A Linear Time Algorithm for the Arc Disjoint Menger Problem in Planar Directed Graphs
(Extended Abstract)

Ulrik Brandes and Dorothea Wagner

Universität Konstanz, Fakultät für Mathematik und Informatik
Fach D 188, D-78457 Konstanz, Germany
{Ulrik.Brandes , Dorothea.Wagner}@uni-konstanz.de

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 [Wei97], while the fastest algorithm for the arc disjoint version is Weihe's general maximum flow algorithm for planar networks [Wei94], 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.

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 [Men27] 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 [RLWW97] and edge disjoint case [Wei97]. 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 [AMO93]. The first algorithm tailored to solve the maximum flow problem with arbitrary capacities especially in planar graphs was presented in [IS79]. Faster algorithms have subsequently been developed, e.g. [JV82] and

[KRRHS94]. By now, the fastest algorithm is that of [Wei94] yielding a running time of $\mathcal{O}(n \log n)$, where $n = |V|$. Here we concentrate on the more special Menger problem 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 (cf. [RLWW95]). In particular, the optimal algorithms for the Menger problem in undirected graphs are based on this variant of depth-first-search [RLWW97,Wei97]. In a right-first-search, arcs are chosen according to a *right-hand-rule*, i.e. the continuation arc is the counterclockwise next in the adjacency list of the vertex entered by the current arc. One of the main difficulties encountered by this strategy is the treatment of right cycles. Similar to [Wei97], we therefore use an observation of [KNK93] 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 Sect. 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 Sect. 4. In Sect. 5, properties of partial solutions are examined. Based on these properties, a linear time implementation of the algorithm is described in Sect. 6.

2 Preliminaries

Let us 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 will 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, e.g. 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 *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 labelling 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 $f_{\mathcal{C}} : A_{\mathcal{C}} \rightarrow \{0,1\}$ is a maximum integral flow in the residual graph $G_{\mathcal{C}}$, then a maximum integral flow $f : A \rightarrow \{0,1\}$ 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 [KNK93] it can be seen that there always exists a set $\mathcal{C}$ of right cycles such that the residual graph $G_{\mathcal{C}}$ contains left cycles only[1]. Moreover, this special set $\mathcal{C}$ can always be found in linear time using a breadth-first-search in the planar dual of G, which can also be obtained in linear time. Thus, a linear time algorithm solving the arc disjoint Menger problem in $G_{\mathcal{C}}$ is sufficient to provide a linear time solution for the problem in G. This technique was also used in [Wei97], where further details can be found.

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) face[2] and contains no right cycle. We may further assume that there are 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 non-simple) arc disjoint (s,t)–paths in an embedded planar directed graph with t on the outer face and no right cycle (cf. the previous section). 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, namely right-first-search, which is suitable for many problems involving paths in planar graphs [RLWW95]. 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.

[1] This set $\mathcal{C}$ corresponds to the *bottom element* of the distributive lattice formed by all circulations of G with appropriately defined operations for *meet* and *join* [KNK93].

[2] Such an embedding can always be obtained in linear time [HT74].

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 alternatingly when traversing the cyclic order of arcs incident to v. See Fig. 1. A set of arc disjoint paths and cycles is said to be *non-crossing*, if no two pairs of consecutive arcs form a crossing. All partial solutions will be non-crossing.

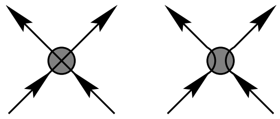


Fig. 1. Crossing (left) and non-crossing (right) pairs of consecutive arcs

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. Which arc exactly need not be specified in this general version of the algorithm. In the implementation presented in Sect. 6, this choice is subject to certain configurations and the stage of the algorithm. If a non-simple 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 left-over 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 Fig. 2 and note that there is a unique reassignment of consecutive arcs satisfying these conditions.

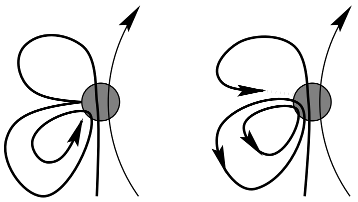
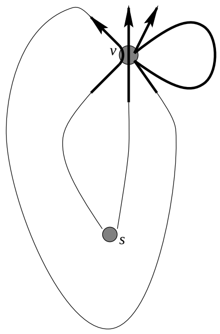


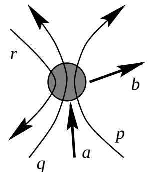
Fig. 2. After backtracking with the second arc of the search path entering the leading vertex, there are two resulting left cycles (containing four irrelevant arcs incident to the formerly leading vertex). The removed arc is indicated by the dotted line.

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*. Then, every arc that belongs to a path or cycle and is not irrelevant is called *relevant*[3].

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 non-simple, then s is in the exterior of every cycle formed by the subpath. Fig. 3 (a) gives an example. If existent, the last v–passage of the search path is called *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 sides 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. Fig. 3 (b) summarizes these definitions.



(a)



(b)

Fig. 3. (a) Two paths making up for three v–passages. (b) Three passages p, q, r, and two arcs a, b incident to the same vertex. Passage p touches passage q on the right, r touches q on the left, while 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. a and b are not consecutive.

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 counter-clockwise cyclic order of arcs incident to v. The search path is said to be realigned with the corresponding path or cycle, if the leading arc is made consecutive with (v,w), such that (u,v) becomes the new leading arc.

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 two-fold: on one hand, they prevent crossings, and on the other hand, they ensure that the search path is, in a sense, leftmost at the leading vertex when backtracking has to be performed.

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

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

---

4 Correctness

In this section, we outline the steps to prove that after termination of Algorithm 1 the set of (s,t)–paths generated is maximum. The proofs themselves are given in the full version of this paper [BW97]. A set of arcs whose removal disconnects s and t is called a (directed) (s,t)*–cut*. By Menger's theorem, 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*. Three observations are trivial:

Lemma 1. *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.

There are two easy cases, for which the following lemma yields correctness of Algorithm 1.

Lemma 2. *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.*

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 non-crossing.
(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.

Based on the above observations, Algorithm 2 determines a saturated directed (s,t)–cut in the output of Algorithm 1. This cut is induced by the exterior of a cycle enclosing s. This obviously suffices to prove correctness. From the discussion in Sect. 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 forward direction, while arcs belonging to paths or cycles are searched in backward direction.[4] 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 resulting cycle is a right cycle[5] 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.

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
```

To avoid confusion, an arc is denoted *ignored*, if it was not searched by Algorithm 1. An arc is then 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.*

[4] Basically, the algorithm tries to find an augmenting path.
[5] Since some arcs are searched in backward direction, right cycles are possible.

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.*

Lemma 6. *The leading arc of Algorithm 2 never enters an (s, t)–path produced by Algorithm 1 from the right.*

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 $C = (v_1, a_1, v_2, a_2, \ldots, v_k = v_1)$ 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. 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.*

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.*

Corollary 9. *The set of arc disjoint (s, t)–paths determined by Algorithm 1 is maximum.*

5 Properties of Partial Solutions

In Sect. 3, an algorithm solving the arc disjoint Menger problem in planar directed graphs was described. In this section now, we state a number of invariants that are used to efficiently implement this algorithm. Again, details are given in the full version of this paper [BW97].

Since our goal is to achieve linear running time, the possibly more than linear number of realignments cannot actually be performed. While the arc to be searched next is still computed easily, it can be difficult to identify an arc on the search path that may be used for backtracking, when it is not known which arcs are consecutive. We subsequently analyze the structure of partial solutions. This will permit an implementation that does not need to explicitly represent which arcs are consecutive.

A first 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 Fig. 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 different than other v–passages.

Lemma 10. *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.*

By the above property, all but at most one specific v–passage (which then is the leading v–passage) of a vertex $v \in V \setminus \{s,t\}$ are oriented likewise. We define the *leftmost* v–passage to be the unique leftmost of these, and *lastLeft*(v) to be its last arc. As an immediate, yet crucial, consequence of (P5) the following corollary states that knowledge of *lastLeft*(v) is sufficient to identify an arc that may be used in a backtracking step (i.e. an arbitrary arc of the search path entering the leading vertex).

Corollary 11. *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.*

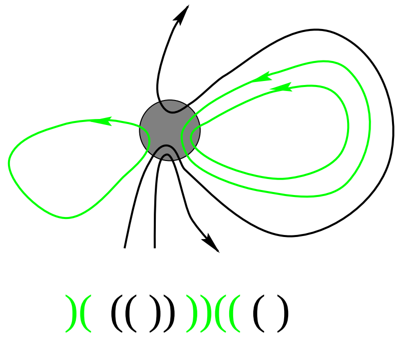


Fig. 4. A vertex v and its associated parenthesis expressions. $M(v)$ is depicted darker than $m(v)$, *lastLeft*(v) is the arc leaving v upwards.

We can gain even further knowledge from arc *lastLeft*(v): If the relevant arcs incident to v are traversed in counterclockwise order such that *lastLeft*(v) is encountered last, define $M(v)$ to be a string of parentheses, one opening for each incoming relevant arc, and one closing for each outgoing relevant arc of v. If v is the leading vertex, the leading arc does not contribute to $M(v)$. Analogously, let $m(v)$ be the string of parentheses for the irrelevant arcs. We say that a string of parentheses has *parenthesis structure*, if the number of closing parentheses does not exceed the number of opening parentheses in any prefix of the string. Likewise, it has *inverse parenthesis structure*, if the number of closing parentheses in a prefix is never less than the number of opening parentheses. Two strings of (possibly inverse) parenthesis structure are called *interleaved* (in a common superstring), if no parenthesis of one string is positioned between a pair of matching parentheses in the other.

The following property is not only useful to recompute pairs of consecutive arcs from *lastLeft*(v), but interesting in its own right.

Lemma 12. *During the execution of Algorithm 1, the following property remains invariant:*

> *(P7) For $v \in V \setminus \{s,t\}$, $M(v)$ has parenthesis structure, $m(v)$ has inverse parenthesis structure, $M(v)$ and $m(v)$ are interleaved, and every pair of consecutive relevant or irrelevant arcs incident to v corresponds to a pair of matching parentheses in $M(v)$ or $m(v)$, respectively.*

Even though *lastLeft*(v) provides all the information needed to implement Algorithm 1 efficiently, we are not yet done, because it is not always possible to update *lastLeft*(v) correctly based on local knowledge only (an example is given in Fig. 5).

Define *lastLeading*(v) to be the last arc of the search path leaving v. We now give a simple rule to mark an arc *last*(v), which equals at least one of *lastLeft*(v) and *lastLeading*(v). Consider, for a vertex $v \in V \setminus \{s,t\}$ that is not the leading vertex, the arc that has most recently been searched or removed. If it has been searched, it is outgoing and we let *last*(v) be just this arc. If it is removed, it is incoming, and we let *last*(v) be the clockwise next relevant outgoing arc. Now, if v is the leading vertex, it will be clear from context, whether *last*(v) refers to the current or next arc with the above properties. Furthermore we define *first*(v) to be the incoming relevant arc corresponding to the opening parenthesis immediately after *last*(v) in a cyclic traversal of $M(v)$.

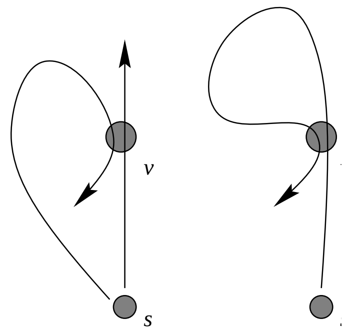


Fig. 5. In the left situation there is an (s,t)–path touched on the left by the search path. Here, *lastLeft*(v) is the arc leaving v on the (s,t)–path, and hence *lastLeft*$(v) \neq$ *lastLeading*(v) $=$ *last*(v). In the right situation, the search path returns to v, and we have *last*(v) $=$ *lastLeading*(v) $=$ *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.

The next lemma states that *last*(v) is the last arc of the v–passage that is to the left of all likewise oriented v–passages. Recall from (P5) that at most the leading v–passage is oriented differently, and observe that *lastLeft*(v) equals *lastLeading*(v), if and only if v has a leading v–passage that is to the left of every other v–passage and oriented likewise.

Lemma 13. *During the execution of Algorithm 1, the following property remains invariant:*

> *(P8) 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.*

Our final lemma gives a sufficient condition for *last*(v) to equal *lastLeft*(v).

Lemma 14. *Let $v \in V \setminus \{s,t\}$ be the leading vertex. If the leading arc is positioned between* first(v) *and* last(v) *in a cyclic traversal of $M(v)$, then* last$(v) =$ lastLeft(v)*.*

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 avoid an explicit maintainance of consecutive arcs, at least in general. Fortunately, due to the highly structured partial solutions generated by the algorithm, realignments need not be performed explicitly. We show that the arc to be searched next can be determined from the current leading arc, while an arc to backtrack with can be determined from *first*(v) and *last*(v). Sometimes arcs need to be paired according to a substring of $M(v)$ and $m(v)$. Therefore, procedures `match_all`, `match_left`, and `match_right` compute the preceding arc `a.pred` for outgoing arcs a of a particular vertex, where the first one matches all pairs of arcs, while the other match maximal substrings of $m(v)$ at the left and right end of $M(v)$, respectively.

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 search and backtracking steps only. A variable `global_mode` takes values `SEARCH` or `BACKTRACK`, and is used to determine which step is to be performed next. For a search step, consider the next arc that is searched. By (P2), it is the counterclockwise next unsearched arc after the leading arc, even if no realignment is performed. Algorithm 3 is a more formal description of possible cases. For each vertex $v \in V \setminus \{s,t\}$, a field `v.mode` is used to store a *local mode*, which is either `FORWARD` (v still has outgoing unsearched arcs), `TRANSITION` (all outgoing arcs are searched, but *lastLeft*(v) is not yet identified), `SKIP` (all outgoing arcs are searched and *lastLeft*(v) = *last*(v)), or `DONE` (all outgoing arcs of the vertex are removed).

For a backtracking step, consider the step immediately afterwards. Let (u,v) be the arc that is removed, then the next step is either a search or a backtracking step with leading vertex u. If it is a search step, we again have that the counterclockwise next unsearched arc after (u,v) is the next unsearched arc chosen by Algorithm 1 (possibly after a number of realignments). If it is a backtracking step, an arc of the search path must be determined that is removed next. When it is known that *last*(u) equals *lastLeft*(v), this is done according to Corollary 11, otherwise the preceding arc of (u,v) is retrieved from a temporary assignment of consecutive arcs. According to (P7), this can easily be computed the first time that there is no outgoing arc at u. By Lemma 14 it is only needed for a certain transition phase, in which there is no (not even implicit) realignment at u. Algorithm 4 gives a more formal description.

Theorem 15. *A maximum set of arc disjoint (non-crossing) (s,t)–paths of G can be determined in linear time.*

Proof. Above, we have described an implementation of Algorithm 1, which by Corollary 9 computes a maximum solution. For its linear time realization, 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

Algorithm 3 Search Step

```
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
            inverse_match_left(v)
         global_mode := BACKTRACK
      DONE:
         global_mode := BACKTRACK
```

predecessor of an arc is computed at most twice, and because of (P7) 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 the counterclockwise next arc after the current leading arc. It was shown in [WW95] how Gabow and Tarjan's technique for the efficient implementation of certain *union-find*-structures [GT85] 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. One easily verifies that during the update of fields `first` and `last` in modes `FORWARD` and `TRANSITION` every incident arc of a vertex v needs to be traversed at most once. □

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

Algorithm 4 Backtrack Step

```
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
               inverse_match_right(u)
            leading_arc := u.first
            inverse_match_left(u)
remove dummy from graph
```

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows*. Prentice Hall, 1993.

[BW97] Ulrik Brandes and Dorothea Wagner. A linear time algorithm for the arc disjoint Menger problem in planar directed graphs. Konstanzer Schriften in Mathematik und Informatik 29, Universität Konstanz, 1997.

[GT85] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. of Computer and System Sciences*, 30:209–221, 1985.

[HT74] John E. Hopcroft and Robert E. Tarjan. Efficient planarity testing. *J. of the Association for Computing Machinery*, 21:549–568, 1974.

[IS79] Alon Itai and Yossi Shiloach. Maximum flows in planar networks. *SIAM J. Comput.*, 8:135–150, 1979.

[JV82] David S. Johnson and S.M. Venkatesan. Using divide and conquer to find flows in directed planar networks in $\mathcal{O}(n^{3/2}\log n)$ time. In *Proceedings 20th Ann. Allerton Conf. Comm., Control, and Comp.*, pages 898–905, 1982.

[KNK93] 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.

[KRRHS94] Philip Klein, Satish B. Rao, Monika Rauch-Henzinger, and S. Subramanian. Faster shortest–path algorithms for planar graphs. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing, STOC'94*, pages 27–37, 1994.

[Men27] Karl Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, 10:95–115, 1927.

[RLWW95] 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, *DIMACS Series in Discrete Mathematics and Computer Science*, volume 20, pages 295–354. American Mathematical Society, 1995.

[RLWW97] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. The vertex-disjoint Menger problem in planar graphs. *SIAM J. Comput.*, 1997. to appear.

[Wei94] Karsten Weihe. Maximum (s,t)–flows in planar network in $O(n\log n)$ time. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, FOCS'94*, pages 178–189, 1994.

[Wei97] Karsten Weihe. Edge–disjoint (s,t)–paths in undirected planar graphs in linear time. *J. of Algorithms*, 23:121–138, 1997.

[WW95] Dorothea Wagner and Karsten Weihe. A linear time algorithm for edge-disjoint paths in planar graphs. *Combinatorica*, 15:135–150, 1995.