a vertex in V^s. Every maximal connected subset in $V^u - C$ will be taken to be the interior vertices V_i^u of a subregion H_i. Any edge between vertices in V_i^u will remain in subregion H_i.

The set V_i^b is determined as follows. Vertex v_j in V^b is included in V_i^b if either v_j is adjacent to some vertex y in V_i^u, or the shortest path to some vertex v_k in V^s contains both v_j and a vertex x which is adjacent to some vertex y in V_i^u. In the former case, there will be an edge included in H_i from v_j to y. In the latter case, for any such pair v_j and y, there will be an edge (v_j, y) in H_i, of cost $d(v_j, x) + c(x, y)$ which is minimum over all such x.

With respect to the region in Figure 4b, a separator set V^s for the subgraph induced by V^u is shown in Figure 4c, along with shortest paths to vertices in V^s. These paths partition the region into three nice subregions, which are shown in prepared form in Figure 4d. For instance, the edge (v_6, y) in Figure 4d results from y being adjacent to vertex x, which is on the shortest path from v_6 to one of the separator vertices in Figure 4c. The cost 11 follows since $c(x, y) = 2$ and $d(v_6, x) = 6+3 = 9$. Similarly, edges (v_2, z) and (v_6, z) in Figure 4d result from z being adjacent to separator vertices that are claimed by v_2 and v_6 in Figure 4c. Their costs of 9 and 13, resp., represent the costs 6+1+2 and 6+3+2+2 of the corresponding paths.

Lemma 5. Consider the dividing of a region H into subregions $\{H_i\}$ based on paths as above. Let $b_i = |V_i^b|$. Let $I = \{i \mid b_i > 2\}$. Then $\sum_{i \in I} (b_i - 2) \leq b - 2$.

Proof. By induction on $|I|$. The basis with $|I| = 1$ is immediate. For $|I| > 1$, we

assume that the lemma is true for any region H' with $|I'| < |I|$. We show how to split H into H' and H'', so that the induction hypothesis can be applied to each, and the bound established. Index the boundary vertices in H in order around the exterior face from 1 to b. Consider any i' in I.

Let v_j and $v_{j'}$ be in $V_{i'}^b$ such that $j < j'$ and there is no $v_{j''}$ in $V_{i'}^b$ with $j < j'' < j'$. Then for any $V_{i''}^b$, either all vertices v_k in $V_{i''}^b$ have $j \le k \le j'$, or no vertices v_k in $V_{i''}^b$ have $j < k < j'$. Suppose $V_{i''}^b$ contained v_k and $v_{k'}$, where $j < k < j'$ and either $k' < j$ or $j' < k'$. There would be a path from v_j to $v_{j'}$ in $H_{i'}$, and a path from v_k to $v_{k'}$ in $H_{i''}$. Since all vertices in V^b are on the exterior face in H, the paths must cross, an impossibility.

Let $i', i'' \in I$, and without loss of generality choose j and j' as above so that all vertices v_k in $V_{i''}^b$ have $j \le k \le j'$. Now partition the H_i so that $i \in I'$ if and only if H_i contains no vertex v_k in $V_{i'}^b$ with $j < k < j'$, and $i \in I''$ otherwise. This induces a dividing of H into H' and H'', with $H_{i'}$ within H', and $H_{i''}$ within H''. Thus I is partitioned into I' and I'', with each of smaller cardinality than I. The only boundary vertices shared by H' and H'' are v_j and $v_{j'}$, so that $b' + b'' \le b + 2$. Applying the induction hypothesis, $\sum_{i \in I'} (b_i - 2) \le b' - 2$, and $\sum_{i \in I''} (b_i - 2) \le b'' - 2$. Thus $\sum_{i \in I} (b_i - 2) \le (b' - 2) + (b'' - 2) \le b - 2$. □

We summarize the construction of the decision tree for a region H which has an embedding with all boundary vertices on an exterior face. If $m = 0$ or $b = 1$, then the decision tree consists of a leaf labelled with the shortest path tree in H. If $b = 2$, then the decision tree is of height m, with a test at every level determining for each interior vertex

x in turn whether the shortest path to x goes through v_1 or v_2. If $m > 0$ and $b > 2$, then first build a decision tree of height $(b-1)s$, which determines for each vertex v_k in V^s through which vertex in V^b a shortest path from the source passes. Each leaf in this tree represents a certain division of H into t subregions based on the shortest paths from the source to vertices on these paths. Each subregion H_i has a corresponding prepared form, and the decision tree for H_i can be found recursively. Such a decision tree for H_1 will replace the leaf for that particular partition. The decision tree for H_2 will replace each of the leaves of H_1, and so on for the rest of the H_i's resulting from that partition. Each leaf will be labelled with the shortest path tree that is the result of the comparisons on the path down to that leaf. When all partitions of H have been handled, the decision tree for H is complete.

From the above discussion, the height of the resulting decision tree, and thus the time to search it, can be described by the recurrence

$$T(0, b) = T(m, 1) = 0$$

$$T(m, 2) = m$$

$$T(m, b) \leq \max_{s,\, t,\, m_i} \{s(b-1) + \textstyle\sum_{i=1}^{t} T(m_i, b_i)\}$$

where $m_i \leq 2/3$ (by the planar separator algorithm), and $\sum_{i=1}^{t} m_i \leq m$.

Lemma 6. For $b > 2$, the solution to the above recurrence satisfies

$$T(m, b) \leq c\sqrt{m}\ b + m$$

where $c = 6\sqrt{2} + 4\sqrt{3}$.

Proof. By induction on m. The basis is for $m = 0$. $T(0, b) = 0$ clearly satisfies the

above bound. The induction step is for $m > 0$. If $b = 1$ or $b = 2$, then the bound is clearly satisfied. If $b > 2$, then we have

$$\begin{aligned} T(m, b) &\leq \max_{s, t, m_i} \{s(b-1) + \textstyle\sum_{i=1}^{t} T(m_i, b_i)\} \\ &= \max_{s, t, m_i} \{s(b-1) + \textstyle\sum_{i \in I} T(m_i, b_i) + \sum_{i \notin I} T(m_i, b_i)\} \\ &\leq \max_{s, t, m_i} \{s(b-1) + \textstyle\sum_{i \in I} (c\sqrt{m_i}\, b_i + m_i) + \sum_{i \notin I} m_i\} \end{aligned}$$

Use of the planar separator ensures that $m_i \leq 2m/3$. From the manner in which the subregions are generated, $m \geq \sum_{i=1}^{t} m_i$, and $\sum_{i \in I} (b_i - 2) \leq b - 2$. Thus the sums are maximized if $|I| = 1$, with $m_i = 2/3$ and $b_i = b$ for the one i in I. The term involving s is maximized when s is as large as possible, which can be at most $2\sqrt{2}\sqrt{m}$ by the separator theorem [LT1]. Thus

$$\begin{aligned} T(m, b) &\leq 2\sqrt{2}\sqrt{m}\,(b-1) + c\sqrt{2m/3}\, b + m \\ &< (2\sqrt{2} + c\sqrt{2/3})\sqrt{m}\; b + m \end{aligned}$$

Taking c as above gives the claimed result. □

We now discuss the handling of a region for which there is no planar embedding with all boundary vertices on the exterior face. An example of such a region is region A in Figure 2. We show how to build a decision tree which finds a subset of boundary vertices with the desired property, while still allowing for the correct computation of shortest paths. We assume that a shortest path tree from the source to each of the boundary vertices in the graph will be known, and that the boundary vertices in this tree will have been given preorder and postorder numbers.

As input to the tree are shortest distances from the source vertex to each boundary vertex of the region, along with the preorder and postorder numbers of each boundary

vertex of the region with respect to the preorder and postorder numbering of all boundary vertices in a shortest path tree in the graph rooted at the source. It is well-known that a vertex v_j is an ancestor of v_k in a tree if and only if $preorder(v_j) \leq preorder(v_k)$ and $postorder(v_j) \geq postorder(v_k)$. Thus it is possible to determine the structure of the shortest path tree with respect to the b boundary vertices of H by comparing each preorder number with each other, and similarly with postorder numbers. This will take $b(b-1)$ comparisons. Precisely those boundary vertices in H with no ancestor that is a boundary vertex in H will be in the subset. All others will be viewed as interior.

To preserve shortest paths, certain pseudo-edges must be introduced. If v_k is a boundary vertex not included in the subset, with nearest boundary vertex ancestor v_j, then pseudo-edge $v_j, v_k)$ is introduced if and only if $d(source, v_k) - d(source, v_j)$ is less than the distance within H from v_j to v_k. Such an edge is of cost $d(source, v_k) - d(source, v_j)$, and represents the shortest path from v_j to v_k. At most $b-2$ comparisons are needed to determine which pseudo-edges must be introduced.

Thus we build a decision tree of height at most $b(b-1)+b-2$. On each pair of the first $b(b-1)$ levels a different pair v_j, v_k is tested to determine whether v_j is an ancestor of v_k, or vice versa. On the following at most $b-2$ levels it is determined which new edges are introduced. Associated with each leaf of the decision tree is an induced nice region, having as boundary vertices a subset of boundary vertices from the original region, and including the additional edges as described above.

Given this decision tree, we can extend it to a decision tree that finds the portion of shortest paths from a source going through any region. In place of each leaf in the

above tree, put a decision tree for the corresponding induced nice region. The resulting decision tree has the top at most $b(b-1)+b-2$ levels determining the induced region, and the remaining levels finding the shortest paths in the induced region. Leaves of the resulting decision tree will not contain pseudoedges.

Theorem 4. Let H be a region in a suitable r-division of a planar graph G. The portion in H of a shortest path tree for the source in G can be found using a decision tree of height $O(r)$, which takes as input the shortest distances from the source to each of the boundary vertices in H, plus the preceding boundary vertex in each such path. The decision tree can be built in $O(r^2 \log r\ 2^{cr})$ time, and will use $O(r2^{cr})$ space, for some constant c.

Proof. The region will have $O(\sqrt{r})$ boundary vertices, and at most r interior vertices. Thus the prepared form for region H will have b be $O(\sqrt{r})$ and $m \leq r$. By Lemma 6 and the preceding discussion, the height of the decision tree for region H will be $O(r)$. Determining shortest paths within the region, encoded as r shortest path trees, will use $O(r^2 \log r)$ time. The amount of work to determine the information generated by a set of prescribed tests will be $O(r^2)$. If $b = 2$, the result for each set of outcomes of the m comparisons can be translated into a shortest path tree in $O(r)$ time per vertex, or $O(r^2)$ time total. If $m > 0$ and $b > 2$, the result for each set of outcomes of the $(b-1)s$ comparisons can be translated into a dividing of a region into subregions in $O(b+r)$ time per separator plus $O(r)$ time per vertex, or $O(r^2)$ time in total. Since the decision tree is a binary tree, and of height at most cr for some constant c, the number of nodes in the tree is $O(2^{cr})$. Each of $O(2^{cr})$ leaves will use $O(r)$ space. □

7. A very fast search strategy

We use the decision tree technique from the previous section, plus an additional technique, to generate a search strategy that takes $O(n)$ time. In the section 5, by choosing $r_1 = (\log n)^2$, we could have realized $O(n \text{ loglog } n)$ search time, at the expense of $O(n \log n)$ preprocessing time. Furthermore, we could have reduced the search time to $O(n \text{ logloglog } n)$ if we had done the following. Instead of using Dijkstra's algorithm on level 1 regions for the mop-up, we could have run our main thrust on level 1 regions and then used Dijkstra's algorithm for the mop-up within level 2 regions. Assuming that $r_1 = (\log n)^2$ and $r_2 = (\text{loglog } n)^2$, this would have yielded a search time of $O(n \text{ logloglog } n)$. This idea can be carried to its limit by using $\log^* n$ levels. At $O(n)$ main thrust time per level, this would give $O(n \log^* n)$ time total.

We show how to do better using the decision trees. The preprocessing is the following. First find a suitable r_1-division of the graph, where $r_1 = (\log n)^2$. For each level 1 region, find a suitable r_2-division, where $r_2 = (\text{loglog } n)^2$. When finding this division, start with each level 1 boundary vertex automatically being a level 2 boundary vertex. For each level 2 region, find a suitable r_3-division, where $r_3 = (\text{logloglog } n)^2$. When finding these divisions, start with each level 2 boundary vertex automatically being a level 3 boundary vertex. For each level 3 region, build a decision tree. Then for each level 3 region, find the shortest paths between every pair of level 3 boundary vertices. This should be done by using the decision tree to solve each a single source problem for each boundary vertex. For each level 2 region, find the shortest paths between every pair of level 2 boundary vertices. The main thrust is used for this, using each level 2 boun-

dary vertex in turn as the source. Finally for each level 1 region, find the shortest paths between every pair of level 1 boundary vertices, using the main thrust from each level 1 boundary vertex.

The search of the graph is then the following. To perform a single source computation, first find the shortest paths from the source to the boundary vertices of the level 1 region containing the source. This can be done using Dijkstra's algorithm. Given the shortest distances to boundary vertices in the level 1 region containing the source, perform the main thrust on level 1 boundary vertices. Given these results, perform the main thrust within each level 1 region on level 2 boundary vertices. Then perform the main thrust in each level 2 region on level 3 boundary vertices. Finally, complete the problem in each level 3 region by using the decision tree.

Theorem 5. An n-vertex planar graph can be preprocessed in $O(n \log n)$ time so that each subsequent shortest path computation can be performed in $O(n)$ time.

Proof. The r_1-division can be found in $O(n \log n)$ time. The r_2- and r_3-divisions can be seen to require no more time than this. Building decision trees for all of the $\Theta(n/r_3)$ level 3 regions, will take time $O(nr_3 \log r_3\, 2^{cr_3})$. With $r_3 = (\log\log\log n)^2$, the time will be $O(n \log n)$. Finding the shortest paths between boundary vertices at levels 3, 2 and 1 can be seen to take $O(n \log n)$ time by arguments similar to previous ones. Thus the preprocessing time is $O(n \log n)$. For the time to solve the next single source problem, we have the following. The initial shortest path computation within the level 1 region containing the source will use $O(r_1 \log r_1)$ time, which is $O((\log n)^2 \log\log n)$. The searches on boundary vertices at levels 1, 2 and 3 can be seen to take $O(n)$ time. The

search within a level 3 region, using the decision tree for the region, will be $O(r_3)$, which yields $O(n)$ total for all level 3 regions. □

The best previous algorithm for all pairs shortest paths in a planar graph is presented in Johnson [J]: For the case of nonnegative edge costs, Dijkstra's algorithm with a heap implementation may be run from each vertex, yielding $O(n^2 \log n)$ time. If there are edges of negative cost, but no negative cycles, and the edges are directed, then the best single source algorithm for planar graphs [LRT] uses $O(n^{3/2})$ time. However, such an algorithm need be used only once in an all pairs computation. Dijkstra's algorithm will perform correctly if vertices are closed in order of $d(v)+h(v)$, where $h(v)$ is the shortest distance from v to a specific vertex u [EK, J, Ne, Ni]. Using our algorithm in place of Dijkstra's, we achieve the following.

Theorem 6. The all pairs shortest paths problem on an n-vertex planar graph with either undirected edges and no negative costs or directed edges and negative costs but no negative cycles can be solved in $O(n^2)$ time. □

8. Minimum cut and maximum flow

Our shortest path procedure may be used to speed up Reif's algorithm for finding a minimum $s-t$ cut in an undirected planar network, which runs in $O(n(\log n)^2)$ time [R]. The idea behind the algorithm is that a minimum cut corresponds in the planar dual to a shortest cycle that separates s and t, and that this cycle can be found efficiently by a divide-and-conquer strategy. Reif's algorithm proceeds in the following manner. The

graph G is embedded in the plane, and the dual graph $D(G)$ is determined. Each edge in $D(G)$ has cost equal to the capacity of its corresponding edge in G. Let P_{st} be a minimum cost path in $D(G)$ from a vertex representing a face in G that borders s to a vertex representing a face in G that borders t.

If P_{st} contains one vertex, v, then s and t are on the same face, f_v, in G. The minimum $s-t$ cut in G can be found by using a shortest path algorithm in an augmented dual $D'(G)$ generated as follows. Vertex v is replaced by new vertices v' and v''. Each edge (v,w) incident on v is replaced by by (v',w) if the corresponding edge in G is in that portion of the boundary of face f_v that starts at s and proceeds clockwise to t, and by (v'',w) otherwise. The shortest path P_v between v' and v'' corresponds to the minimum cost cycle in $D(G)$ containing v that separates face s from face t. (This portion of Reif's algorithm corresponds to the $O(n \log n)$ time algorithms of Itai and Shiloach [IS] and Hassin [Hs1] for finding a maximum $s-t$ flow in a planar graph when both s and t are on the same face.)

If P_{st} contains more than one vertex, Reif's algorithm does the following. The augmented dual $D'(G)$ is generated by splitting apart the path P_{st} into two paths P_{st}' and P_{st}'', duplicating the vertices and edges of P_{st}, and replacing the edges incident on P_{st} as follows. Any edge (v,w) that is not in P_{st} but is incident on vertex v in P_{st} is replaced by (v',w) if (v,w) is to the right of P_{st} in $D(G)$, and by (v'',w) if (v,w) is to the left of P_{st} in $D(G)$. Once $D'(G)$ is generated, it is searched as follows. Let v be a midpoint of the original path P_{st}. Let v' and v'' be the corresponding vertices in $D'(G)$. A shortest path computation in $D'(G)$ is performed using v' as the source and v'' as the destination.

The shortest path P_v between v' and v'' corresponds to the minimum cost cycle in $D(G)$ containing v that separates face s from face t. The network $D'(G)$ is split into two subnetworks along P_v, including vertices and edges from P_v in both subnetworks. Any maximal path of at least two bridges is replaced by a new edge, of cost the sum of costs in the maximal path. Then Reif's algorithm recurses on each half. Among the cycles so identified, the cycle of minimum cost will correspond to a minimum $s-t$ cut. The work performed across each level of recursion is dominated by the shortest path computations, which are $O(n \log n)$ time if Dijkstra's algorithm is used. The time to handle all levels of recursion is $O(n(\log n)^2)$, since there are $O(\log n)$ levels of recursion.

In place of Dijkstra's shortest path algorithm, we will use our own algorithm. We perform preprocessing similar to that discussed in section 7 on the augmented dual $D'(G)$. First find a suitable r_1-division of the graph, where $r_1 = (\log n)^2$, and then a suitable r_2-division of each level 1 region, where $r_2 = (\log\log n)^2$. When finding this division, start with each level 1 boundary vertex automatically being a level 2 boundary vertex. For each level 2 region, find the shortest paths between every pair of level 2 boundary vertices, using Dijkstra's algorithm. Finally for each level 1 region, find the shortest paths between every pair of level 1 boundary vertices, using the main thrust from each level 1 boundary vertex.

Now perform Reif's minimum cut algorithm, but whenever a source-destination shortest path computation is required, use the following. Find the shortest paths from the source to the boundary vertices of the level 2 region containing the source. Dijkstra's algorithm can be used for this purpose. Then find the shortest paths to the boundary ver-

tices of the level 1 region containing the source. Our main thrust can be used within this level 1 region. Now perform our main thrust on level 1 boundary vertices. This will in particular give the shortest distances to the boundary vertices of the level 1 region containing the destination. Perform the main thrust on level 2 boundary vertices within this region. This will give the shortest distances to the boundary vertices of the level 2 region containing the destination. Finally perform a shortest path computation within this region, using Dijkstra's algorithm. This will then give the shortest path from the source to the destination.

Whenever a shortest path corresponding to a cycle is identified, a number of regions in general will need to be split. However it is not hard to do this if the boundaries are maintained in a convenient form. The regions can be split so that an efficient main thrust can still be carried out. For each region, maintain a list of its boundary vertices, ordered by boundary set, and within boundary set ordered by the order along the boundary. For each boundary vertex in the region, shortest distance information to all other boundary vertices is kept in this same order. When a region is split, the vertices are partitioned into two sets, each of which can be described by a sequence of index pointers into the original list. If a subnetwork contains a sequence of split regions with just two boundary vertices, then this sequence should be replaced by a pseudo split region with just two boundary vertices. This operation is analogous to replacing a sequence of bridges in Reif's algorithm.

During a search, a split region will be handled in the following fashion. When a boundary vertex in one set of the partition is closed, only vertices in the same set of the partition are updated. This means that for all searches at any level of the recursion in the

modification of Reif's algorithm, no more work is done in updating the heap than at the top level of the recursion. This follows since the same number of boundary vertices are closed as in the top level of recursion, and for each boundary vertex that is closed, the updating is at worst no more expensive than at the top level.

Theorem 7. A minimum $s-t$ cut in an undirected planar network can be identified in $O(n \log n)$ time.

Proof. The time required is the following. The preprocessing time is $O(n \log n)$. The time to search from a source to the boundary vertices of the level 2 region containing it is $O(r_2 \log r_2)$, which is $O((\text{loglog } n)^2 \text{logloglog } n)$. The time to search from the boundary vertices of the level 2 region containing the destination to the destination itself will be the same. Since there are $O(n)$ source-destination computations, these activities will use in total $O(n(\text{loglog } n)^2 \text{logloglog } n)$ time. The remaining activities in the source-destination computations are accounted for by considering all such computations on one level of recursion. The main thrusts on any one such level of recursion will take $O(n)$ time. Note that the time to split regions corresponding to the source-destination paths found will total no more than proportional to the number of edges in the graph, which is $O(n)$. Since there are $O(\log n)$ levels, the total time for all source-destination computations is $O(n \log n)$. □

Hassin and Johnson [HJ] have shown how to use Reif's construction to find a maximum flow in a planar undirected network in $O(n(\log n)^2)$ time. The output that they need from Reif's algorithm is the minimum cost $s-t$ cut cycle in $D(G)$ that con-

tains v, for each vertex v in P_{st}. Aside from the time to find a minimum $s-t$ cut, the Hassin and Johnson algorithm uses $O(n \log n)$ time. Since our minimum cut algorithm produces the same information about cut cycles as Reif's, ours may be used in its place.

Corollary 2. A maximum $s-t$ flow in an undirected planar network can be found in $O(n \log n)$ time. □

9. Multicommodity flows

In this section we discuss the multicommodity flow problem in a planar graph in which all sources and sinks are on the same face. Algorithms for determining feasibility and finding feasible flows in such graphs have been presented in [Hs2, MNS]. The feasibility test for multicommodity flows given in [MNS] uses $O(\min\{n^2 \log^* n, kn\sqrt{\log n}\})$ time, where k is the number of source-sink pairs. A flow construction algorithm is also presented in [MNS], which runs in $O(kn + n^2\sqrt{\log n})$ time. We show how to adapt our techniques to yield a faster feasibility test.

Let B be the set of edges bounding the face f that contains all the sources and sinks as vertices. Let b be the number of vertices of this face that serve as sinks and/or sources. We assume that $b \leq 2k$, since if $b > 2k$, then additional edges of capacity 0 can be added to the graph to yield $b \leq 2k$. As discussed in [Hs2, MNS], these edges will connect consecutive source and/or sink vertices on the boundary of face f. Thus the only vertices on the boundary of face f in the modified graph will be sources and/or sinks, implying that the number of edges on the boundary of this face will be at most $2k$. We assume that the input graph has already been so modified, and quote our results in

these terms. Also, $k \leq b(b-1)/2$, since there are at most this many distinct pairs from among the vertices bounding f.

The feasibility can be tested as follows. Let e and e' be two edges in B. Let $C(e, e')$ be the set of cuts that contain as the only edges from B the edges e and e'. For a cut X, let $c(X)$ be the total capacity of X, and $d(X)$ the total demand of X. Let $m(X) = c(X)-d(X)$ be the *margin* of cut X. Define $m(e, e')$ as the minimum $m(X)$ such that X is in $C(e, e')$. From results in [OS], it is shown in [Hs2, MNS] that there is a feasible k-commodity flow if and only if $m(e, e') > 0$ for every pair e, e' in B.

In [Hs2, MNS], the values $m(e, e')$ are determined in the following way. For a fixed e in B and all e' in B, $d(e, e')$ can be determined in $O(k+b)$ time. By traversing the edges of B in order starting at e, the values of $d(e, e')$ for all pairs (e, e') can be found in $O(k+b^2)$ time. To compute the values of $c(e, e')$, a dual graph $D(G)$ is constructed, with costs on the dual edges equal to capacities of the corresponding original edges. Let $c(e, e')$ be the minimum value of $c(X)$ for X in $C(e, e')$. Then $c(e, e')$ is equal to the sum of the capacities of e and e' plus the length of a shortest nontrivial path joining e and e' in $D(G)$. Using Dijkstra's algorithm, all values $c(e, e')$ can be found in $O(k + bn \log n)$ time. In [MNS], it is noted that our single source and all pairs algorithms (presented in [F2]) can be used to reduce the time to $O(\min\{n^2 \log^* n, bn\sqrt{\log n}\})$.

However we can speed up the computation of the $c(e, e')$ values more by using the shortest path methods described in sections 6 and 7. We use our approach of one-time preprocessing and many-time search, but tailor it to the value of b. If $b \leq \log n$,

choose $r_1 = b \log n$. If $b > \log n$, choose $r_1 = (\log n)^2$, as before.

Theorem 8. The testing for feasibility of a k-commodity flow in an n-vertex planar graph in which all sources and sinks are on one face, bounded by b edges, $b \leq 2k$, can be performed in $O(bn + n\sqrt{b \log n})$ time.

Proof. First consider the case when $b \leq \log n$, and $r_1 = b \log n$. The preprocessing time will be dominated by the time to find shortest path trees for each level 1 boundary vertex, which in total will be $O(n\sqrt{r_1})$, or $O(n\sqrt{b \log n})$. The b iterations of search will each be dominated by the main thrust at level 1, which is $O(b(n + (n/\sqrt{r_1})\log n))$ by Theorem 2, or $O(n\sqrt{b \log n})$.

Next consider the case when $b > \log n$, and $r_1 = (\log n)^2$. By Theorem 5, the preprocessing will be $O(n \log n)$, which is $O(bn)$. The b iterations of search and post-processing will use $O(bn)$ time.

Finally, the $O(k + b^2)$ time to determine all $d(e, e')$ values is $O(b^2)$ by previous discussion, and the latter is $O(bn)$. □

Acknowledgements. The author would like to thank Takao Nishizeki, Knut Moeller, Carsten Vogt, Peter Klosterberg, and an anonymous referee for their careful reading of the manuscript and many helpful suggestions.

References

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[DP] N. Deo and C. Pang, Shortest-path algorithms: taxonomy and annotation, *Networks 14* (1984) 275-323.

[D] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik 1* (1959) 269-271.

[EK] J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *J.ACM 19*, 2 (April 1972) 248-264.

[F1] G. N. Frederickson, Data structures for on-line updating of minimum spanning trees, with applications, *SIAM J. on Computing 14*, 4 (November 1985) 781-798.

[F2] G. N. Frederickson, Shortest path problems in planar graphs, *Proc. 24th IEEE Symp. on Foundations of Computer Science*, Tucson (November 1983) 242-247.

[Hr] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass. (1969).

[Hs1] R. Hassin, Maximum flow in (s,t) planar networks, *Inf. Proc. Lett. 13*, 3 (Dec. 1981) 107.

[Hs2] R. Hassin, On multicommodity flows in planar graphs, *Networks 14* (1984) 225-235.

[HJ] R. Hassin and D. B. Johnson, An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks, to appear in *SIAM J. on Computing 24*, 3 (Aug. 1985).

[IS] A. Itai and Y. Shiloach, Maximum flow in planar networks, *SIAM J. Comput. 8* (1979) 135-150.

[J] D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM 24*, 1 (January 1977) 1-13.

[LRT] R. J. Lipton, D. J. Rose and R. E. Tarjan, Generalized nested dissection, *SIAM J. Numer. Anal. 16* (1979) 177-189.

[LT1] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math. 36*, 2 (April 1979) 177-189.

[LT2] R. J. Lipton and R. E. Tarjan, Applications of a planar separator theorem, *SIAM J. Comput. 9*, 3 (August 1980) 615-627.

[MNS] K. Matsumoto, T. Nishizeki, and N. Saito, An efficient algorithm for finding multicommodity flows in planar networks, *SIAM J. Computing 14*, 2 (May 1985) 289-302.

[Ne] G. L. Nemhauser, A generalized permanent label setting algorithm for the shortest path between specified nodes, *J. Math. Anal. Appl. 38*, 2 (May 1972) 328-334.

[Ni] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York (1971).

[OS] H. Okamura and P. D. Seymour, Multicommodity flows in planar graphs, *J. Combinatorial Theory, Series B 31* (1981) 75-81.

[R] J. H. Reif, Minimum $s-t$ cut of a planar undirected network in $O(n \log^2(n))$ time, *SIAM J. Comput. 12*, 1 (Feb. 1983) 71-81.

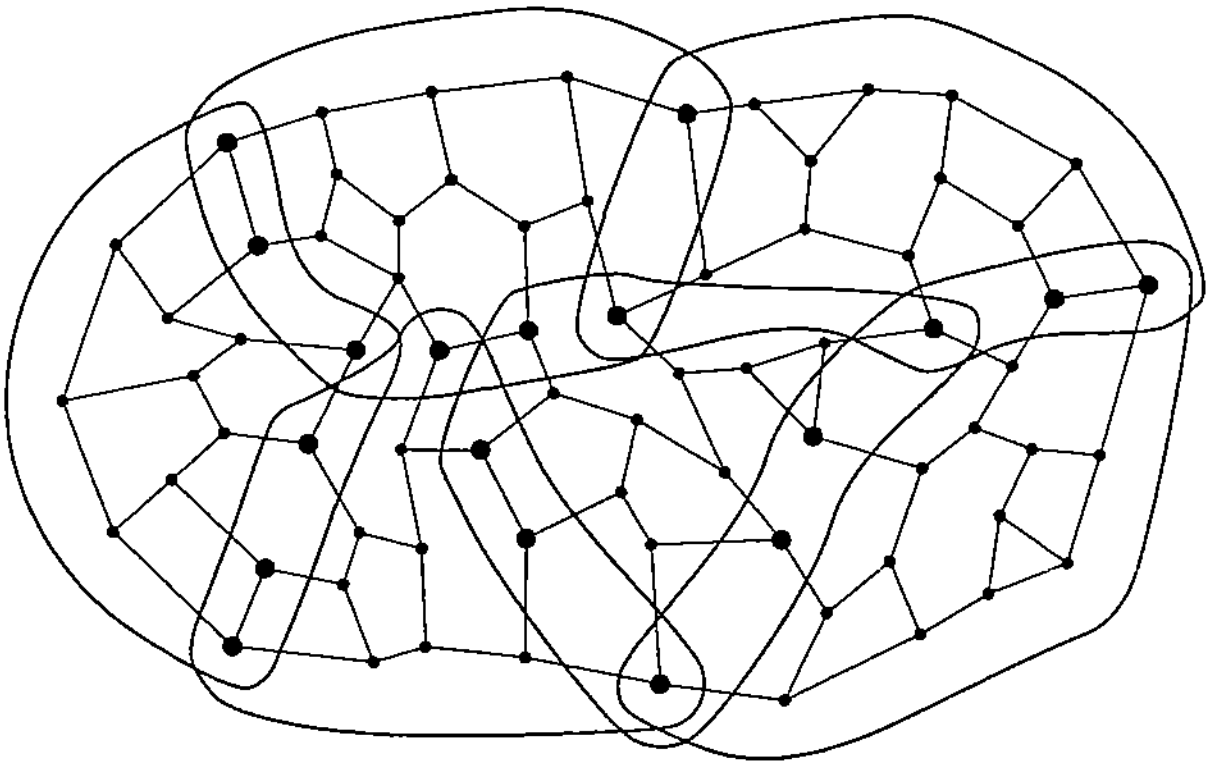


Figure 1. Division of a planar graph into regions.

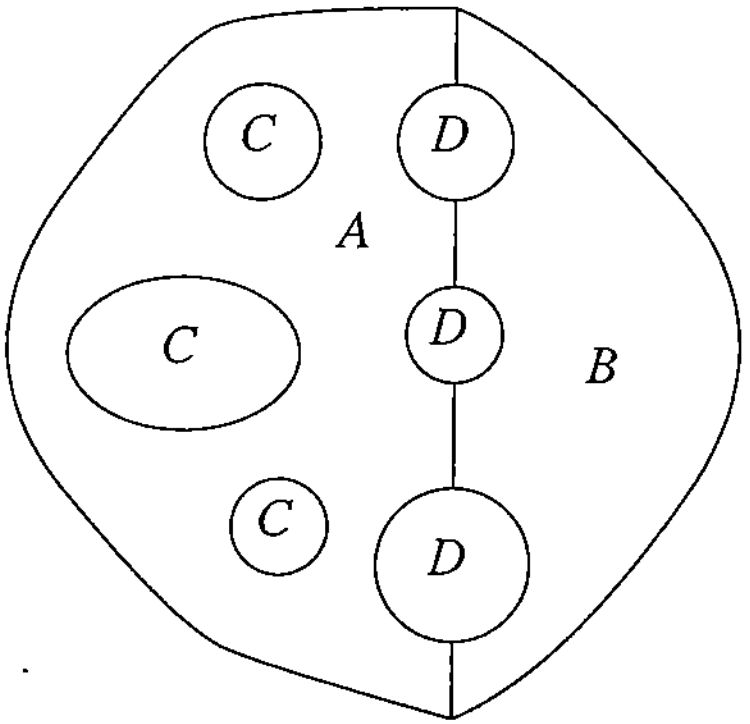
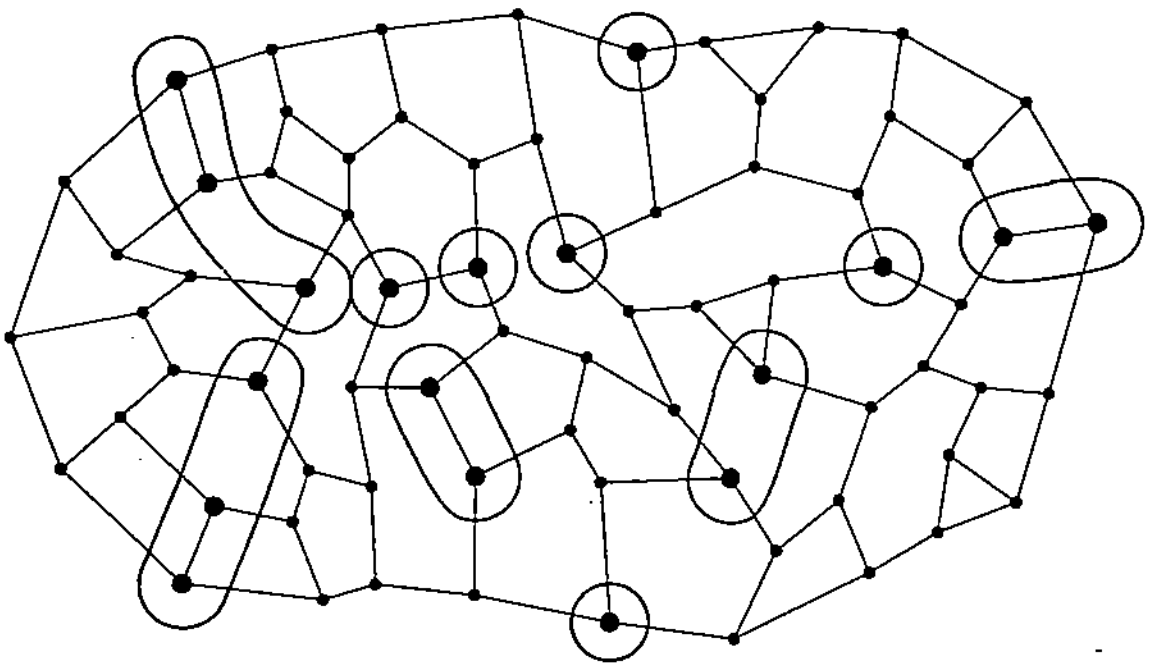
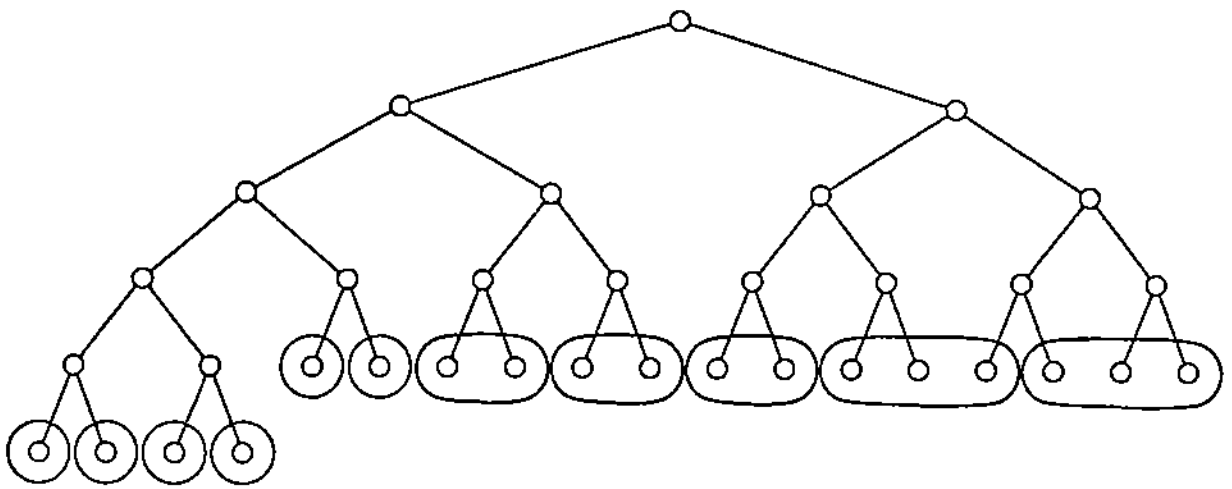


Figure 2. Examples of regions that are unions of connected components.



(a)



(b)

Figure 3. a. Boundary sets for the regions of Figure 1, and the
b. Corresponding topology-based heap.

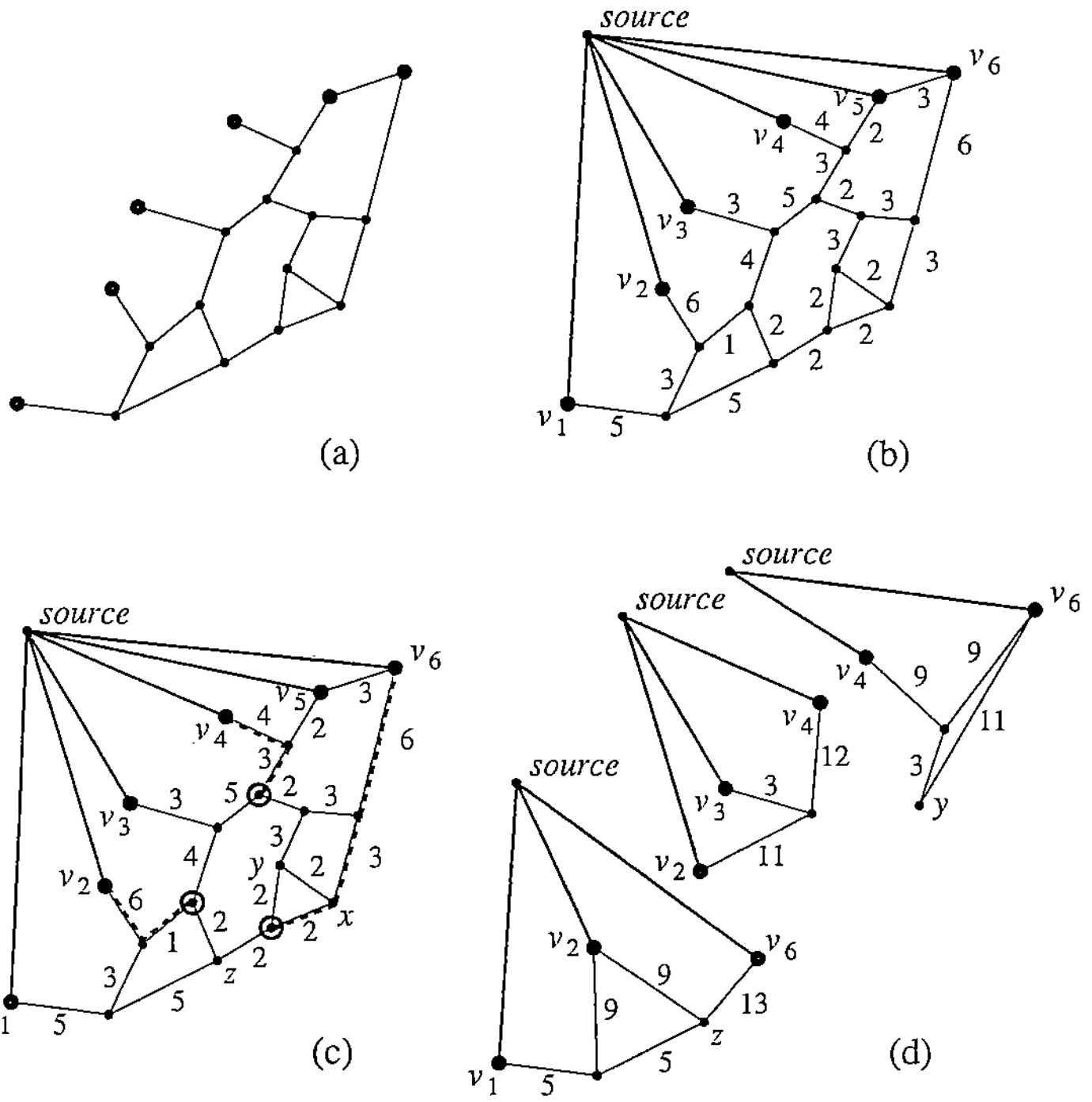


Figure 4. a. A region from the division in Figure 1,
b. Its corresponding prepared form, with edge costs shown,
c. Separator vertices circled, and shortest paths to them shown,
d. Prepared subregions induced by shortest paths to separator vertices.