First publ. in: Algorithmica 28 (2000), 1, pp. 16-36

A Linear Time Algorithm for the Arc Disjoint Menger Problem in Planar Directed Graphs[1]

U. Brandes[2] and D. Wagner[2]

Abstract. Given a graph $G = (V, E)$ and two vertices $s, t \in V$, $s \neq t$, the Menger problem is to find a maximum number of disjoint paths connecting s and t. Depending on whether the input graph is directed or not, and what kind of disjointness criterion is demanded, this general formulation is specialized to the directed or undirected vertex, and the edge or arc disjoint Menger problem, respectively.

For planar graphs the edge disjoint Menger problem has been solved to optimality [W2], while the fastest algorithm for the arc disjoint version is Weihe's general maximum flow algorithm for planar networks [W1], which has running time $\mathcal{O}(|V| \log |V|)$. Here we present a linear time, i.e., asymptotically optimal, algorithm for the arc disjoint version in planar directed graphs.

Key Words. Graph algorithms, Disjoint paths, Planar graphs.

1. Introduction. Due to their importance, in their own right as well as in bottleneck routines of other algorithms, disjoint path problems have been studied extensively. The famous Menger theorems [M] are structural in nature. However, they have not only been generalized to capacitated versions like the max-flow/min-cut theorem, but also extended to algorithms actually constructing disjoint paths, separators, or cuts.

A generic formulation of Menger's problem is the following: Given a graph $G = (V, E)$ and two distinct vertices $s, t \in V$, find a maximum cardinality set of disjoint paths connecting s and t. This leads to four concrete versions of the problem. The instances are either directed or undirected, and the (s, t)-paths have to be vertex or edge (arc) disjoint.

For planar undirected graphs, linear time algorithms exist for both the vertex [RWW2] and edge disjoint case [W2]. However, there is no such algorithm for either case when the planar input graphs are directed. In any graph the arc disjoint Menger problem obviously corresponds to a maximum flow problem with unit capacities [AMO]. The first algorithm tailored to solve the maximum flow problem with arbitrary capacities especially in planar graphs was presented in [IS]. Faster algorithms have subsequently been developed, e.g., [JV] and [KRR$^+$]. By now, the fastest algorithm is that of [W1] yielding a running time of $\mathcal{O}(n \log n)$, where $n = |V|$. Here we concentrate on the more special Menger problem

[1] The authors acknowledge the Deutsche Forschungsgemeinschaft for partially supporting this research under Grant Wa 654/10-1.

[2] Department of Computer and Information Science, University of Konstanz, Box D 188, 78457 Konstanz, Germany. {Ulrik.Brandes, Dorothea.Wagner}@uni-konstanz.de. http://www.informatik.uni-konstanz.de/∼{brandes,wagner}.

and present a linear time solution. Our algorithm is not only faster than the max-flow algorithm, but also considerably simpler.

Our approach is based on right-first-search, which appears to be extremely suitable for path problems in planar graphs (see [RWW1]). In particular, the optimal algorithms for the Menger problem in undirected graphs [RWW2], [W] are based on this variant of depth-first-search. Given a fixed planar embedding of the input graph, a right-first-search chooses arcs according to a *right-hand rule*, i.e., the continuation arc is the counterclockwise next arc leaving the vertex that is entered by the current arc. One of the main difficulties encountered by this strategy is the treatment of right cycles. Similar to [W2], we therefore use an observation of [KNK] to restrict the set of input instances to graphs without right cycles.

Roughly speaking, the algorithm successively occupies arcs in order to build a set of (s, t)-paths. The paths in this set are frequently reorganized, such that the determination of consecutive arcs on the same path becomes intricate. Another problem is the efficient choice of an arc to backtrack with when the path that is currently built can no longer be extended. Together, these problems make a linear time implementation rather difficult. The obstacles are overcome by a careful analysis of partial solutions which leads to local characterizations resolving both problems.

In Section 2 we introduce our basic terminology and show how to restrict the problem to certain input instances. Section 3 gives a description of the algorithm on an abstract level, providing a better understanding of the underlying ideas. Its correctness is proved in Section 4. In Section 5 properties of partial solutions are examined. Based on these properties, a linear time implementation of the algorithm is described in Section 6. We conclude in Section 7 with a short discussion on the vertex disjoint Menger problem in planar directed graphs.

2. Preliminaries. We first introduce our basic assumptions and terminology. We are given an embedded planar graph $G = (V, A)$ with distinct vertices $s \neq t$. The *adjacency list* of a vertex $v \in V$ is a cyclic list of all arcs incident to v, arranged in the order in which they appear around v in the embedding. We often make use of this ordering, and say that an arc a is the *first arc after* b in (counter)clockwise order around v, if b is an immediate successor of a when the adjacency list of v is traversed in a (counter)clockwise fashion.

With the assumption of a fixed embedding we can make heavy use of spatially descriptive terms, like left and right, inside and outside, etc. For example, the right side of a directed path is its right-hand side when following its arcs directions. A directed cycle divides the plane into two disjoint regions, its left-hand region and its right-hand region. The region containing the outer face is called its *exterior*, the other is called its *interior*. A cycle is called a *left* (*right*) *cycle* if its interior equals its left-hand region (right-hand region). Cycles with s in their interior are called *orbits*.

Note that a maximum collection of arc disjoint directed (s, t)-paths in G corresponds to a maximum flow from s to t, if all arcs have unit capacity. Conversely, it is easy to construct a maximum collection of (s, t)-paths from an integral maximum flow. Also, given a maximum integral flow, a partition of the vertices inducing a minimum cut can always be found in linear time by a simple labeling algorithm.

Moreover, the maximum flow value does not change when a set of right cycles is replaced by left cycles which are obtained by simply altering arc orientations. Therefore, let $\mathcal{C}$ be a set of right cycles. Then $G_{\mathcal{C}} = (V, A_{\mathcal{C}})$ is called the *residual graph*, where $A_{\mathcal{C}}$ is the set of all arcs (v, w), with $(v, w) \in A$ and (v, w) does not belong to a cycle in $\mathcal{C}$, or $(w, v) \in A$ and (w, v) does belong to a cycle in $\mathcal{C}$. Note that reversion of arcs may introduce multiple arcs, which makes $A_{\mathcal{C}}$ a multiset. If $f_{\mathcal{C}}\colon A_{\mathcal{C}} \to \{0, 1\}$ is a maximum integral flow in the residual graph $G_{\mathcal{C}}$, then a maximum integral flow $f\colon A \to \{0, 1\}$ in G is obtained by setting $f(v, w) = f_{\mathcal{C}}(v, w)$, if $(v, w) \in A$ and $(v, w) \in A_{\mathcal{C}}$, and $f(v, w) = 1 - f_{\mathcal{C}}(w, v)$, if (w, v) is the replacement of (v, w) in $A_{\mathcal{C}}$. From [KNK] it can be seen that there exists a set $\mathcal{C}$ of right cycles such that the residual graph $G_{\mathcal{C}_r}$ contains left cycles only.[3] Moreover, $\mathcal{C}_r$ can be found in linear time using a breadth-first-search in the planar dual of G, which can be obtained in linear time as well. This technique was first applied in [W2]. Figure 1 gives an example. A linear time algorithm solving the arc disjoint Menger problem in $G_{\mathcal{C}_r}$ is thus sufficient to provide a linear time solution for the problem in G.

In the remainder we assume that we are given a planar directed graph $G = (V, A)$ that is embedded in the plane such that t is on the boundary of the outer (i.e., the infinite)

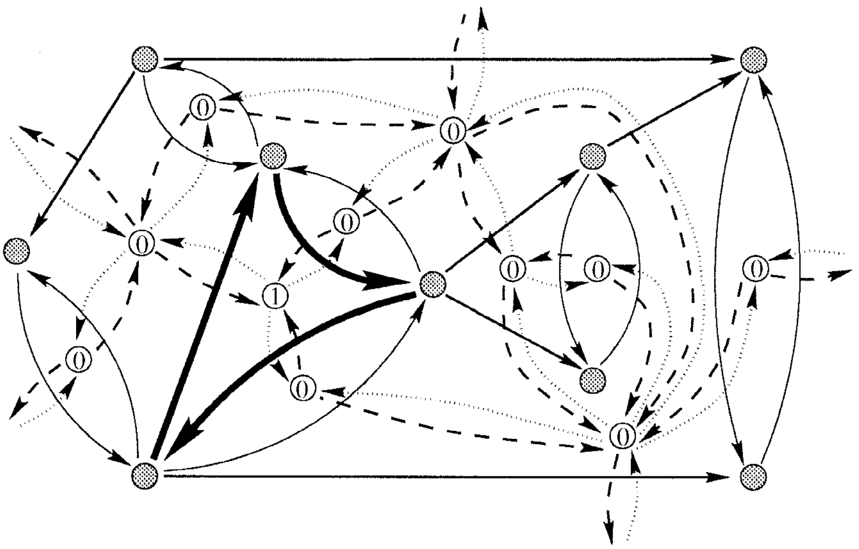

Fig. 1. Elimination of right cycles in a directed planar graph: Arcs between the same vertices are embedded such that they form a counterclockwise cycle. For each primal arc (drawn solid), there is a dual arc of length zero crossing from right to left (dotted), and an arc of length one crossing from left to right (dashed). Each face is assigned a potential equal to its distance from the outer face (shown inside the dual vertices). A primal arc belongs to the set of right cycles that are reversed, if its right face has higher potential than its left face (thick arcs).

[3] $\mathcal{C}_r$ corresponds to the *bottom element* of the distributive lattice formed by all circulations of G with appropriately defined operations for *meet* and *join* [KNK].

face[4] and contains no right cycle. We may further assume that there are no loops, no arcs entering s, and no arcs leaving t, since these obviously do not affect the maximum number of arc disjoint directed (s, t)-paths.

3. The Algorithm.

In this section we present an algorithm that determines a maximum set of (possibly nonsimple) arc disjoint (s, t)-paths in a planar directed graph that contains no right cycle, and is embedded (at least combinatorially) such that t is on the boundary of the outer face. We have outlined in the previous section that the arc disjoint Menger problem can be solved for any planar directed graph with linear overhead, if it can be solved for instances of this particular class of graphs.

For convenience, we here describe the algorithm on an abstract level, which both facilitates understanding and displays the basic simplicity of our approach. Nonetheless, it is not at all obvious how a linear worst case complexity can be obtained.

The algorithm applies a special variant of depth-first-search, right-first-search, which is suitable for many problems involving paths in planar graphs [RWW1]. As a by-product, the resulting solution is rightmost in the sense that no path can be routed further to the right without changing others.

All paths and cycles in this section are directed. After each step, the partial solution consists of a *search path*, which starts at s and ends at some vertex $v \neq t$, and a set of (s, t)-paths and left cycles, such that every arc belongs to at most one path or cycle. Given such a set of arc disjoint directed paths and cycles, each path (cycle) induces a straightforward (cyclic) *traversal order* on its arcs. For a directed (sub)path, its *first* and *last* arc are well defined, then. We say that two arcs are *consecutive* if they are immediate successors in the traversal order of a path or cycle, respectively. The last arc of the search path is called the *leading arc*, and its head is called the *leading vertex*. We say that two pairs of consecutive arcs form a *crossing* if they share their middle vertex v, and their arcs are encountered alternately when traversing the cyclic order of arcs incident to v. See Figure 2(a). A set of arc disjoint paths and cycles is said to be *noncrossing* if no two pairs of consecutive arcs form a crossing.

The algorithm uses only three basic operations: *search steps*, *backtracking steps*, and *realignments*.

Search Step. An unsearched arc leaving the leading vertex is added to the search path. Among all unsearched arcs, the counterclockwise first after the current leading arc in the adjacency list of the leading vertex is chosen (right-hand rule).

Backtracking Step. Some arc of the search path entering the leading vertex is removed from the graph. In this general version of the algorithm, any such arc is suitable. In the implementation presented in Section 6, our specific choice is subject to certain local configurations and the stage of the algorithm. If a nonsimple search path has more than one arc entering the leading vertex, the removal may split the search path into the new search path starting at s and ending at the removed arc's tail, and a remaining

[4] Such an embedding can always be obtained in linear time [HT].

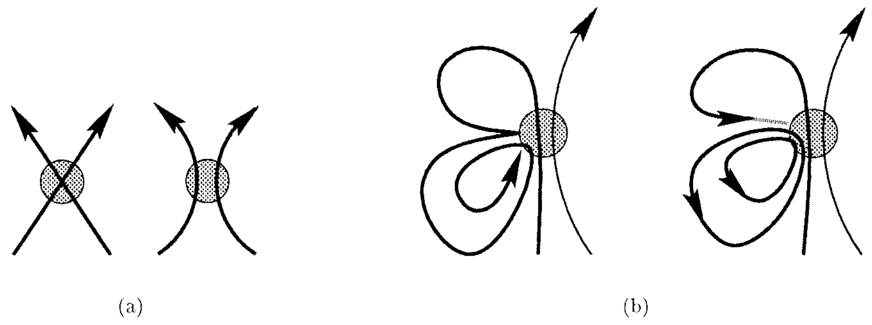

Fig. 2. (a) Crossing (left) and noncrossing (right) pairs of consecutive arcs. (b) After backtracking with the second arc of the search path entering the leading vertex, there are two resulting left cycles (containing four irrelevant arcs incident to the formerly leading vertex). The removed arc is indicated by the hashed line.

subpath starting and ending at the leading vertex, say v. We then modify the traversal order with respect to the arcs of the cut-off end of the search path that are incident to v, such that the subpath is transformed into a set of left cycles that do not cross at v. Each of these cycles is constrained to have exactly two arcs incident to v. See Figure 2(b) and note that there is a unique reassignment of consecutive arcs satisfying these conditions.

We refer to the reassignment of consecutive arcs during a backtracking step as *closing left cycles*, and to the arcs that are reassigned as *irrelevant*. Every arc that belongs to a path or cycle and is not irrelevant is called *relevant*.[5]

In order to introduce the third operation, some more terminology is needed. For a vertex $v \in V \setminus \{s, t\}$, we define a *passage through* v, or v*-passage* for short, of a path or cycle to be an (inclusion-)maximal subpath with the following properties: Its first arc is a relevant arc entering v and its last arc is a relevant arc leaving v. Moreover, if it is nonsimple, then s is in the exterior of every cycle formed by the subpath. Figure 3(a) gives an example. If existent, the last v-passage of the search path is called the *leading* v*-passage*. An arc (u, v) is said to *hit* some v-passage p from the right (left), if it is on the v-passage's right (left) side, and p and (u, v) are on the same side of every other v-passage. An arc (v, w) is said to *leave* a v-passage to the right (left) in the analogous situation. Two v-passages *touch* at v, if they do not cross at v, and each of them contains an arc hitting or leaving the other. Paths and cycles are hit, left, or touched at a vertex v if one of their v-passages is. Figure 3(b) summarizes these definitions.

Realignment. Let v be the leading vertex, and let some v-passage be hit from the right by the leading arc. Since there are no right cycles, there must be consecutive arcs (u, v) and (v, w) of the v-passage hit, such that the leading arc appears between (u, v) and (v, w) in the counterclockwise cyclic order of arcs incident to v. The search path is said to

[5] Actually, arcs ought to be called relevant or irrelevant *with respect to* one of their incident vertices. We omit this distinction, because from the context it should always be clear with respect to which vertex an arc is relevant or irrelevant, respectively.

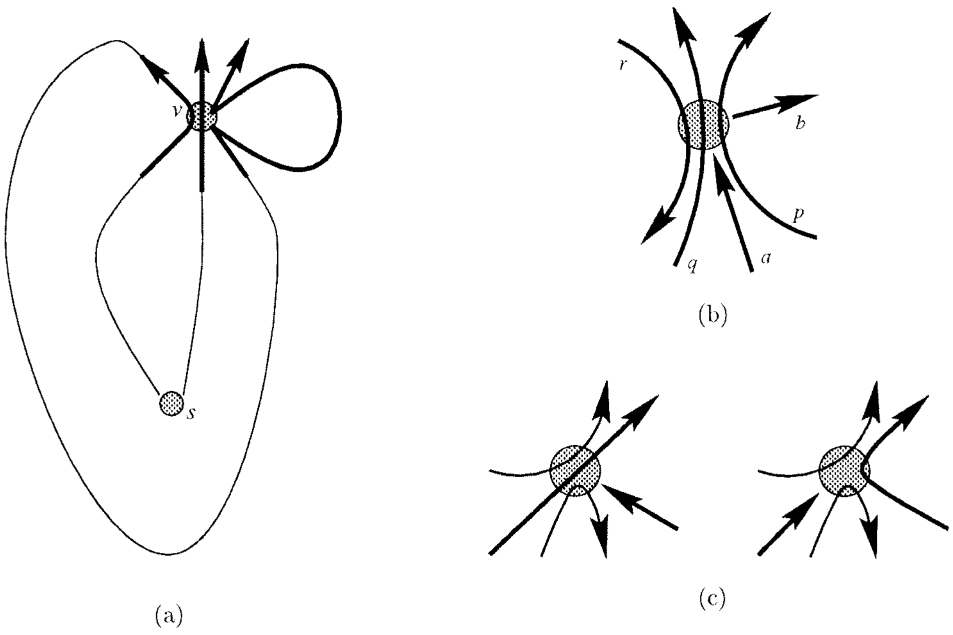

Fig. 3. (a) Two (s, t)-paths forming three v-passages. Note that one path has two v-passages, because it surrounds s. (b) Three passages p, q, r and two arcs a, b incident to the same vertex. Passage p touches passage q on the right, while r touches q on the left side. On the other hand, p and r do not touch at all. Arc a hits q from the right, and p from the left. However, it does not hit r. Arc b leaves p to the right, but neither q nor r. Since they do not lie on a common path or cycle, a and b are not consecutive. (c) Realignment of a passage that is hit from the right.

be *realigned* with the corresponding path or cycle if the leading arc is made consecutive with (v, w), so that (u, v) becomes the new leading arc. See Figure 3(c).

We are now ready to state our algorithm in simple terms. The bottom line is that we always try to go as far to the right as possible. The contribution of realignments is twofold: on one hand, they prevent crossings, and on the other hand, they ensure that the search path is in some sense leftmost at the leading vertex when backtracking has to be performed.

4. Correctness.

In this section we prove that after termination of Algorithm 1 the set of (s, t)-paths generated is maximum. A set of arcs whose removal disconnects s and t is called a (directed) (s, t)-*cut*. By the appropriate version of the Menger theorems, a set of (s, t)-paths is maximum if and only if its cardinality equals the cardinality of a minimum (s, t)-cut. Such (s, t)-cuts are called *saturated*. There are two easy cases, for which the following lemma yields correctness of Algorithm 1.

LEMMA 1. *If either all or none of the arcs leaving s have been removed by Algorithm* 1, *the arcs leaving s that have not been removed form a saturated directed (s, t)-cut.*

Algorithm 1. Menger algorithm

```
for each outgoing arc $a$ of $s$ do
    let the search path consist of arc $a$
    while the leading vertex is neither $s$ nor $t$ do
        if the leading arc hits some passage from the right then
            realign the search path with the corresponding path or cycle
        else
            if there is an unsearched arc leaving the leading vertex then
                perform a search step
            else
                perform a backtracking step
```

PROOF. If none of the arcs leaving s has been removed, each of them is part of a distinct (s,t)-path, because no arc is entering s and Algorithm 1 does only produce (s,t)-paths and left cycles. For the other extreme, note that the number of (s,t)-paths found never decreases during execution of Algorithm 1. As long as no (s,t)-path has been found, every outgoing arc of a vertex on the search path is searched before the incoming searched arc is removed in a backtracking step. Therefore, right-first-search eventually finds a path to every vertex reachable from s and there is no (s,t)-path if every arc leaving s is removed. □

Three more observations are trivial and stated without proof:

LEMMA 2. *During the execution of Algorithm* 1, *the following statements hold*:

(a) *Just before a search or backtracking step, the leading arc does not hit any passage through the leading vertex from the right.*
(b) *Just before a search step, there is no irrelevant arc incident to the leading vertex and no removed arc entering the leading vertex.*
(c) *Just before a backtracking step, every arc leaving the leading vertex has already been searched.*

Correctness is based on the fact that the solution produced by Algorithm 1 is maximal and rightmost, i.e., no (s,t)-path can be routed further to the right without changing others, too. This notion of rightmost is characterized by the following four invariants:

LEMMA 3. *During the execution of Algorithm* 1, *the following properties remain invariant*:

(P1) *All paths and left cycles are arc disjoint and noncrossing.*
(P2) *No unsearched arc leaves a passage to the right.*
(P3) *No removed arc hits a passage from the right.*
(P4) *No two passages mutually touch their right sides.*

PROOF. Initially, the set of paths and cycles is empty and every condition is met. It is then sufficient to show that (P1)–(P4) are invariants of the *while*-loop, i.e., they remain satisfied after a single realignment, or a single search or backtracking step.

By (P1) and the fact that the search path is realigned only when it hits a passage through the leading vertex from the right, a realignment does not affect any of (P1)–(P4).

(P1) Clearly, no arc is assigned to two different paths or cycles at the same time. Searching does not cause a crossing because of Lemma 2(a) and (P2). By Lemma 2(a) and the way we close left cycles, no crossing is produced when an arc of the search path is removed.

(P2) By our choice of the new arc, (P2) remains valid after a search step. We need not consider backtracking because of Lemma 2(c).

(P3) Lemma 2(b) shows that (P3) remains satisfied after a search step. According to Lemma 2(a), backtracking does not violate (P3), since all reassigned arcs become irrelevant arcs.

(P4) By Lemma 2(a), we can safely add a new arc to the search path. All changes caused by backtracking involve only irrelevant arcs. □

Based on the above observations, Algorithm 2 determines a saturated directed (s, t)-cut in the output of Algorithm 1. This is obviously sufficient to prove correctness. From the discussion in Section 6, it is easy to see that Algorithm 2 can also be implemented with linear running time. It is based on the analogous variant of depth-first-search, left-first-search. Arcs removed or not searched by Algorithm 1 are searched in the forward direction, while arcs belonging to paths or cycles are searched in the backward direction.[6] When Algorithm 2 backtracks, the backtrack arc is said to be *discarded* from the search path. Algorithm 2 terminates when the search path hits itself from the left, such that the

Algorithm 2. Saturated cut algorithm

```
if all or none of the arcs leaving $s$ are removed then
    let the search path consist of $s$ only
else
    let the search path consist of a removed arc leaving $s$
    repeat
        if there is an unsearched candidate arc then
            search the clockwise next candidate arc
        else
            discard the leading arc from the search path
    until the search path consists of $s$ only
            and the clockwise next candidate arc has been searched
        or the clockwise next candidate arc is part of the search path
            and $s$ does not lie in the exterior of the resulting cycle
```

[6] Basically, the algorithm tries to find an augmenting path.

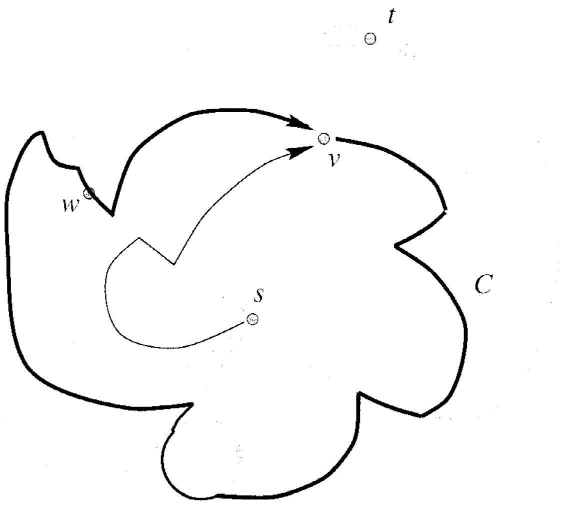

Fig. 4. Algorithm 1 has produced five (s, t)-paths and one left cycle, indicated by grey lines. In its output, Algorithm 2 terminates after returning to v_1. The final search path is depicted by black lines, with the resulting right cycle, C, drawn thicker. At vertex w, the search path of Algorithm 2 hit a search path from the right, but did not enter it (see Lemma 6).

resulting cycle is a right cycle[7] surrounding s. The arcs on (s, t)-paths having their tail on this cycle and their head in the exterior then form the desired cut. Figure 4 may serve to build an intuition.

To avoid confusion, an arc is denoted *ignored* if it was not searched by Algorithm 1. An arc is called a *candidate arc* if it leaves the leading vertex and was either removed or ignored, or if it enters the leading vertex and is part of an (s, t)-path or left cycle produced by Algorithm 1.

LEMMA 4. *Algorithm* 2 *never discards an arc from the search path that belongs to an* (s, t)*-path produced by Algorithm* 1.

PROOF. If the leading arc of Algorithm 2 belongs to an (s, t)-path produced by Algorithm 1, there either is a preceding arc on this path, or the leading vertex is s. In the latter case, the clockwise next candidate arc is searched next, or the algorithm terminates. □

LEMMA 5. *If an ignored arc is searched by Algorithm* 2, *it lies in the interior of one of the left cycles produced by Algorithm* 1.

PROOF. By Lemma 2(c), the head of a removed arc is never the tail of an ignored arc. Hence, the immediate predecessor of an ignored arc on the search path of Algorithm 2 must be an ignored arc, or the arc of an (s, t)-path or left cycle produced by Algorithm 1.

[7] Since some arcs are searched in the backward direction, right cycles are possible.

Since candidate arcs are searched in clockwise order, Lemma 4 and (P2) prove the claim. □

LEMMA 6. *The search path of Algorithm* 2 *never enters an* (s, t)*-path determined by Algorithm* 1 *from the right.*

PROOF. Assume that in the next step the search path enters an (s, t)-path of Algorithm 1 from the right. Then the current leading arc is either an ignored or removed arc hitting the (s, t)-path, or a path or cycle arc leaving it. Because, by Lemma 4, arcs of (s, t)-paths are never discarded, (P1), (P4), and the clockwise selection of candidate arcs imply that the leading arc does not belong to a passage through the leading vertex. From Lemma 2(a) it follows that irrelevant arcs appear only on the left-hand side of passages through the respective vertex, i.e., inside of a left cycle or to the left of an (s, t)-path. The leading arc is therefore neither a path nor a cycle arc because of (P1). Removed arcs are excluded by (P3), and since Lemma 5 states that ignored arcs are searched inside of left cycles only, (P1) and (P4) rule them out, too. □

After termination of Algorithm 2, the search path either consists of s alone, or it hits itself from the left while surrounding s. In the second case, let $C = (v_1, a_1, v_2, a_2, \dots, v_k = v_1)$ be the sequence of vertices and arcs of the search path beginning with the leading vertex and the clockwise next arc on the search path, and ending with the leading arc and the leading vertex. See Figure 4. In the first case, let C be the trivial cycle s.

LEMMA 7. *If C does not equal s, it is a right cycle with s in its interior or on its boundary.*

PROOF. Clearly, s does not lie in the exterior of C. If only some of the arcs leaving s are removed arcs, Algorithm 1 has found at least one (s, t)-path, because every arc that was neither removed nor ignored lies on an (s, t)-path or a left cycle and no arc is entering s. Thus, Lemma 6 and the clockwise selection of candidate arcs imply that C cannot be a left cycle. □

LEMMA 8. *The set of arcs on* (s, t)*-paths produced by Algorithm* 1 *having their tail on C and their head in its exterior form a saturated directed* (s, t)*-cut.*

PROOF. If the precondition in the first line of Algorithm 2 is true, we may apply Lemma 1. Therefore assume it is false.

Because no arc is leaving vertex t and no incoming arc of t is removed, Algorithm 2 can enter t on ignored arcs only. However, t lies on the outer face, and from Lemma 5 we can thus deduce that it is never reached. Using Lemma 7, we have that C is a right cycle with s in its interior or on its boundary. To verify that the arcs leaving C on (s, t)-paths form a directed cut, consider an arbitrary simple path $(s = v'_1, a'_1, \dots, v'_l = t)$ from s to t. Let $v'_i = v_j$ be its last vertex on the boundary of C, and assume that a'_i is a removed or ignored arc. a'_i must have been the clockwise next candidate arc of

Algorithm 2 some time that v_j was the leading vertex. Then a_i' was indeed searched and later discarded by Algorithm 2. Also, every arc reachable from a_i' was searched and discarded. By Lemma 4, no (s, t)-path of Algorithm 1 was hit in the meantime. (Note that it does not matter that left cycles produced by Algorithm 1 are searched the other way round.) This is a contradiction, because t was not reached. Neither can a_i' belong to a left cycle produced by Algorithm 1, because candidate arcs are chosen in clockwise order. Hence, a_i' must be the arc of an (s, t)-path. Since by Lemma 6 Algorithm 2 does not enter (s, t)-paths from the right, the clockwise choice of candidate arcs implies that every (s, t)-path has exactly one arc leaving the right cycle formed by C. □

The appropriate version of the Menger theorems now yields correctness of our algorithm.

COROLLARY 1. *The set of arc disjoint (s, t)-paths determined by Algorithm* 1 *is maximum.*

5. Properties of Partial Solutions. In Section 3 an algorithm solving the arc disjoint Menger problem in certain planar directed graphs was described. In this section we prove a number of invariants that are used to implement this algorithm efficiently.

Since our goal is to achieve linear running time, the possibly more than linear number of realignments cannot actually be performed. The subsequent analysis of the structure of partial solutions leads to an implementation that does not need to represent consecutiveness of arcs explicitly. Although the arc to be searched next is still computed easily, it can be difficult to identify a backtracking arc without knowing which arcs are consecutive. Therefore, two arcs are stored at each vertex in order to identify a set of arcs that is to the left of all passages through the vertex. Since Algorithm 1 realigns until the search path ends up on this left side, the implementation skips realignments and simply registers relevant changes to the left side.

A precursory structural insight is the relative orientation of passages. Two passages p, q through the same vertex are said to be *oriented likewise*, if p is completely to the left of q, while q is completely to the right of p. In Figure 3(b), passages p, q are oriented likewise, while passages p, r and q, r are oriented differently. The following lemma shows that at most the last v-passage of the search path can be oriented differently from other v-passages.

LEMMA 9. *During the execution of Algorithm* 1, *the following property remains invariant*:

(P5) *For all $v \in V \setminus \{s, t\}$, all v-passages are oriented likewise, possibly except for the leading v-passage.*

PROOF. Just like in the proof of Lemma 3, we only have to consider a single realignment, or a single search or backtracking step.

The search path is either realigned with an (s, t)-path, with a cycle, or with itself. First, let the search path be realigned with an (s, t)-path and assume that (P5) is not satisfied

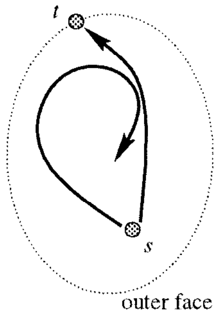

Fig. 5. If the search path touches a differently oriented (s, t)-path from the left, it does not hit an (s, t)-path from the right, because of (P1).

afterward. By induction, there has to be a vertex v such that before the realignment the leading v-passage is oriented differently than another v-passage it touches, while after the realignment it belongs to the (s, t)-path hit. By (P4) these two v-passages touch on their left sides, since differently oriented passages mutually touch on the same side. Therefore, by (P1), the touched v-passage does not belong to a left cycle, but to an (s, t)-path or orbit. However, then, again by (P1), the search path did not hit an (s, t)-path from the right (recall that t is on the outer face and note that there is no (s, t)-path, if there is an orbit). Figure 5 illustrates the situation of a touched (s, t)-path.

If the search path is realigned with itself (because it surrounds s), the considerations are almost the same.

If the search path is realigned with a cycle, the cycle does not surround s (i.e., it is not an orbit), because of (P1). Similar arguments thus show that any leading v-passage oriented differently than another v-passage still is a leading v-passage after the realignment.

Obviously, search steps do not affect the invariant, since only leading passages are changed. Note that the orientation of the new and augmented leading passage of the leading arc is the same as before in case the search path is extended after hitting itself from the left.

The only way a backtracking step can violate this invariant is by the creation of a left cycle which has, for some $v \in V \backslash \{s, t\}$, a v-passage that is oriented differently than another v-passage. In this case, every arc of the search path that is incident to the leading vertex either remains on the same passage or becomes irrelevant. Hence, this v is not the leading vertex. Just like above, the search path then has to touch a v-passage of an (s, t)-path or orbit on the left side. Because of (P1) and t being on the outer face, this is impossible. $\square$

If $v \in V \backslash \{s, t\}$ is on the search path, let *lastLeading*(v) be the last arc of the search path leaving v. By the above property, all but at most one specific v-passage (which then is the leading v-passage) of a vertex $v \in V \backslash \{s, t\}$ are oriented likewise. We define the *leftmost* v-passage to be the unique leftmost of these. Furthermore, let *firstLeft*(v) and *lastLeft*(v) be its first and last arc, respectively. Clearly, *lastLeading*(v) equals *lastLeft*(v), if and only if the leading v-passage is oriented like every other v-passage.

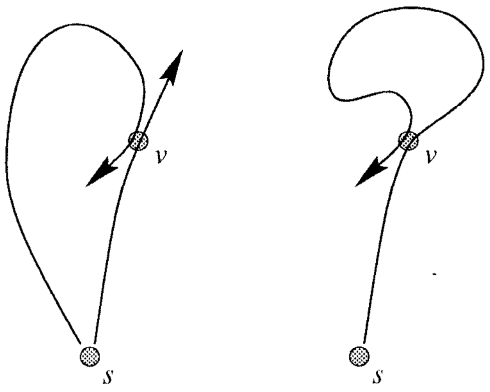

Fig. 6. In the left situation there is an (s, t)-path touched on the left by the search path, such that *lastLeft*$(v) \neq$ *lastLeading*(v) = *last*(v). In the right situation, the search path returns to v and we have *last*(v) = *lastLeading*(v) = *lastLeft*(v). Note that these situations cannot be distinguished solely based on the arcs incident to v and the order in which they were searched.

As an immediate, yet crucial, consequence of (P5) the following corollary states that knowledge of *lastLeft*(v) is sufficient to identify an incoming arc that may be used in a backtracking step (i.e., any arc of the search path entering the leading vertex).

COROLLARY 2. *During the execution of Algorithm* 1, *the following property remains invariant*:

(P6) *If $v \in V \setminus \{s, t\}$ is the leading vertex and the search path does not hit a v-passage from the right, then the counterclockwise next relevant arc after lastLeft(v) is an incoming arc of the search path.*

Unfortunately, it is difficult to keep track of *lastLeft*(v) efficiently (see Figure 6). We therefore maintain an arc *last*(v) that equals at least one of *lastLeft*(v) and *lastLeading*(v).

Until we need to backtrack from a vertex $v \in V \setminus \{s, t\}$ for the first time, *last*(v) is defined to be the arc incident to v that has most recently been added to the search path. By definition, it was equal to *lastLeading*(v) when it was added. After backtracking from v for the first time, *last*(v) is defined to be the clockwise next relevant outgoing arc after the backtracking arc entering v that was most recently removed.

The next lemma states that these definitions ensure the desired property of *last*(v), namely, that it is the last arc of one of the at most two v-passages that are to the left of all other v-passages.

LEMMA 10. *During the execution of Algorithm* 1, *the following property remains invariant*:

(P7) *For every $v \in V \setminus \{s, t\}$, last$(v)$ equals lastLeft(v) or lastLeading(v). If last(v) equals lastLeading(v), the leading v-passage is to the left of every other v-passage.*

PROOF. Again, we only consider a single step. Let $v \in V \backslash \{s, t\}$ be the leading vertex.

If the search path is realigned, we first consider those vertices $u \in V \backslash \{s, v, t\}$, for which $\mathit{last}(u) = \mathit{lastLeading}(u)$. If the search path is realigned with an (s, t)-path, the leading u-passage afterward belongs to the resulting (s, t)-path. However, since by induction the leading u-passage is to the left of every other u-passage, it cannot be oriented differently because of (P1). Hence, $\mathit{last}(u)$ equals $\mathit{lastLeft}(u)$. In case the search path is realigned with itself, similar arguments hold. If the leading u-passage is to the left of every other u-passage, a left cycle hit by the search path does not have a u-passage because of (P1) and (P4). Therefore, the leading u-passage remains leading when the search path is realigned with a left cycle. Now, consider the leading vertex v. If the search path is realigned, it is not to the left of every other v-passage. Therefore, $\mathit{last}(v) = \mathit{lastLeft}(v)$ by induction and neither of them is altered.

For a search step that does not make t the new leading vertex, we only have to consider the current leading vertex v. By definition, $\mathit{last}(v)$ becomes $\mathit{lastLeading}(v)$. By Lemma 2(a), the new leading v-passage is to the left of every other v-passage, and no other vertex is affected. If, in turn, the new leading vertex is t, the situation compares to a realignment with an (s, t)-path. Then it follows just like above that $\mathit{last}(u)$ also equals $\mathit{lastLeft}(u)$ for every vertex $u \in V \backslash \{s, t\}$ with $\mathit{last}(u) = \mathit{lastLeading}(u)$.

In case of a backtracking step, we need to treat the leading vertex v and those vertices u on the search path that after the step lie on a closed left cycle. If the leading passage of such a vertex u is to the left of every other u-passage, the leading passage is not oriented differently than the other u-passages because of (P1) and (P4). Thus, $\mathit{lastLeading}(u)$ equals $\mathit{lastLeft}(u)$ so that $\mathit{last}(u)$ equals $\mathit{lastLeft}(u)$ by induction. Now consider the leading vertex. By Lemma 2(a), the arc to be removed is to the left of every v-passage. Thus, (P1) and the absence of right cycles imply that the clockwise next relevant arc after the removed arc is outgoing. It becomes the new $\mathit{last}(v)$, which is thus also to the left of every v-passage. Moreover, it equals $\mathit{lastLeft}(v)$ or $\mathit{lastLeading}(v)$, since the removed arc is on the search path. In any case, a leading v-passage remains left of every other v-passage. □

Finally, we give a sufficient condition for $\mathit{last}(v)$ to equal $\mathit{lastLeft}(v)$. Depending on the current $\mathit{last}(v)$ of $v \in V \backslash \{s, t\}$, define $\mathit{first}(v)$ to be the counterclockwise first relevant arc after $\mathit{last}(v)$. Note that $\mathit{first}(v)$ equals $\mathit{firstLeft}(v)$, if $\mathit{last}(v)$ equals $\mathit{lastLeft}(v)$. In a cyclic clockwise traversal of the adjacency list of v, the *left side of* v is defined to be the set of arcs in the clockwise interval from $\mathit{first}(v)$ to $\mathit{last}(v)$, exclusively. Complementary, the *right side of* v is the set of arcs in the counterclockwise interval from $\mathit{first}(v)$ to $\mathit{last}(v)$, including both boundary arcs. See Figure 7.

LEMMA 11. *Let $v \in V \backslash \{s, t\}$ be the leading vertex at some stage of Algorithm* 1. *Suppose that, until now, backtracking from v has always been performed with the then leading arc. If the current leading arc is contained in the right side of v, then $\mathit{last}(v) = \mathit{lastLeft}(v)$.*

PROOF. Assume the preconditions are satisfied, but $\mathit{last}(v) \neq \mathit{lastLeft}(v)$. Then (P5) and (P7) imply that there is a leading v-passage that is to the left of every other v-passage

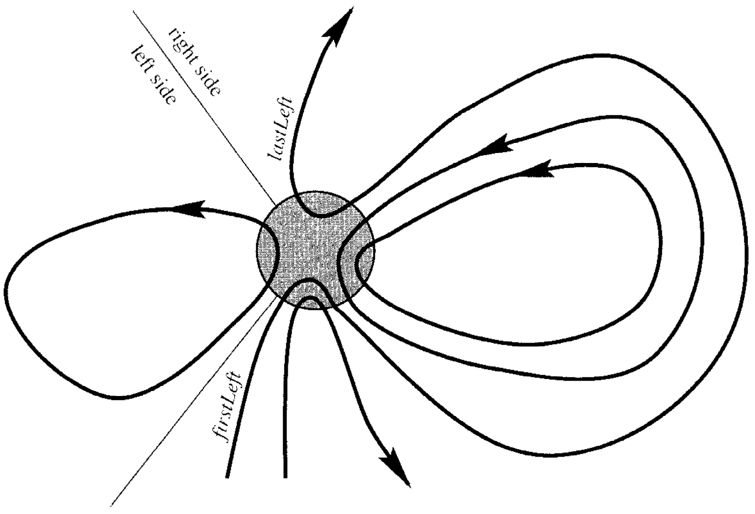

Fig. 7. The arcs contained in the left and right sides of v, respectively. Note that the right side of v contains the arcs of the leftmost v-passage and of two cycles that are to the passages left.

and oriented differently from every other v-passage. However, since the leading arc is contained in the right side of v, and since there are no right cycles, the search path also hits some v-passage from the right. This contradicts (P1). □

6. Linear Time Implementation. In this section we show that Algorithm 1 can be realized with linear running time. Since the number of changes caused by realignments can be more than linear in the number of arcs, it is crucial to, at least in general, avoid an explicit maintenance of consecutive arcs. Fortunately, due to the highly structured partial solutions generated by the algorithm, realignments need not be performed explicitly. We show that, in general, a potential search or backtracking arc Algorithm 1 chooses after realigning can be determined solely from the leading arc, *first*(v), and *last*(v).

In each iteration of the main loop, the search path is initialized with the particular arc leaving s, and the inner loop is executed. The implementation of the inner loop consists of intended search and backtracking steps only. In a search step, the algorithm tries to extend the search path by an unsearched arc leaving the leading vertex, while in a backtracking step, it removes the current leading arc. Let v denote the leading vertex in a search step, and the tail of the leading vertex in a backtracking step, respectively. If $v \in V \backslash \{s, t\}$, four situations are distinguished.

Forward Mode. Vertex v is in *forward mode*, if there are unsearched arcs leaving v. No matter whether the current step is a search backtracking step, we let the counterclockwise next unsearched arc after the current leading arc be the new leading arc. By (P1) and (P2), this arc is exactly the arc that Algorithm 1 chooses after all necessary realignments. If there are no unsearched outgoing arcs remaining, v enters transition mode (see below), and each outgoing arc of v is assigned its preceding incoming arc. This is easily done according to (P1), (P5), and the fact that *last*(v) is the last arc of some v-passage. Finally, another search step is initiated.

Transition Mode. When all outgoing arcs are searched, but *lastLeft*(v) is not yet determined, vertex v is in *transition mode*. If the leading arc is contained in the right side of v, *last*(v) equals *lastLeft*(v) by Lemma 11, so that we let v enter skip mode (see below). Otherwise the leading arc is contained in the left side of v. In a search step, the search path cannot be extended. Hence a backtracking step removing the leading arc is invoked. In a backtracking step, the temporarily stored preceding arc that was computed at the end of the forward mode is used for backtracking in order to satisfy the precondition in Lemma 11. Note that no reassignment of consecutive arcs is necessary for a vertex in transition mode. Following their definition, *last*(v) and *first*(v) are updated to be the clockwise and counterclockwise next relevant arc, respectively, after the new leading arc.

Skip Mode. At the end of the transition mode, *last*(v) equals *lastLeft*(v) by Lemma 11. From that time, v is in *skip mode* until its last outgoing arc is removed or the algorithm terminates. When its last outgoing arc is removed, v finally enters done mode (described below). Since there are no unsearched outgoing arcs left at vertices in skip mode, an arc to backtrack within the next step must be determined, no matter whether the current step is a search or backtracking step.

The arcs contained in the left side of v are never involved in a realignment of Algorithm 1, since they are either unsearched and incoming, irrelevant, or belong to the search path. We therefore maintain a fixed assignment of predecessors for the outgoing arcs of the left side. If the leading arc is contained in the left side of v, it is used for backtracking when the current step is a search step. Its stored predecessor is used in case the current step is a backtracking step.

If the leading arc is contained in the right side of v, the realignments of Algorithm 1 cause some subpath of the leftmost v-passage to become the new search path. Therefore, *first*(v) is suitable for backtracking because of Corollary 2 and Lemma 11. According to (P1), the closing of left cycles contained in the new left side is easily performed in the following way: Starting right after *first*(v), the adjacency list of v is traversed counterclockwise. Every outgoing arc is placed on a stack, whereas every incoming arc becomes the predecessor of the topmost outgoing arc, until an incoming arc is encountered while the stack is empty. Finally, *first*(v) is updated to be this last incoming arc, which must be relevant. In case of a backtracking step with leading arc *last*(v), a similar procedure is also used to update *last*(v) and close left cycles in the resulting new interval of the left side of v. Note that the case of no remaining relevant arc is easily recognized.

Done Mode. If v has no (remaining) outgoing arc at all, it is said to be in *done mode*. In a search step, backtracking with the leading arc is initiated. Note that there cannot be a backtracking step with v the tail of the leading arc.

A formal description of the main loop is given in Algorithm 3. Variable `global_mode` determines whether the next intended step is a search or a backtracking step. Variable `leading_arc` stores the leading arc. Vertices and arcs are represented by records

Algorithm 3. Menger implementation

```
for each a ∈ $A$ do a.flow := 0
for each v ∈ $V\setminus\{s,t\}$ do
   if v *has no outgoing arcs* then
      v.mode := DONE
   else
      v.mode := FORWARD

for each *outgoing arc* a *of* s do
   leading_arc := a
   leading_arc.flow := 1
   global_mode := SEARCH
   while (global_mode = SEARCH and head(leading_arc) ≠ t) or
         (global_mode = BACKTRACK and tail(leading_arc) ≠ s) do
      case global_mode of
         SEARCH:    search
         BACKTRACK: backtrack
   if global_mode = BACKTRACK then
      *remove* leading_arc *from graph*
```

containing fields of data. Field `v.mode` is used to store the mode of vertex v. Every arc a that is not removed from the graph has a field `a.flow` containing either `0` or `1`. It has value `1` if and only if the arc is occupied and hence belongs to a path or cycle.

The search step is formally described in Algorithm 4. An additional field `v.last` is used to store *last*(v) for every vertex $v \in V\setminus\{s,t\}$. When v enters transition mode, the current pairs of consecutive arcs are computed by a subroutine `match_all` computing the predecessor `a.pred` of every outgoing arc a of v. According to (P1) and the fact that *last*(v) is the last arc of some v-passage, a simple stack algorithm is sufficient. Field `v.first` is introduced to store *first*(v). When `v.first` is used for backtracking according to Corollary 2, the field must be updated. Every arc between the former and the new `v.first` is irrelevant. Subroutine `match_left` realizes the update of `v.first` and matches outgoing arcs newly contained in the left side of v with their predecessors.

Algorithm 5 implements the backtracking step. If an arc removed in skip mode equals *lastLeft*(u), the arcs newly contained in the left side of u are matched in subroutine `match_right`, which is analogous to `match_left`. Observe that the arc that is removed during a backtracking step has `flow 1`. Thus, its tail u cannot be in local mode `DONE`.

THEOREM 1. *Algorithm* 3 *determines a maximum set of arc disjoint* (*noncrossing*) (s,t)-*paths of G in linear time.*

PROOF. From the discussion above we see that Algorithm 3 is indeed an implementation of Algorithm 1 and therefore computes a maximum solution (Corollary 1).

For the linear running time, observe that every time an arc is used, its state is altered from unsearched (`flow 0`) to searched (`flow 1`), or from searched to removed (no longer present). The predecessor of an arc is computed at most twice, and because of (P1)

Algorithm 4. Procedure `search`

```
v := head(leading_arc)
if there is an outgoing arc of v with flow 0 then
  leading_arc := first arc after leading_arc
                   in counterclockwise order around v
                   that is outgoing and has flow 0
  leading_arc.flow := 1
  v.last := leading_arc
else
  case v.mode of
    FORWARD:
      match_all(v)
      v.first := first arc after v.last
                  in counterclockwise order around v
                  that is incoming with flow 1
                  and does not equal leading_arc
      v.mode := TRANSITION
    TRANSITION:
      if leading_arc is to the right of {v.first,v.last} then
        v.mode := SKIP
      else
        global_mode := BACKTRACK
    SKIP:
      if leading_arc is to the right of {v.first,v.last} then
        leading_arc := v.first
        match_left(v)
      global_mode := BACKTRACK
    DONE:
      global_mode := BACKTRACK
```

a simple stack algorithm matches consecutive arcs in linear time. Thus it is sufficient to show that, computation of consecutive arcs not accounted for, a single search or backtracking step can be implemented with constant amortized running time. The only critical operation of a search step is the determination of a counterclockwise next arc after the current leading arc. It was shown in [WW] how Gabow and Tarjan's technique for the efficient implementation of certain *union-find* structures [GT] can be adapted to determine this arc in constant amortized time. The corresponding operation needed during a backtracking step can be performed on the same data structure. Verify that during all updates of fields `v.first` and `v.last` in modes `FORWARD` and `TRANSITION` every incident arc of a vertex v needs to be traversed at most once. □

7. Discussion. The Menger problem has four basic variants: edge disjoint (s,t)-paths in undirected graphs, arc disjoint (s,t)-paths in directed graphs, vertex disjoint (s,t)-paths in undirected graphs, and vertex disjoint (s,t)-paths in directed graph. For planar

Algorithm 5. Procedure `backtrack`

```
u := tail(leading_arc)
dummy := leading_arc
if leading_arc is the only outgoing arc of u then
   leading_arc := the incoming arc with flow 1
   u.mode := DONE
else if there is an outgoing arc of u with flow 0 then
   leading_arc := first arc after leading_arc
                  in counterclockwise order around u
                  that is outgoing and has flow 0
   leading_arc.flow := 1
   u.last := leading_arc
   global_mode := SEARCH
else
   case u.mode of
      FORWARD:
         match_all(u)
         u.first := first arc after u.last
                    in counterclockwise order around u
                    that is incoming with flow 1
                    and does not equal leading_arc
         u.mode := TRANSITION
      TRANSITION:
         if leading_arc = u.last then
            leading_arc := leading_arc.pred
            u.last := first arc after leading_arc
                      in clockwise order around u
                      that is outgoing and has flow 1
            u.first := first arc after leading_arc
                       in counterclockwise order around u
                       that is incoming and has flow 1
         else
            u.mode := SKIP
      SKIP:
         if leading_arc is to the left of {u.first,u.last} then
            leading_arc := leading_arc.pred
         else
            if leading_arc = u.last then
               match_right(u)
            leading_arc := u.first
            match_left(u)
remove dummy from graph
```

graphs, three of these four cases have been solved to optimality by the algorithms in [W2], [RWW2], and in this paper.

The linear time algorithm for the edge disjoint Menger problem in planar undirected graphs transforms an undirected input graph into a directed graph [W2]. Each undirected edge is replaced by two arcs, one for either direction. Even though right cycles are eliminated as described in Section 2, this results in a very special directed graph. A right-first-search without backtracking is used to find a maximum number of (s, t)-paths. Backtracking is never needed, since every time a vertex is entered by the search path, there must be an unsearched outgoing arc. It is precisely the potential need to backtrack which makes the directed version much more difficult.

By now, in planar graphs no linear time solution is known only for the directed vertex disjoint Menger problem. From the undirected versions of the problem one may draw the conclusion that a linear algorithm for this problem might be more difficult to find (assuming there is one at all), since approaches using right-first-search run into difficulties when right cycles are present in the graph. In [KNK] it was argued that in the case of vertex capacities the set of maximum flows does not have a lattice structure. However, it was precisely this structure that allowed an easy restriction to planar graphs without right cycles. In other terms, it appears to be more difficult to resolve the problems caused by right cycles in the case of vertex disjointness.

Acknowledgments. The authors would like to thank Annegret Liebers and Karsten Weihe for their helpful comments and suggestions.

References

[AMO] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[GT] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.*, 30:209–221, 1985.

[HT] John E. Hopcroft and Robert E. Tarjan. Efficient planarity testing. *J. Assoc. Comput. Mach.*, 21:549–568, 1974.

[IS] Alon Itai and Yossi Shiloach. Maximum flows in planar networks. *SIAM J. Comput.*, 8:135–150, 1979.

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

[KNK] Samir Khuller, Joseph (Seffi) Naor, and Philip Klein. The lattice structure of flow in planar graphs. *SIAM J. Discrete Math.*, 6(3):477–490, 1993.

[KRR$^+$] Philip Klein, Satish B. Rao, Monika Rauch, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. System Sci.*, 55(1):3–23, 1997.

[M] Karl Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, 10:95–115, 1927.

[RWW1] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. Efficient algorithms for disjoint paths in planar graphs. In William Cook, Laszlo Lovász, and Paul Seymour, editors, *Combinatorial Optimization*: *Papers from the DIMACS special year*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, volume 20, pages 295–354. American Mathematical Society, Providence, RI, 1995.

[RWW2] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. The vertex-disjoint Menger problem in planar graphs. *SIAM J. Comput.*, 26:331–349, 1997.

[W1] Karsten Weihe. Maximum (s,t)-flows in planar network in $O(n \log n)$ time. In *Proceedings of the* 35*th Annual Symposium on Foundations of Computer Science*, *FOCS* '94, pages 178–189, 1994.

[W2] Karsten Weihe. Edge-disjoint (s,t)-paths in undirected planar graphs in linear time. *J. Algorithms*, 23:121–138, 1997.

[WW] Dorothea Wagner and Karsten Weihe. A linear time algorithm for edge-disjoint paths in planar graphs. *Combinatorica*, 15:135–150, 1995.