COMBINATORICA
Akadémiai Kiadó – Springer-Verlag

COMBINATORICA **15** (1) (1995) 135–150

A LINEAR-TIME ALGORITHM FOR EDGE-DISJOINT PATHS IN PLANAR GRAPHS*

DOROTHEA WAGNER and KARSTEN WEIHE

Received March 30, 1993
Revised November 25, 1994

In this paper we discuss the problem of finding edge-disjoint paths in a planar, undirected graph such that each path connects two specified vertices on the boundary of the graph. We will focus on the "classical" case where an instance additionally fulfills the so-called *evenness-condition*. The fastest algorithm for this problem known from the literature requires $O\left(n^{5/3}(\log\log n)^{1/3}\right)$ time, where n denotes the number of vertices. In this paper now, we introduce a new approach to this problem, which results in an $O(n)$ algorithm. The proof of correctness immediately yields an alternative proof of the Theorem of Okamura and Seymour, which states a necessary and sufficient condition for solvability.

1. Introduction

Let $G=(V,E)$ be an undirected, planar graph. A *net* $\{s,t\}$ is a pair of vertices $s,t\in V, s\neq t$, such that both s and t, the *terminals* of net $\{s,t\}$, are on the boundary of G, that is, incident to a fixed *outer* face. An instance of the problem we consider in this paper is a pair (G,N), where $N=\left\{\{s_1,t_1\},\dots,\{s_k,t_k\}\right\}$ is a set of nets of G. Additionally, the so-called *evenness-condition* is fulfilled, which means that the extended graph $\left(V,E+\{s_1,t_1\}+\dots+\{s_k,t_k\}\right)$ is Eulerian. The problem is to decide whether there are edge-disjoint paths $p_1,\dots,p_k$, such that p_i connects s_i with t_i for $i=1,\dots,k$, and if so, to determine such a set of paths.

The asymptotically fastest algorithm known so far is due to Kaufmann and Klär and requires $O\left(n^{5/3}(\log\log n)^{1/3}\right)$ time [9], where $n=|V|$. This algorithm relies on an $O(n^2)$ algorithm introduced by Becker and Mehlhorn [1] and uses decomposition techniques of Frederickson [2]. In this paper, we now give a *linear time* algorithm.

Becker and Mehlhorn also considered a certain generalization, which arises in VLSI layout. There, only the *weak evenness-condition* must be fulfilled, which means that each vertex *not* incident to the outer face has even degree. They propose an $O(bn+T(n))$ algorithm, where b denotes the number of vertices on the boundary of G and $T(n)$ the time required to solve any instance fulfilling the strong evenness-condition. Clearly, in connection with our result, this immediately yields

Mathematics Subject Classification (1991): 05 C 85, 68 Q 35

* The authors acknowledge the *Deutsche Forschungsgemeinschaft* for supporting this research under grant *Mö 446/1-3*

an $O(bn)$ algorithm for that more general problem. The relation to VLSI layout is not surprising at all since any grid graph without internal holes fulfills at least the weak evenness-condition.

The "capacitated" version of the problem is solved by Hassin [5], by Matsumoto, Nishizeki and Saito [12], and by Weihe [17]. Note that the problem of finding edge-disjoint paths between designated pairs of vertices can be seen as an integral multicommodity flow problem with unit capacities and unit supplies/demands. In the capacitated version, the restriction to *unit* values is dropped, and now the evenness condition means that, for any vertex, the sum of the capacities of the incident edges *plus* the sum of all supplies and demands arising at this vertex is an even number. The algorithms require $O(n^4)$, $O(kn+n^2)$, and $O(kn)$ time, respectively. The last two algorithms use a recent result by Klein, Rao, Rauch, and Subramanian on shortest paths to achieve the respective complexity [11]. (Actually, the earlier result of Frederickson [2] suffices for the algorithm by Nishizeki et al.)

The algorithm in [17] is based on the approach that we are going to introduce now. All other algorithms cited above, as well as many other algorithms tailored to special cases and similar problems (e.g. [8, 10, 13]) are closely related to the *Theorem of Okamura and Seymour* and to its highly constructive proof [14]. This theorem gives a necessary and sufficient condition for solvability. To state it, we need some terminology.

Let (G,N) be an instance of the problem. A *cut* is simply a set $X \subset V$ of vertices with $X \neq \emptyset$ and $V \setminus X \neq \emptyset$. The *capacity* of X, $cap(X)$, is the number of edges having one endvertex in X and the other one in $V \setminus X$. The *density* of X, $dens(X)$, is the number of nets $\{s_i,t_i\} \in N$ with one terminal in X and the other one in $V \setminus X$. The *free capacity* is defined to be $fcap(X) = cap(X) - dens(X)$. A cut X is called *saturated*, if $fcap(X) = 0$, and *oversaturated*, if $fcap(X) < 0$. Obviously, if (G,N) is solvable, no cut is oversaturated *(cut condition)*. Okamura and Seymour have shown that the cut condition is also sufficient [14].

Theorem 1.1. (Okamura–Seymour, 1981) *An instance is solvable if and only if the cut condition holds.*

With the following Lemma 1.2, Okamura and Seymour have strengthened their result further, which is crucial for all algorithms relying on Theorem 1.1. A cut X is *essential*, if the subgraphs induced by X and $V \setminus X$, respectively, are connected and neither set is disjoint with the boundary of G. (In fact, in this case each of X and $V \setminus X$ shares one single connected interval with the boundary.)

Lemma 1.2. (Okamura–Seymour, 1981) *The cut condition holds for all cuts if and only if it holds for all essential cuts.*

Our algorithm works as follows. In a preprocessing step, we determine a specific solution for a certain auxiliary instance, which helps us to determine the solution, that is paths $p_1,\dots,p_k$, correctly. The core of the algorithm consists in a loop, where in each iteration exactly one of the paths $p_1,\dots,p_k$ is drawn. If the algorithm fails at any stage, it invokes a subroutine, which constructs an oversaturated cut from the past history of the failure. Clearly, this serves as a certificate for non-solvability. In particular, the correctness proof amounts to showing that each of the two subroutines, one for the preprocessing step and one for the core procedure, actually constructs an oversaturated cut when invoked after a failure. This is proved without using Theorem 1.1. Therefore, this in turn yields an alternative proof for Theorem 1.1.

For a linear-time realization, we will make use of a technique proposed by Gabow and Tarjan [3], which will enable our algorithm to perform certain sequences of *union-find* operations in linear time. Since our algorithm is linear in the worst case, it is optimal. In particular, it improves on some results for special cases [8, 10], and for even instances it is as fast as the algorithm in [13], which is in general applicable to weakly even instances with the additional restriction that the underlying graph forms a convex grid.

The paper is organized as follows. After some preliminaries in Section 2, we will introduce the new algorithm in Section 3 and prove its correctness (and Theorem 1.1, in particular). We conclude with the proof of the linear worst case bound in Section 4.

2. Preliminaries

Throughout this paper, $G=(V,E)$ is an undirected, connected, planar graph without loops or multiple edges (although the algorithm easily extends to that case). Graph G is given along with a fixed combinatorial embedding, that is, the adjacency list of each vertex is sorted according to a fixed geometric embedding in the plane, and there is one designated face, the *outer face*.

We assume that the set of nets is not empty, $N \neq \emptyset$, and that $x \in V$ is a fixed terminal, the *start terminal*. Moreover, we assume that all terminals have degree 1 and all other vertices have even degree. Obviously, a simple modification transforms any instance into a completely equivalent instance that fulfills this assumption. W.l.o.g., according to a counterclockwise ordering of all terminals starting with x, s_i precedes t_i for $i=1,\ldots,k$, and t_i precedes t_{i+1} for $i=1,\ldots,k-1$. The latter clearly means that in a sense all t-terminals are sorted in increasing order. The i-th terminal in counterclockwise ordering, starting with x, is denoted by x_i. Finally, we assume that the resulting graph after removing all terminals and their incident edges, $G \setminus \{s_1,\ldots,s_k,t_1,\ldots,t_k\}$, is biconnected, because otherwise we could solve each biconnected component separately.

We will also consider another, auxiliary, instance, which is denoted by $(G,N^{()})$. The set $N^{()}=\{\{s_1^{()},t_1^{()}\},\ldots,\{s_k^{()},t_k^{()}\}\}$ consists of the same terminals as N itself, and we have $t_i=t_i^{()}$ for all $i=1,\ldots,k$. But nonetheless, the pairing is different, namely according to a (unique) "parenthesis structure." That is, consider a $2k$-string with a left parenthesis at the i-th position, if x_i is an s-terminal, and a right parenthesis otherwise. Then two terminals are paired if and only if the corresponding parentheses match. This is equivalent to the restriction that for no pair $j,\ell=1,\ldots k$, the counterclockwise order around G is $s_j^{()}<s_\ell^{()}<t_j^{()}<t_\ell^{()}$. In particular, a solvable instance allows a solution where no two paths cross each other, if and only if it has a parenthesis structure. In fact, the solutions that we will produce for our auxiliary instances will be non-crossing in this sense.

At the end of this section, we now prove a fact that will not only be useful later on, but might also give an interesting insight into the interplay of input instances with their induced auxiliary instances in view of Theorem 1.1 and Lemma 1.2. For a cut $X \subset V$, let $dens_x(X)$ denote the density of X with respect to the auxiliary instance induced by start terminal x.

Lemma 2.1. *We have $dens(X) = \max_{i=1,\dots,2k} dens_{x_i}(X)$ for each essential cut X. In particular, the cut condition holds for $(G, N^{()})$ if and only if it holds for all auxiliary instances induced by the $2k$ possible start terminals.*

Lemma 2.1 follows immediately from a more general, "combinatorial," lemma, which we will state and prove next. Let S be a linear string of length $2k$, consisting of k emerald and k ruby pearls. A pairing $\mathcal{P}$ of the ruby pearls with the emerald pearls is *legal*, if each emerald pearl precedes its respective mate. The unique legal pairing $\mathcal{P}^{()}$ with *parenthesis structure* is defined by the restriction that we have $e_1 < e_2 < r_1 < r_2$ for no two pairs $(e_1, r_1), (e_2, r_2) \in \mathcal{P}^{()}$. Let R be a connected substring of S with $R \neq \emptyset$ and $S \setminus R \neq \emptyset$. Then for any ruby-emerald pairing $\mathcal{P}$, $n_{\mathcal{P}}(R)$ is the number of pairs with both pearls inside R. Substring R is called a *prefix* of S, if R starts with the first pearl of S.

Lemma 2.2. *Let S be a $2k$-string of emerald and ruby pearls that allows legal pairings, and let R be a substring of S with $R \neq \emptyset$ and $S \setminus R \neq \emptyset$. Then we have $n_{\mathcal{P}}(R) \leq n_{\mathcal{P}^{()}}(R)$ for each legal pairing $\mathcal{P}$. If R is a prefix of S, this holds with equality.*

Proof. Let $\mathcal{P} \neq \mathcal{P}^{()}$ be a legal pairing. Then there are two pairs $(e_1, r_1), (e_2, r_2) \in \mathcal{P}$ with $e_1 < e_2 < r_1 < r_2$. Let $\mathcal{P}'$ be the legal pairing arising from $\mathcal{P}$ by exchanging r_1 with r_2. Since $\mathcal{P}^{()}$ can be constructed from $\mathcal{P}$ by a sequence of such exchange steps, it suffices to show $n_{\mathcal{P}'}(R) \leq n_{\mathcal{P}}(R)$, and equality, if R is a prefix.

It is easy to see that $n_{\mathcal{P}'}(R) \neq n_{\mathcal{P}}(R)$ is possible only if R contains e_2 and r_1, but neither of e_1 and r_2. Since R is no prefix in this case, this already proves the latter claim of Lemma 2.2. Moreover, in this case we obviously have $n_{\mathcal{P}'}(R) = n_{\mathcal{P}}(R) - 2$, which proves the former claim as well. ∎

3. The Algorithm

In principle, our algorithm works as follows. First a preprocessing step is done, which tries to construct the specific solution for the auxiliary instance mentioned in the introduction. If this preprocessing step fails, we determine an oversaturated cut from the incomplete auxiliary solution constructed until the first appearance of a failure. Otherwise the auxiliary solution is used for constructing a solution for the input instance. If the latter procedure, the *core* procedure, fails, we again determine an oversaturated cut from the "past history" of this failure.

We are now going to handle preprocessing and core separately and in detail.

3.1. Preprocessing

In the preprocessing step, we construct a particular solution $(q_1,\dots,q_k)$ for instance $(G,N^{()})$, which is in a sense "extremal." In this procedure we will use a *right-first search* for each path, starting with the respective s-terminal. This means a depth-first search where in each step all possibilities of going forward are searched "from right to left." In other words, in any search step we select the counterclockwise next edge after the ingoing edge in the adjacency list of the current vertex.

In principle, we proceed in the same way as the well-known *stack-algorithm* for a similar problem, where *vertex-disjoint* paths are to be drawn [15]. It is easy to see that this procedure succeeds if and only if $(G,N^{()})$ is solvable.

Preprocessing: determine the auxiliary solution

```
FOR $i$:=1 TO $k$ DO
  1. let $q_i$ initially consist of the unique edge incident to $s_i^{()}$;
  2. $v$:= the unique vertex adjacent to $s_i^{()}$;
  3. WHILE $v$ is no terminal DO
     (a) let $e$ be the edge added to $q_i$ last;
     (b) let $\{v,w\}$ be the counterclockwise next free edge after $e$ in the adjacency
         list of $v$ that is not passed yet in this procedure;
     (c) add $\{v,w\}$ to $q_i$;
     (d) orient $\{v,w\}$ from $v$ to $w$;
     (e) $v$:=$w$;
  4. IF $v = t_i^{()}$ THEN indicate success ELSE indicate failure AND RETURN
     $\{q_1,\dots,q_{i-1}\}$.
RETURN $\{q_1,\dots,q_k\}$.
```

An instance (G,N) and its auxiliary graph are shown in Figure 1.

We notice that, due to the evenness condition, the procedure never gets stuck because of an unforeseen situation: While the current vertex is no terminal, there is always at least one edge left to proceed with. Therefore, the procedure does not crash down, but terminates properly.

Note that the paths $q_1,\dots,q_i$ do not cross themselves nor each other. However, nonetheless they may not be simple. This means the following. There may be two pairs of subsequent edges of path q_j, say, incident to the same vertex, but the elements of the pairs are ordered non-alternatingly around this vertex. In other words, when we pass q_j from $s_j^{()}$ to $t_j^{()}$, we never cross our own trace. Another fact worth to be mentioned is that all edges incident to a path q_j from the right belong to one of the paths $q_1,\dots,q_{j-1}$.

The following Lemma 3.1 generalizes the obvious fact that none of the paths $q_1,\dots,q_i$ contains a clockwise cycle, that is, a cycle that does not cross itself, and whose right side is just its interior. (Clearly, the right side and the interior of a non-crossing cycle are well defined.) Let $A(G,N,x)$ denote the directed graph formed by all paths $q_1,\dots,q_i$ ($i=k$, if no failure has occurred), where the direction of an

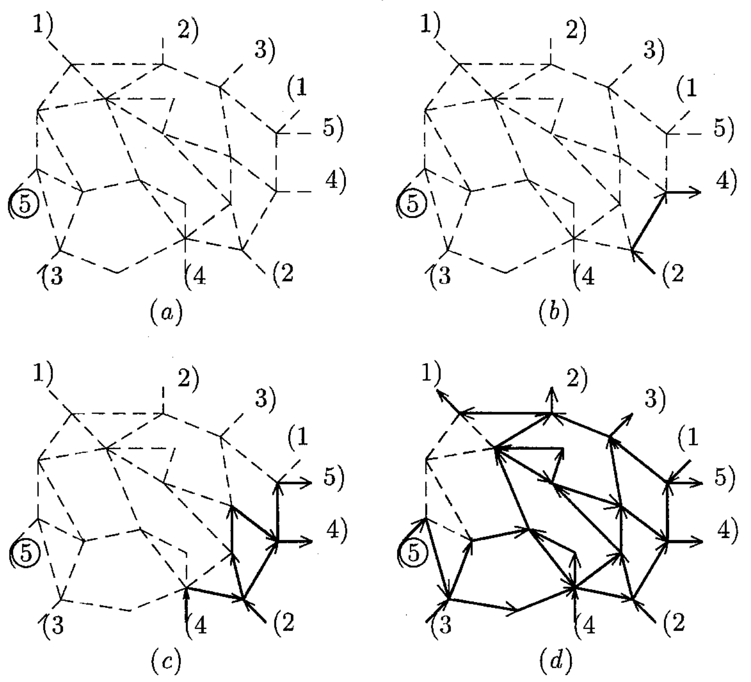

Fig. 1. An example, which satisfies all requirements stated in Section 2, to illustrate the preprocessing: (a) an even instance and the parenthesis structure induced by the encircled start terminal 5; (b) the first auxiliary path; (c) first and second paths; (d) the whole auxiliary graph $A(G,N,x)$ determined by the preprocessing. Note that the auxiliary path from 5 to 1 contains two counterclockwise cycles

edge is "inherited" from the orientation of its respective path from the s-terminal to the t-terminal.

Lemma 3.1. *There is no clockwise cycle in $A(G,N,x)$.*

Proof. Let $\mathscr{C}$ be a non-crossing cycle of $A(G,N,x)$, let e be the first edge of $\mathscr{C}$ added to one of the paths $q_1,\dots,q_i$ in the preprocessing step, and let e' be the immediate predecessor of e with respect to that path. Then e' points to $\mathscr{C}$ from outside, which immediately implies that e is given the orientation conforming to a *counter*clockwise orientation of $\mathscr{C}$. ∎

Now we are going to show how to construct an oversaturated cut if the preprocessing fails, that is, if $i<k$. For this, we make use of a technique that is folklore [6] and is extensively investigated by Itai and Shiloach and by Hassin in connection with the maximum flow problem in (s,t)-planar networks [7, 4].

Let $G_0=(V_0,E_0)$ denote the graph G after removing all terminals and all edges incident to terminals. Recall from Section 2 that G_0 is biconnected. Next let $G^d=(V^d,E^d)$ be the graph that arises from the dual graph of G_0 as follows: The dual vertex corresponding to the outer face is split into two vertices, r and u, and each edge on the boundary of G_0 is incident in G^d either to r or to u. Namely, the edges made incident to r are just the edges counterclockwise after the (unique) vertex of G_0 adjacent to $s_i^{()}$ and before the (unique) vertex adjacent to $t_i^{()}$. And analogously, the edges made incident to u are just the edges on the other interval of the boundary of G_0 between $s_i^{()}$ and $t_i^{()}$.

The graph $G^* = (V^d, E^*)$ is a *directed* subgraph of G^d, which means that each edge in E^* is given an orientation in addition. Let q'_j denote the path q_j restricted to G_0, that is, without the first and the last edge. Then E^* consists of all dual edges crossing one of the paths $q'_1, \dots, q'_{i-1}$, and each edge is given the orientation such that it crosses its respective path "from right to left." Let $m \in \{1, \dots, i-1\}$ be maximum such that $t_m^{()} < s_i^{()}$. If no such m exists, we set $m = 0$.

Now the crucial insight is the following.

Lemma 3.2. *There is a vertex $v \in V^d$ such that v can be reached from u by a directed path p_1 in G^* which crosses only the paths $q'_1, \dots, q'_m$, and from r, by a directed path p_2 in G^* which crosses only $q'_{m+1}, \dots, q'_{i-1}$. In particular, $v = u$ if $m = 0$, and $v = r$ if $m = i-1$.*

Before proving Lemma 3.2, we first give its consequences.

Corollary 3.3. *There is an oversaturated cut in (G, N).*

Proof of Corollary 3.3. Let $v \in V^d$ be a vertex as guaranteed by Lemma 3.2, and let p_1 and p_2 be a dual (u, v)-path and a dual (r, v)-path as described in Lemma 3.2. In particular, v, p_1 and p_2 can be chosen so that p_1 and p_2 are vertex-disjoint except for v. Clearly, the concatenation $p_1 + p_2$ separates $s_i^{()}$ from $t_i^{()}$. Let $X \subseteq V$ denote the set of all vertices on the $s_i^{()}$-side of $p_1 + p_2$. Then each edge *entering* X belongs to one of the paths $q'_1, \dots, q'_m$, and each edge *leaving* X belongs to $q'_{m+1}, \dots, q'_{i-1}$. Therefore, all edges connecting X with $V \setminus X$ belong to pairwise different paths. As $q_1, \dots, q_{i-1}$ are correct, each net $(s_j^{()}, t_j^{()})$ whose path q_j occupies an edge between X and $V \setminus X$ contributes to the density of X. Hence, these nets already saturate the cut X in $(G, N^{()})$, and the net $(s_i^{()}, t_i^{()})$ even *over*saturates X in $(G, N^{()})$. Now Lemma 2.1 implies that X is oversaturated in (G, N) as well, since X is obviously essential. ∎

Lemma 3.2 and Corollary 3.3 immediately suggest the following algorithm.

After failure of preprocessing: determine an oversaturated cut

```
1. construct $G_0 = (V_0, E_0)$;
2. construct the two-source dual graph $G^d = (V^d, E^d)$ from $G_0$;
3. $E^* := E^d$;
4. remove all edges from $E^*$ that cross none of the paths $q'_1, \dots, q'_{i-1}$;
5. orient each edge in $E^*$ such that it crosses its path from right to left;
6. $G^* := (V^d, E^*)$;
7. let $T$ be a tree in $G^*$ which is rooted at $r$ and spans all dual vertices that can
   be reached from $r$ in $G^*$;
```

```
 8. find a directed path $p_1$ in $G$ from $u$ to some vertex $v$ in $T$ so that $p_1$ is
    internally vertex-disjoint from $T$;
 9. let $p_2$ be the path from $r$ to $v$ in $T$;     (* $p_1+p_2$ separates $s_i^{()}$ from $t_i^{()}$. *)
10. RETURN the set of all vertices on the $s_i^{()}$-side of $p_1+p_2$.
```

Each step of this algorithm is linear. (For step 7, simply take a depth-first search, and for step 8, a depth-first search which terminates immediately when a vertex of T is seen.) And correctness is immediate from Lemma 3.2 and the proof of Corollary 3.3. Hence, for the preprocessing, it remains to prove Lemma 3.2. We divide this proof into several lemmas.

Lemma 3.4. *If a vertex $v \in V^d \setminus \{r,u\}$ is left in G^* by an edge crossing path q'_j, say, v is entered in G^* by an edge crossing a path q'_ℓ with $s_j^{()} < s_\ell^{()} < t_\ell^{()} < t_j^{()}$.*

Proof. Let $v \in V^d \setminus \{r,u\}$ be left by an edge crossing path q'_j. This means that q'_j runs a bit clockwise around the face corresponding to v. By Lemma 3.1, there is a primal edge of G_0 incident to v which either belongs to no path at all, or belongs to some path that runs a bit *counter*clockwise along the face of v.

First note that the latter is true in any case, that is, one of these edges belongs to some path q'_ℓ running counterclockwise along the face of v. In fact, otherwise the layout of q_j had run into a clockwise cycle around v. As the primal edges of q'_ℓ on the face of v correspond to dual edges of G^* that enter v, it remains to show that $s_j^{()} < s_\ell^{()} < t_\ell^{()} < t_j^{()}$. To see this, next note that $\ell \neq j$, because otherwise, as is easy to see, q'_j had to cross itself to run twice in different directions along the same face, which is obviously not the case.

In summary, a path q'_ℓ with $\ell \neq j$ shares an edge with the face of v and orients this edge counterclockwise around v. As the auxiliary paths do not cross each other, this may happen only if $(s_\ell^{()}, t_\ell^{()})$ is properly nested in $(s_j^{()}, t_j^{()})$. This proves Lemma 3.4. ∎

Lemma 3.5. *All dual vertices incident to edges crossing paths $q'_1, \ldots, q'_m$ can be reached via directed paths from u in G^*, and all dual vertices incident to edges crossing paths $q'_{m+1}, \ldots, q'_{i-1}$ can be reached from r in G^*.*

Proof. First let $(s_j^{()}, t_j^{()})$ be an innermost nested net in the parenthesis structure. Then q'_j runs counterclockwise along the boundary of G_0, which means that, for each edge on q'_j, the dual vertex immediately on the right side is either u or r. In fact, it is u if and only if $j \leq m$, since the vertices adjacent to the terminals $s_1^{()}, \ldots, s_m^{()}, t_1^{()}, \ldots, t_m^{()}$, respectively, all belong to the part of the boundary of G_0 connected with u, and the vertices adjacent to the terminals $s_{m+1}^{()}, \ldots, s_{i-1}^{()}, t_{m+1}^{()}, \ldots, t_{i-1}^{()}$ all belong to the part connected with r. Moreover, each vertex on the *left* side of an edge on q'_j is entered in G^* by an edge crossing q'_j, which has tail u if $j \leq m$, or r, otherwise. This proves the claim for innermost nets. Using Lemma 3.4, a straightforward induction shows the claim for the other nets as well. ∎

Lemma 3.6. *In G^*, r and u are weakly connected, that is, there is a path from r to u, which may contain forward and backward edges.*

Proof. Note that there is a cut in G_0 which separates $s_i^{()}$ from $t_i^{()}$, and all of whose (primal) edges belong to the paths $q'_1,\dots,q'_j$, because otherwise the i-th iteration had succeeded. The dual edges corresponding to this cut form a path as desired. ∎

Proof of Lemma 3.2. Let p be an (r,u)-path as guaranteed by Lemma 3.6. If all vertices on p can be reached from u via directed paths from u, we define p_1 to be a directed path from u to r and $p_2 = \{r\}$, and we are done. Otherwise, let v be the vertex closest to r on p that can be reached from u (possibly $v = u$). Then p_1 is a directed path from u to v. Let w be the next vertex after v on p towards r. By the specific choice of v, vertex w cannot be reached from u. Hence, Lemma 3.5 implies that w can be reached from r. However, since v can be reached from u and w cannot, we know that the edge $\{v,w\}$ is oriented towards v. This means that v is reachable from r as well, and we may construct a path p_2 from r to v via w. ∎

3.2. Core Procedure

We now assume that the preprocessing step has succeeded. Next we determine the paths $p_1,\dots,p_k$ for our original instance (G,N), in this order in fact. (Recall that t_i precedes t_{i+1} for all i counterclockwise after the start terminal.) For each path p_i we use a directed right-first search in $A(G,N,x)$, starting with s_i. (Remember the definition of $A(G,N,x)$ in Section 3.1.) "Directed" means that we may pass an edge only according to its orientation, from tail to head. The adjacency list of v is a cyclic list representing *all* edges incident to v in $A(G,N,x)$, leaving v or entering v. Now we are able to formulate the core of our algorithm formally.

Core procedure: try to determine a solution for (G,N)

```
FOR $i := 1$ TO $k$ DO
  1. let $p_i$ initially consist of the unique edge leaving $s_i$ in $A(G,N,x)$;
  2. $v :=$ the head of this edge;
  3. WHILE $v$ is no terminal DO
     (a) let $e$ be the edge added last to $p_i$;
     (b) let $(v,w)$ be the counterclockwise next free edge after $e$ in the adjacency
         list of $v$ that leaves $v$ and is not passed yet in this procedure;
     (c) add $(v,w)$ to $p_i$;
     (d) $v := w$;
  4. IF $v = t_i$ THEN indicate success ELSE indicate a failure and break loop;
RETURN the paths $p_1,\dots,p_i$ constructed so far.
```

In Figure 2, the auxiliary graph of Figure 1 is shown, along with the situation after the first four iterations of the core procedure, respectively. Again we notice that the procedure will in any case be terminated correctly by the RETURN statement.

Note that all edges of $A(G,N,x)$ leaving path p_j, say, on the right side belong to one of the paths $p_1,\dots,p_{j-1}$. Before proceeding with the case of a failure, we shall cite an insightful result from the conference version of this paper, where that result has been used for proving correctness of the algorithm in a completely different manner.

Lemma 3.7. [18] *Consider the residual instance that arises from (G,N) by removing all nets $\{s_1,t_1\},\dots,\{s_{j-1},t_{j-1}\}$ and all edges on the paths $p_1,\dots,p_{j-1}$. Then the restriction of $A(G,N,x)$ to this residual graph equals the auxiliary graph in this residual graph with respect to an appropriate start terminal.*

In particular, it would suffice to show that the very first path p_1 is correct, i.e. connects the correct terminals and leaves a solvable residual instance: Any other path is just the first path in a residual instance.

Now we are going to construct an oversaturated cut from $p_1,\dots,p_i$, if the i-th iteration of the core procedure has failed. Again, the first $i-1$ paths are correct, that is, end with $t_1,\dots,t_{i-1}$, respectively. On the other hand, the i-th path ends with a terminal $x_r \neq t_i$.

Let H denote the graph formed by all vertices and edges on the paths $p_1,\dots,p_i$. Then H is a directed graph, and the orientation of an edge conforms to the orientation of the path p_j it belongs to.

We construct the oversaturated cut by a reverse directed *left*-first search in H, which starts with x_r and terminates when another terminal x_ℓ, say, is seen. "Reverse" means that an edge is passed only from head to tail. This defines a path p in H going from x_ℓ to x_r. The oversaturated cut consists of all vertices on the right side of p. (Lemma 3.9 will show that the right side of p is well defined.)

This procedure terminates properly, too.

After failure of core procedure: determine an oversaturated cut

```
1. let $p$ initially consist of the unique edge incident to $x_r$;
2. $v :=$ the unique vertex adjacent to $x_r$;
3. WHILE $v$ is no terminal DO
   (a) let $e$ be the edge added last to $p$;
   (b) let $(u,v)$ be the clockwise first edge after $e$ in the adjacency list of $v$ that enters
       $v$ and is not passed yet in this procedure;
   (c) add $(u,v)$ to $p$;
   (d) $v := u$;
4. RETURN the vertices on the right side of $p$.
```

This completes the description of the algorithm. It remains to show that the last procedure (henceforth called the *failure-handler*) actually finds an oversaturated cut when invoked after a failure of the core procedure. We will prove the following, even stronger theorem. We call an iteration of the core procedure "good," if no error has been detected in this iteration or in any previous one.

Theorem 3.8. *Consider an iteration of the core procedure such that all previous iterations were good. If the failure-handler is deliberately invoked after this iteration, the failure-handler returns a saturated or oversaturated cut. Moreover, if this iteration is bad, the cut returned is oversaturated.*

We divide the proof of Theorem 3.8 into several lemmas. Suppose that the first $i-1$ iterations of the core procedure were good, and that we invoked the failure-handler after the i-th iteration, good or bad. Let p be the path produced by the

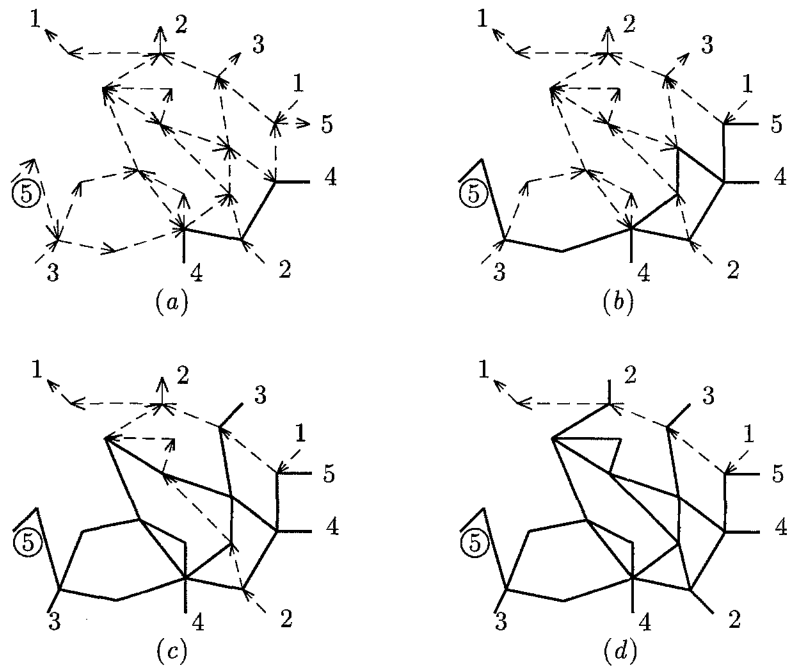

Fig. 2. Edge-disjoint paths determined in the auxiliary graph of Figure 1: (a) the first "final" path; (b) the first and the second "final" paths; (c) the first three "final" paths; (d) the first four "final" paths determined by the algorithm. The fifth path will use up all edges of $A(G,N,x)$

failure-handler when being invoked after the i-th iteration. First of all, we have to show that the right side of p is well defined, because this is the cut returned by the failure-handler.

Lemma 3.9. *Path p does not cross itself. In particular, the right side and the left side of p are well defined.*

Proof. Suppose p *does* cross itself at vertex v. This means that there are four edges of $A(G,N,x)$, e_1, e_2, e_3, e_4, incident to v such that e_1 (resp., e_2) is the immediate predecessor of e_3 (e_4) on p and the counterclockwise order of these four edges around v is $e_1 < e_2 < e_3 < e_4$. By construction of p, e_1 and e_3 must precede e_2 and e_4 on p, since otherwise, by our right-first strategy, e_3 were the immediate successor of e_2 rather than e_4.

Therefore, path p decomposes into three subpaths, p^1, p^2, and p^3, where p^1 is the subpath from the s-terminal up to e_1, p^2 is the (cyclic) subpath from e_3 to e_2, and p^3 is the subpath from e_4 to the t-terminal. W.l.o.g. assume that no internal vertex of p^1 is a crossing of p. In particular, p^2 does not cross p^1.

Clearly, at this point of our argumentation we cannot exclude the case that p^2 crosses itself. But p^2 may be reduced to a non-crossing cycle which still contains e_3 and e_2 as consecutive edges and still does not cross p^1. The latter implies that e_1 points to this reduced cycle from outside. On the other hand, e_1 points to this reduced cycle from the left side between e_2 and e_3. Hence, the reduced cycle is a clockwise cycle of $A(G,N,x)$, which contradicts Lemma 3.1. ∎

Let X denote the right side of p and let $\delta(X)$ be the set of all edges in $A(G,N,x)$ connecting a vertex in X with a vertex in $V\backslash X$. Moreover, let $(X,V\backslash X)$ and $(V\setminus X,X)$ denote the sets of all edges in $\delta(X)$ that leave and enter X, respectively. An edge is called *occupied*, if it belongs to one of the paths $p_1,\ldots,p_i$, and *free*, otherwise.

Lemma 3.10. *Each edge in $(X,V\setminus X)\subseteq A(G,N,x)$ is free, and each edge in $(V\backslash X,X)\subseteq A(G,N,x)$ is occupied.*

Proof. The former claim follows immediately from our reverse left-first strategy, applied by the failure-handler to construct p. To see the latter claim by contradiction, assume that $e\in(V\backslash X,X)$ is free. Let v be the tail of e. Then v belongs to p. Let e_1 be the counterclockwise next edge of p after e in the adjacency list of v. As e belongs to the right side of p, e_1 leaves v. Let $p_j, j\leq i$, be the path occupying e_1 and let e_2 be the immediate predecessor of e_1 on p_j. Because of the former claim of Lemma 3.10, e_2 cannot be located counterclockwise after e and before e_1. Consequently, e precedes e_1 counterclockwise after e_2 in the adjacency list of v. But this contradicts the right-first strategy applied to construct p_j, since e had become the immediate successor of e_2 on p_j rather than e_1. ∎

Next we want to prove that all edges of G between X and $V\backslash X$ actually belong to $\delta(X)$. For this aim, we first give two consequences of the right-first strategy in the preprocessing and in the core procedure (Lemmas 3.11 and 3.12). We need some terminology. Let $v\in V$ and let $e,e'\in E$ incident to v. Then $[e,e']_v$ denotes the interval of the adjacency list of v counterclockwise after e and before e' (including both e and e'). Furthermore, $\mathscr{D}_v[e,e']$ denotes the difference of the cardinalities of two certain subsets of $[e,e']_v\cap A(G,N,x)$. More precisely, the number of edges entering v *minus* the number of edges leaving v. Whenever we count only occupied or free edges, we instead write $\mathscr{D}_v^{occ}[e,e']$ and $\mathscr{D}_v^{free}[e,e']$, respectively.

Lemma 3.11. *Let $v\in V$ and let $e,e'\in E$ be incident to v. If e' does not belong to $A(G,N,x)$, the number of edges in $[e,e']_v\cap A(G,N,x)$ entering v is at most the number of edges in $[e,e']_v\cap A(G,N,x)$ leaving v. For short: $\mathscr{D}_v[e,e']\leq 0$.*

Proof. Suppose we have $\mathscr{D}_v[e,e']>0$. A simple pigeon hole argument shows that at least one of the edges in $[e,e']_v$ that enters v is immediately followed on its auxiliary path by an edge outside $[e,e']_v$. Therefore, e' is incident to this auxiliary path from the right side. If e' does not belong to $A(G,N,x)$, this is clearly a contradiction to our right-first strategy applied to this auxiliary path in the preprocessing. ∎

Lemma 3.12. *Let $v\in V$ and let $e,e'\in E$ be incident to v. If e' is a free edge of $A(G,N,x)$ leaving v, the number of occupied edges in $[e,e']_v\cap A(G,N,x)$ entering v is at most the number of occupied edges in $[e,e']_v\cap A(G,N,x)$ leaving v. For short: $\mathscr{D}_v^{occ}[e,e']\leq 0$.*

Proof. Completely analogously to the proof of Lemma 3.11, we may arrive at a contradiction to our right-first strategy in the core procedure. ∎

Now we are able to prove the following Lemma 3.13.

Lemma 3.13. *Each edge of E connecting a vertex in X with a vertex in $V\setminus X$ belongs to $\delta(X)\subseteq A(G,N,x)$.*

Proof. Assume for a contradiction that there is an edge e between X and $V\setminus X$ that does not belong to $A(G,N,x)$. Let v be the endvertex of e in $V\setminus X$. Then v belongs to p. Let e_1 be the counterclockwise next edge of p after e in the adjacency list of v.

There is an edge $e_2\neq e_1$ such that $\mathscr{D}_v^{occ}[e_2,e_1]=0$. (At least the counterclockwise first edge after e_1 will do.) In particular, let e_2 be the clockwise first edge after e_1 with $\mathscr{D}_v^{occ}[e_2,e_1]=0$. Then e_2 enters v and is occupied. We will show $\mathscr{D}_v^{occ}[e_2,e]>0$ and $\mathscr{D}_v^{free}[e_2,e]\geq 0$. Clearly, this proves the claim in connection with Lemma 3.11.

The former claim of Lemma 3.10 implies $e_2\notin[e,e_1]_v$ or, equivalently, $e\in[e_2,e_1]_v$. Moreover, that particular claim also implies $\mathscr{D}_v^{occ}[e,e_1]<0$, since no occupied edge in $[e,e_1]_v$ enters v and at least e_1 leaves v. Because of $\mathscr{D}_v^{occ}[e_2,e_1]=0$, we therefore have $\mathscr{D}_v^{occ}[e_2,e]>0$.

It remains to show $\mathscr{D}_v^{free}[e_2,e]\geq 0$. For this aim, we will show that no free edge in $[e_2,e]_v\cap A(G,N,x)$ leaves v. To see this, let $e_3\in[e_2,e]_v$, $e_3\neq e_2$. By the specific choice of e_2, we have $\mathscr{D}_v^{occ}[e_3,e_1]<0$. However, if e_3 is a free edge, this would imply $\mathscr{D}_v^{occ}[e_2,e_3]>0$. And if e_3 in addition belongs to $A(G,N,x)$ and leaves v, this contradicts Lemma 3.12. ∎

Now we are in a position to prove Theorem 3.8 itself.

Proof. Let $cap_1(X)$ and $cap_2(X)$ denote the number of occupied and free edges of $\delta(X)$, respectively. By Lemma 3.13, $cap(X)=cap_1(X)+cap_2(X)$. Moreover, let $dens_1(X)$ denote the number of pairs $\{s_j,t_j\}, j=1,\dots,i-1$, separated by X. Analogously, let $dens_2(X)$ be the number of pairs $\{s_j,t_j\}$, $j\geq i$, with $s_j\in X$ and $t_j\notin X$, and let $dens_3(X)$ be the number of pairs $\{s_j,t_j\}$, $j\geq i$, with $t_j\in X$ and $s_j\notin X$. Clearly, we have $dens(X)=dens_1(X)+dens_2(X)+dens_3(X)$.

As all occupied edges in $\delta(X)$ are oriented from $V\setminus X$ to X (Lemma 3.10), they all must belong to pairwise different paths $p_1,\dots,p_{i-1}$, respectively connecting an s-terminal in $V\setminus X$ with a t-terminal in X. Since the first $i-1$ iterations have succeeded, this implies $cap_1(X)\leq dens_1(X)$. Therefore, it suffices to show $cap_2(X)\leq dens_2(X)+dens_3(X)$, and that this inequality is strict if a failure was detected.

Obviously, the free edges decompose into directed cycles and directed paths from free s-terminals to free t-terminals. As all free edges between X and $V\setminus X$ are oriented from X to $V\setminus X$ (Lemma 3.10), they all belong to pairwise different paths in this decomposition. Thus, the number of free edges in $\delta(X)$ equals the number of free s-terminals in X *minus* the number of free t-terminals in X. Because of Lemma 3.13, this already proves $cap_2(X)\leq dens_2(X)$. In case of a failure, note that $t_i\in X$ and $s_i\notin X$, which means $dens_3(X)>0$. ∎

This completes the correctness proof. We notice that if the failure-handler is invoked after the first iteration of the core procedure, we have $p=p_1$. Therefore, if (G,N) is solvable, the path p_1 runs along a saturated cut. This has an interesting consequence, which might give a better insight into the nature of the evenness condition. (In fact, the following corollary 3.14 does not hold for non-even instances in general.)

Corollary 3.14. *The right side of p_1 is saturated. In particular, the set of all paths from s_1 to t_1 that belong to solutions for (G,N) has a unique "rightmost" element (namely p_1).*

As the start terminal, x, is arbitrary, each net $\{s_j,t_j\}$ has a unique rightmost solution path, if the interval of the boundary of G counterclockwise after s_i and before t_i contains at most one terminal of any other net. Moreover, it is easy to see that this is true for any other net $\{s_j,t_j\}$, too, if we restrict attention to solutions where no two paths cross more than once. (Obviously, there is always such a solution.) To see this, simply consider the residual instance after removing all nets with both terminals on the right side of $\{s_j,t_j\}$ and removing all edges on the rightmost paths for these nets.

4. A Linear-Time Realization

It is not hard to see that the preprocessing step and the failure-handler need only linear time. Since in each iteration of the WHILE-loop in the core procedure, one edge is added to a path p_i and no edge is added twice, the total number of iterations of the WHILE-loop is linear. Therefore, realizing the core procedure in linear time amounts to designing data structures for vertices and edges such that any statement of the WHILE-loop can be executed in (amortized) constant time. Recall that we store for each vertex v its adjacency list, which is a cyclic list of *all* edges incident to v in $A(G,N,x)$, leaving v or entering v.

Obviously, each statement requires constant time, except for the problem to find the next edge to proceed with in the core procedure and in the failure-handler. For this particular task, we need some terminology. At any stage of the algorithm, the *signpost* of $e \in E$ is the next edge after e that leaves the head of e. Conversely, the set of all edges with signpost $e \in E$ is called the *client set* of e. During the algorithm, we will always maintain the client set of each edge. Any edge "knows" the client set it belongs to, and any client set "knows" its signpost.

Let $v \in V$ be no terminal. Then the adjacency list of v can be uniquely decomposed into maximal connected components such that all edges in such a connected component leave v or all edges enter v. Let $e \in E$ leave v. The initial client set of e is either empty or just the maximal connected component of edges entering v that appears counterclockwise immediately before e in the adjacency list of v in reverse clockwise ordering. The former is true if and only if the clockwise next edge after e in the adjacency list of v leaves v, too.

Whenever an edge e is occupied whose client set is non-empty, we have to update this set. If the next edge after e, say e', leaves v, e' must still be a free edge, because anyway, e has to be considered before e' for going forward from v. Then the signpost of the client set of e is simply set to e'.

In case of e' entering v, there are two different cases. If there is more than one client set consisting of edges entering v, the signpost of the client set of e must from now on be the signpost of the client set to which e' belongs. In other words, we have to unite these two client sets. On the other hand, if there is only one such client set, there is no more free edge leaving v at all. Since there is also no more free edge entering v in this case, we need not consider v any longer.

Therefore, maintaining and using client sets and their signposts amounts to performing, for each non-terminal vertex v, a sequence of *union-find* operations on the set of all edges entering v.

In general, such a sequence cannot be performed in linear time in the worst case [16]. However, Gabow and Tarjan have shown that this is possible for a certain special case [3]. We will show that, with a little trick, this special case covers our problem. The special case considered by Gabow and Tarjan is the following. Let T be a rooted tree. Initially, the underlying set, on which the union-find operations shall be performed, is partitioned into singletons. Any singleton is represented by exactly one vertex of T. Then two partition sets S_1 and S_2 are allowed to be united if and only if there are two elements $s_1 \in S_1$ and $s_2 \in S_2$ such that s_1 is the immediate parent of s_2 in T or vice versa. The name of the union is the name of the set closer to the root of T.

In our application, tree T will simply be a linear list L_v of all edges entering v. This list is sorted according to the plane embedding. In other words, to obtain such a list L_v the cyclic adjacency list of v is broken at some point, and all edges leaving v are removed from it. We select the point where to break the adjacency list such that none of the initial client sets is broken. As a consequence, constructing the initial client sets can be done by the technique of Gabow and Tarjan, applied to "tree" L_v. As a further consequence, for any non-terminal vertex v there is at most one stage where two client sets of L_v that are *not* neighbored in L_v are to be united, namely the first and the last client set of L_v. All other unions can be done using the technique of Gabow and Tarjan.

In order to cope with the single union-operation where this technique fails, we simply maintain that the signpost of the last client set is always equal to the signpost of the first set, once we have arrived at a stage where these two sets are to be united. During the algorithm, we maintain the information, which client set is the first one. Whenever the signpost of the first set is changed (or even the first set itself by a union), the signpost of the last client set is updated accordingly. As a result, we obtain the following theorem.

Theorem 4.1. *The algorithm can be realized such that it requires linear time in the worst case.*

Acknowledgement. We would like to thank *Gabriele Neyer* and *Heiner John* for pointing out, independently of each other, an error in an earlier version of the failure handler for the preprocessing step. This has led to a completely new procedure and correctness proof for the case of a failure in the preprocessing.

References

[1] M. Becker, and K. Mehlhorn: Algorithms for routing in planar graphs, *Acta Inform.*, **23** (1986), 163–176.

[2] G. N. Frederickson: Fast algorithms for shortest paths in planar graphs with applications, *SIAM J. Comput.*, **16** (1987), 1004–1022.

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

[4] R. Hassin: Maximum flows in (s,t) planar networks, *Inform. Process. Lett.*, **13** (1981), 107.

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

[6] T. C. Hu: *Integer programming and network flows*, Addison-Wesley, Reading, MA, 1969.

[7] A. Itai, and Y. Shiloach: Maximum flows in planar networks, *SIAM J. Comput.*, **8** (1979), 135–150.

[8] M. Kaufmann: A linear time algorithm for routing in a convex grid, *IEEE Trans. Comp.-Aided Design*, CAD-**9**, 180–184, 1990.

[9] M. Kaufmann, and G. Klär: A faster algorithm for edge-disjoint paths in planar graphs, In W. L. Hsu and R. C. T. Lee, editors, *ISA'91 Algorithms, Second International Symposium on Algorithms*, pages 336–348. Springer-Verlag, Lecture Notes in Computer Science, vol. 557, 1991.

[10] M. Kaufmann, and K. Mehlhorn: Generalized switchbox routing, *J. Algorithms*, **7** (1985), 510–531.

[11] P. Klein, S. Rao, M. Rauch, and S. Subramanian: Faster shortest-path algorithms for planar graphs, Proceedings of *STOC '94*.

[12] K. Matsumoto, T. Nishizeki, and N. Saito: An efficient algorithm for finding multicommodity flows in planar networks. *SIAM J. Comput.*, **14** 289–302, 1985.

[13] T. Nishizeki, N. Saito, and K. Suzuki: A linear time routing algorithm for convex grids, *IEEE Trans. Comp.-Aided Design*, CAD-4:68–76, 1985.

[14] H. Okamura, and P. D. Seymour: Multicommodity flows in planar graphs. *J. Combin. Theory Ser. B*, **31** (1981), 75–81.

[15] H. Suzuki, T. Akama, and T. Nishizeki: Finding Steiner forests in planar graphs. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'90*, pages 444–453, 1990.

[16] R. E. Tarjan: A class of algorithms which require non-linear time to maintain disjoint sets. *J. Comp. System Sciences*, **18** (1979), 110–127.

[17] K. Weihe: Multicommodity flows in even, planar networks, In K. W. Ng, P. Raghavan, N. V. Balasubramanian, and F. Y. L. Chin, editors, *Algorithms and Computation, 4th International Symposium, ISAAC'93*, pages 333–342. Springer-Verlag, Lecture Notes in Computer Science, vol. 762, 1993.

[18] D. Wagner, and K. Weihe: A linear time algorithm for edge-disjoint paths in planar graphs, In T. Lengauer, editor, *First European Symposium on Algorithms, ESA'93*, pages 384–395. Springer-Verlag, Lecture Notes in Computer Science, vol. 726, 1993.

Dorothea Wagner

Universität Konstanz,
Informatik,
78434 Konstanz, Germany,
dorothea.wagner@uni-konstanz.de

Karsten Weihe

Universität Konstanz,
Informatik,
78434 Konstanz, Germany,
karsten.weihe@uni-konstanz.de

Typeset by TYPOTEX Ltd, Budapest
PRINTED IN HUNGARY
Akadémiai Kiadó és Nyomda Vállalat, Budapest