Simple Algorithms for Steiner Trees and Paths Packing Problems in Planar Graphs

Dorothea Wagner
Fachbereich Mathematik, Technische Universität Berlin,
Straße des 17. Juni 136,
10623 Berlin, Germany.
email: wagner@math.tu-berlin.de

In this paper we give a short survey on efficient algorithms for Steiner trees and paths packing problems in planar graphs. We particularly concentrate on recent results. The first result is a linear-time algorithm for packing vertex-disjoint trees resp. paths within a planar graph, where the vertices to be connected lie on at most two different faces [26, 32]. This algorithm is based on a linear-time algorithm for the vertex-disjoint Menger problem in planar graphs [27]. The second result is a linear-time algorithm for packing edge-disjoint paths within a planar graph, where the vertices to be connected all lie on the boundary of the same face [34]. All algorithms discussed in detail in this paper admit a short and simple description, and are easy to implement.

1. Introduction

In this paper we consider the following type of problems.

The Steiner Trees Packing Problem

Given: A planar graph $G = (V, E)$, $|V| = n$, and sets $N_1, \ldots, N_k \subseteq V$. In this context we call the N_i *nets* and the elements of N_i *terminals*.

Problem: Find, for each $i = 1, \ldots, k$, a *Steiner tree* T_i for N_i (i. e. a tree in G connecting all terminals in N_i) such that $T_1, \ldots, T_k$ are pairwise vertex-disjoint/edge-disjoint.

Steiner trees packing problems have many practical applications, e. g. in the design process of integrated circuits, the design of reliable communication networks or the control of traffic networks. In all these applications, planarity of the underlying graph is a natural restriction. Most Steiner trees packing problems are $\mathcal{NP}$-hard. In this paper we concentrate on restrictions of this

problem which admit efficient solution algorithms. Natural restrictions which are still of practical relevance concern e. g. the position of the terminals or the number of terminals of a net.

The algorithms described here are "really efficient". This means that the polynomial worst-case running time is not only proved theoretically. In fact, the algorithms discussed in detail in this paper admit a short and simple description. They have even linear running time where the order constant is obviously small. Moreover, they are easy to implement, i. e., can be of practical relevance.

2. The Vertex-Disjoint Case

The general vertex-disjoint Steiner trees packing problem in planar graphs is $\mathcal{NP}$-complete [16]. Even when restricted to paths, i. e., when all nets have exactly two terminals, the problem remains $\mathcal{NP}$-complete. However, Robertson & Seymour [29] showed that the vertex-disjoint paths packing problem is solvable in polynomial time (also for non-planar graphs) if k, the number of paths, is fixed. Recently, even a linear-time algorithm has been introduced for planar graphs [23]. But the order constant of these algorithms depends heavily on the number of paths.

The first polynomial time algorithm for the vertex-disjoint Steiner trees packing problem in planar graphs where the terminals lie only on one or two face boundaries was given in [28]. But this algorithm is far off from being implementable. In [15] and in [32] an easy $\mathcal{O}(n)$ algorithm is described for the case that all terminals lie on the boundary of the same face. The proceeding of this algorithm may be seen as a "key idea" for deriving simple linear-time algorithms for similar problems. In fact, the new algorithm discussed in Section 2.3 for the vertex-disjoint Menger problem is in some sense a "generalization" of this algorithm. Also the algorithm for parenthesis problems given in Section 3.1 is a slightly modified version of the algorithm. Moreover, a similar method can be applied to solve a special case of the edge-disjoint one-face paths packing problem discussed in 3.2 in linear time.

2.1. The One-Face Steiner Trees Packing Problem

In this section we describe the algorithm from [15] and [32] solving the following problem.

Vertex-Disjoint One-Face Steiner Trees Packing Problem

Given: A planar graph $G = (V, E)$, $|V| = n$, and pairwise vertex-disjoint sets $N_1, \ldots, N_k \subseteq V$. The graph G is embedded in the plane such that all terminals lie on the boundary of the outer face of G.

Problem: Find, for each $i = 1, \ldots, k$, a Steiner tree T_i for N_i such that $T_1, \ldots, T_k$ are pairwise vertex-disjoint.

The algorithm runs in two phases. First the *topological solvability* is tested, and then the layout of the nets is determined, assuming enough capacity is

available. Informally, to guarantee topological solvability the nets must have a *nested* structure, i. e., are not allowed to cross. More precisely, a *topological solution* is a collection of Steiner trees for the nets that can be drawn disjointly in the plane outside the outer face. So a topological solution is not necessarily contained in G. Obviously, the existence of a topological solution depends only on the position of the terminals on the outer face boundary of G. Topological solvability can be decided by a simple algorithm, the *Stack Algorithm*. The terminals are scanned in anti-clockwise ordering around the boundary beginning with some arbitrary terminal, and every new visited terminal is pushed onto a stack. If the pushed terminal is the last non-visited terminal of the corresponding net, it is tested if all terminals of the net lie on top of the stack. If this is not the case, the problem is not topologically solvable. Otherwise, all terminals of the net are popped. When all the terminals of all nets are visited, and there was no conflict before, the instance is topologically solvable if and only if the stack is empty.

For a formal description of the Stack Algorithm, let us assume that the nets $N_1, \ldots, N_k$ are numbered according to the occurrence of the *last* terminals of the nets, if we walk along the outer face boundary in anti-clockwise ordering (starting with some arbitrary but fixed terminal). For net N_i, we denote the *first* terminal by s_i and the *last* terminal by t_i.

"Stack Algorithm"

```
STACK:= ∅
begin
   walk along outer face boundary in anti-clockwise ordering starting with s_k;
   while not all terminals are visited do
        if the visited vertex v is a terminal then
          if v is a last terminal t_i then
             POP until s_i is popped;
             if not all popped terminals belong to N_i
             then stop: return "topologically unsolvable";
          else PUSH(v);
return " topologically solvable";
```

Obviously, the Stack Algorithm can be implemented to run in linear time. The algorithm to determine a solution, a *layout*, if it exists is now based on the Stack Algorithm. To route the nets correctly, they are considered in the order they have been deleted from the stack. According to our assumption, this is just the ordering $N_1, \ldots, N_k$. The nets are routed anti-clockwise along the boundary. After a net is routed, the boundary is corrected by deleting all used edges and vertices, and all edges incident to them. If there is enough capacity, this method leads to a correct layout.

This algorithm can be interpreted as a *right-first search*, i. e., a depth-first search where in each search step the edges are searched from right to left. We will say that the *next edge after* e w. r. t. v is the first edge to follow e in

the adjacency list of v in anti-clockwise ordering. A backtrack & remove step consists of a backtrack step where in addition the searched edge is deleted from the graph. We now describe the layout algorithm formally as a right-first search. For technical reasons we assume that all s_i have degree one in G.

"One-Face Layout Algorithm"

for $i := 1$ **to** k **do**
  let p_i initially consist of the unique edge incident to s_i;
  $v :=$ the unique vertex adjacent to s_i;
  while not all terminals of N_i are visited and $v \neq s_i$ **do**
    if at least one edge incident to v is not yet searched **then**
      let $\{v, w\}$ be the next edge after the leading edge of p_i w. r. t. v;
      if w is just occupied by some tree different from N_i **then**
        perform a backtrack & remove step;
      else add $\{v, w\}$ to p_i;
      $v := w$;
    else perform a backtrack & remove step;
    $v :=$ the leading vertex of p_i;
  if $v = s_i$ **then** stop: **return** "unsolvable";
return $(p_1, \ldots, p_k)$;

An example illustrating the One-Face Layout Algorithm is shown in Figure 1. The running time of the algorithm is again $\mathcal{O}(n)$.

2.2. The Two-Face Steiner Trees Packing Problem
The problem is much more complicated if the terminals are allowed to lie on two face boundaries.

Vertex-Disjoint Two-Face Steiner Trees Packing Problem

Given: A planar graph $G = (V, E)$, $|V| = n$, and pairwise disjoint sets $N_1, \ldots, N_k \subseteq V$. The graph G is embedded in the plane such that all terminals lie on the boundary of at most two different faces of G, w. l. o. g. the outer face F^{o} and one fixed inner face F^{i}.

Problem: Find, for each $i = 1, \ldots, k$, a Steiner tree T_i for N_i such that $T_1, \ldots, T_k$ are pairwise vertex-disjoint.

Suzuki, Akama & Nishizeki [32] give an algorithm for this problem that runs in time $\mathcal{O}(n \log n)$. There are essentially two cases to consider for solving this problem. The case where all terminals of the same net lie on the boundary of the same face, and the case that at least one net has terminals on both face boundaries. The algorithm presented in [32] for solving the first case has even linear running time. Surprisingly, in some sense it is the more difficult case of the problem. The first case is easier to solve since one single net having terminals on both faces fixes the homotopy of all other nets.

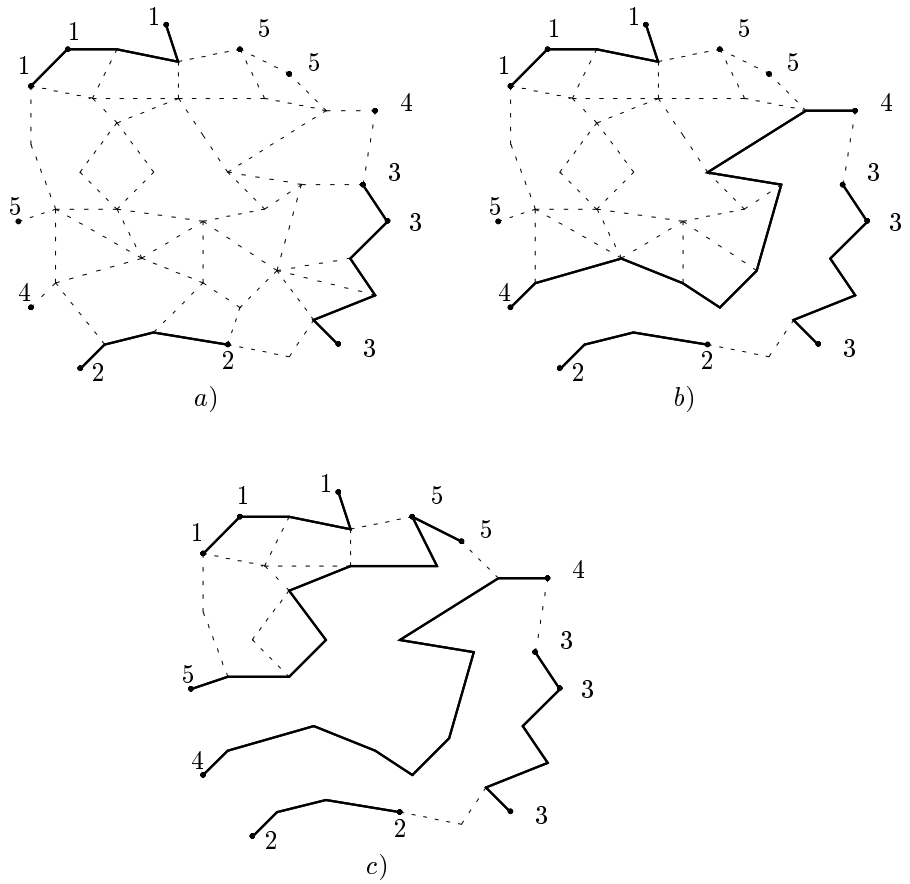

FIGURE 1. An example for the One-Face Layout Algorithm.

The algorithm for solving the second case consists of three parts. In the first part the problem is reduced to the vertex-disjoint *two-face paths packing problem.*

Vertex-Disjoint Two-Face Paths Packing Problem

Given: A planar graph $G = (V, E)$, $|V| = n$, and pairwise disjoint sets of terminal pairs $\{s_1, t_1\}, \ldots, \{s_k, t_k\} \subseteq V$. The graph G is embedded in the plane such that all terminals s_i lie on the boundary of one fixed inner face F^{i}, and all terminals t_i lie on the boundary of the outer face F^{o}.

Problem: Find k pairwise vertex-disjoint paths connecting s_i and t_i, for $1 \leq i \leq k$.

It is not hard to prove that the following reduction to the two-face paths packing problem is correct. The reduction is done by applying the One-Face

Layout Algorithm several times. At first, a layout for the nets having all terminals on the same face is constructed by applying the One-Face Layout Algorithm to the outer and to the inner face respectively. Then all parts of G occupied by this layout, as well as all edges incident to a vertex occupied by this layout, are deleted. Then for each of the remaining nets, trees connecting all terminals incident to the outer face F^{o} are determined by applying the One-Face Layout Algorithm. Analogously, trees connecting all terminals incident to the inner face F^{i} are determined. Now, all edges and vertices occupied by such a tree are contracted to a "super-vertex". These super-vertices induce a two-face paths packing problem. Obviously, all these determinations are done in time $\mathcal{O}(n)$.

The solution to this remaining problem is carried out in two parts (the second and third part of the whole algorithm).

In the second part of the algorithm, vertex-disjoint paths connecting the terminals on the inner face with some vertices on the outer face are determined. But, in fact these paths not necessarily connect the correct pairs of terminals. Basically, this subproblem of the two-face Steiner trees packing problem is just the "classical" *Menger problem*. Just introduce two new vertices s and t. By additional edges, connect all terminals s_i on the inner face F^{i} to s, and all terminals t_i on the outer face F^{o} to t.

Vertex-Disjoint Menger Problem

Given: A graph $G = (V, E)$, $|V| = n$ and vertices s and t.

Problem: Find a maximum set of (resp. k) internally vertex-disjoint paths connecting s and t.

A maximum number of vertex-disjoint paths connecting s and t in undirected graphs can be computed by solving a maximum unit-flow problem [2, 3]. If the graph is planar, this yields an $\mathcal{O}(n^{\frac{3}{2}})$ resp. $\mathcal{O}(kn)$ algorithm, where k is the number of paths. The approach presented in [32] leads to an algorithm of running time $\mathcal{O}(n \log n)$. It is based on divide-and-conquer techniques similar to the methods given in [8] and [24]. Recently, RIPPHAUSEN-LIPA, WAGNER & WEIHE [25, 27] presented a linear-time algorithm for the planar vertex-disjoint Menger Problem. We will discuss the latter approach in Section 2.3.

In the third part of the algorithm solving the two-face Steiner trees packing problem, vertex-disjoint paths connecting the correct terminals are determined. The approach given in [32] yields a linear time algorithm for this problem. In Section 2.4 we will sketch a different approach which is presented in [26]. This algorithm has linear running time as well. It is based on some special properties of the algorithm from [25] solving the Menger Problem.

2.3. The Menger Problem

Now we are going to introduce a linear-time algorithm for the vertex-disjoint Menger Problem. For technical reasons, we will consider a *directed* version of the Menger Problem instead of the undirected problem itself.

Directed Version of the Menger Problem

Given: A planar directed graph $G = (V, E)$, $|V| = n$, and vertices s and t.

Problem: Find a maximum set of (resp. k) internally vertex-disjoint (s, t)-paths, i. e., directed paths from s to t.

In the following we consider a directed graph corresponding to G. Let w. l. o. g. t be on the outer face boundary. The undirected graph G is transformed into a directed graph by replacing each edge $\{v, w\} \in E$ with $v, w \neq s$ by the arcs (v, w) and (w, v), and each edge $\{s, v\} \in E$ by (s, v) only. We denote the set of all arcs by A. Obviously, there is a linear-time algorithm solving the original Menger Problem for undirected graphs, if there is a linear-time algorithm solving the directed version of the Menger Problem for the corresponding directed graphs. Although the Menger algorithm presented here works in directed graphs, it does not solve the directed version of the Menger Problem in arbitrary directed graphs. It only works for directed graphs where (v, w) is an arc if and only if (w, v) is an arc. The directed formulation is used here only to distinguish between "searched from v to w" and "searched from w to v".

The algorithm consists in a loop over all arcs leaving s. Let $e_1, \ldots, e_k$ denote these arcs, in arbitrary order. In the i^{th} iteration the algorithm tries to draw a cycle-free (s, t)-path starting with e_i. These paths are determined by a *directed right-first search*, i.e., a directed depth-first search where in each search step all arcs leaving the current vertex are searched from right to left (except the reverse of the leading arc of the current search path). The i^{th} iteration is finished, when either t is reached or the search returns to s by backtrack steps. We call an iteration of the algorithm also a *search phase*.

The main difficulty of the approach is to handle *conflicts* of the current search path with itself, resp. with other paths, in an appropriate way. In fact, any conflict is resolved by a backtrack step. The idea is to handle all conflicts in a way that enables us to *remove each arc from* G, once we perform a backtrack step with it. As any step of such a right-first search is a forward step or a backtrack step, the number of search steps *in total* is then linear in the size of the graph. If in addition all search steps can be realized in (amortized) constant time, the time complexity of the whole algorithm is again linear.

In general, there are two types of conflicts to distinguish, *conflicts from the left* and *conflicts from the right*. That is, either the current search path touches some vertex occupied by another path or by the current search path itself from the *left* side or from the *right* side.

Conflicts from the left: If a vertex already occupied by another path or by the current search path itself is entered by the current search path from the left side, the conflict is resolved by simply performing a backtrack step. The corresponding arc is removed, and then the right-first search is continued.

Conflicts from the right: If the current search path enters a vertex occupied by *another* path from the right side, the conflict is resolved in the following way. Let v be the vertex where this conflict occurs, let p and q denote the segment

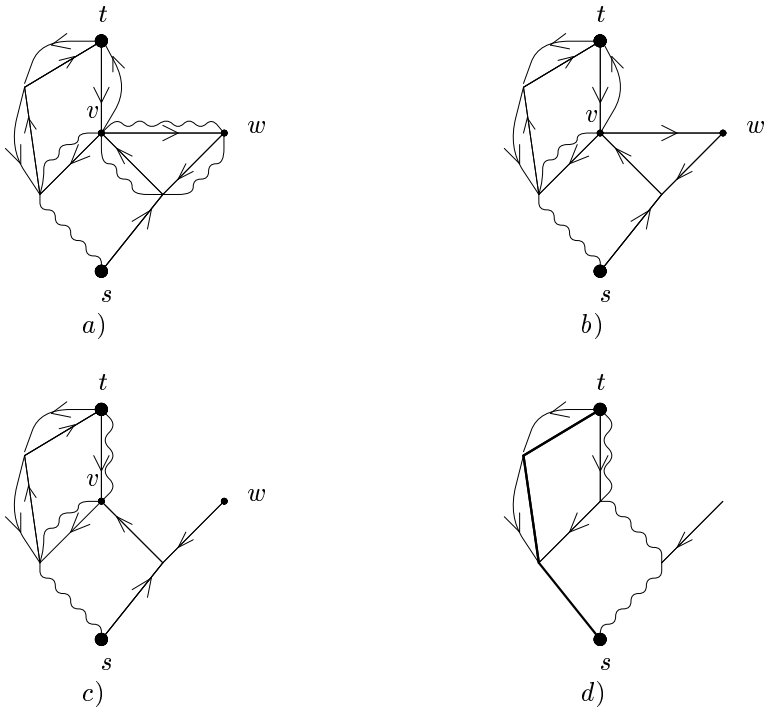

FIGURE 2. An example where a cycle conflict from the right is avoided by deleting arc (v, w): a) cycle conflict from the left, b) the situation after backtracking, c) the first path determined by the algorithm, d) the final solution.

of the other path from s to v and from v to t, respectively, and let r denote the current search path. We now concatenate r with q and let p be the "new" current search path. Now, the same conflict can be seen as a conflict from the left side. That is, the "new" current search path p enters the concatenated path $r + q$ from the left. Again, by a backtrack & remove step this conflict is resolved and the search is continued. By that, a path from s to t which is determined in one search phase, is not necessarily completely contained in the final solution as a path. This procedure may be seen as a "reconfiguration" of paths. In Figure 3 b) and c) the "reconfiguration" of paths is illustrated. The question is now: Can we resolve a *cycle conflict from the right*, i.e., a conflict of the current search path with itself where the occupied vertex is entered from the right side the second time?

Avoiding cycle conflicts from the right: In fact, since we are not able to cope with cycle conflicts from the right, we avoid those conflicts *in advance*. Fortunately, it can be proved that any cycle conflict from the right is in a sense "announced" by a cycle conflict from the left, where the same cycle is involved, but in reverse direction. Let (w, v) be the arc removed because of

this "announcing" cycle conflict from the left. Then clearly, removing (v, w) at this moment prevents the "announced" cycle conflict from the right. That is, in addition to the "regular" removing of arcs because of conflicts from the left, we sometimes remove reverse arcs of such arcs. The proceeding described here is illustrated in Figure 2.

Implementing the algorithm to run in linear time: As previously mentioned, the right-first search has linear running time if each search step can be performed in (amortized) constant time. Obviously, "regular" search steps, i.e., forward steps and remove steps in consequence of backtrack steps, only cost constant time. But, in order to decide if the reverse arc of a removed arc has to be removed as well, we must be able to distinguish between cycle conflicts from the left and conflicts where the current search path enters another path from the left. We could do this by comparing the "names" of the paths occupying the two end-vertices of such an arc. But, in consequence of the reconfiguration of paths, the "name" of the path occupying a vertex possibly changes during the algorithm. On the other hand, updating the names of the occupying paths every time a reconfiguration of paths takes place requires too much time.

This problem of "identifying" paths is solved by the following "trick". We maintain a *global time counter* and for each vertex a *local time stamp*. The global time counter is increased by 1, whenever the current search path changes, that is, when either a new search phase is started, or a conflict occurs where the current search path touches another path from the right side. The time stamp of a vertex is set to the value of the global time counter, whenever this vertex becomes the leading vertex of the current search path. Then the following procedure prevents all cycle conflicts from the right in advance: Whenever an arc is considered for going forward, we first compare the time stamp of its head with that of its tail. The arc is removed if and only if both are equal. Otherwise, it is actually used for going forward.

We now give a formal description of the algorithm. If an arc of a path p enters (leaves) a vertex v, we call it an *in-going arc* (*out-going arc*) of v. Let $e_1, \ldots, e_k$ be the arcs leaving s in arbitrary ordering.

"Menger Algorithm"

$time\_counter := 0$;
for $i := 1$ **to** k **do**
$time\_counter := time\_counter + 1$;
let the current search path be $e_i = (s, v)$;
repeat
  $time\_stamp(v) := time\_counter$;
  if the current search path touches some path at v from the left **then**
    perform a backtrack & remove step;
  else
    if the current search path touches some path at v from the right **then**
      $time\_counter := time\_counter + 1$;
      let p be the segment of this path from s to v, and let q be the remaining segment;

connect the current search path with q and let p be the current search path;
perform a backtrack & remove step;
else
if at least one arc leaving v is not yet searched (except the reverse of the in-going arc) **then** let (v, w) be the first such arc to appear after the in-going arc in anti-clockwise ordering;
if $time\_stamp(v) = time\_stamp(w)$ **then** remove (v, w);
else go forward via (v, w);
else perform a backtrack & remove step;
$v :=$ the leading vertex of the current search path;
until $v \in \{s, t\}$;

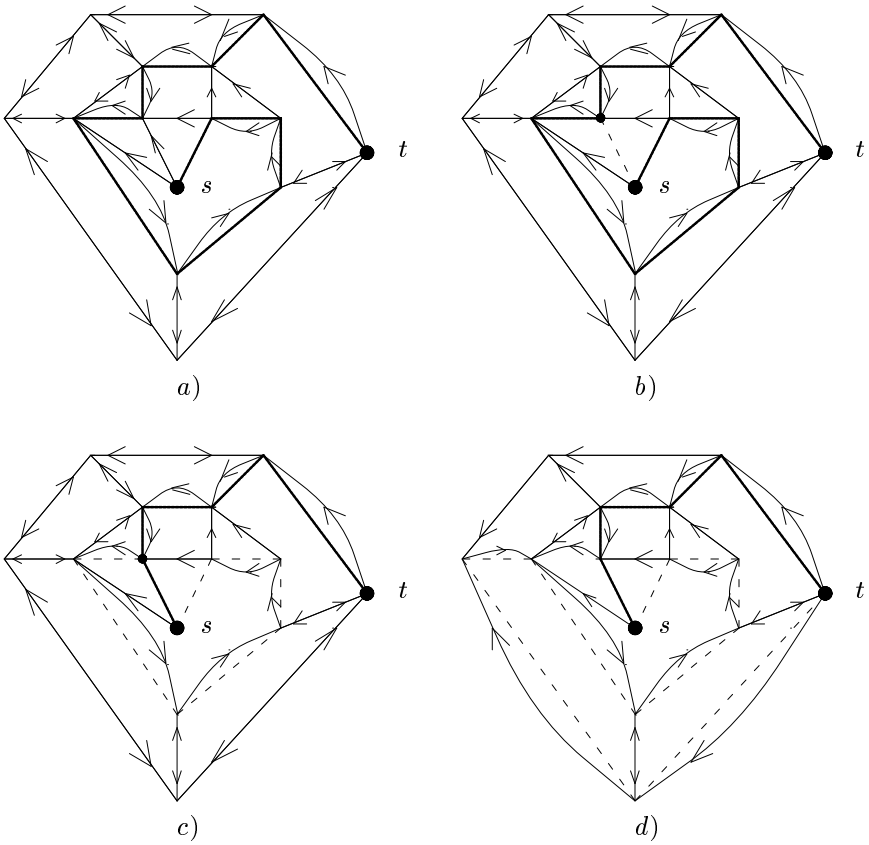

FIGURE 3. An example for the Menger Algorithm: a) the first path determined by the algorithm, b) a confict from the right between the the second search path and the first path in, c) the conflict from the left after reconfiguration of paths in, d) the first two paths determined by the algorithm.

THEOREM 2.1 [27]. *The vertex-disjoint Menger Problem in planar graphs can be solved in time $\mathcal{O}(n)$.*

The correctness of the Menger Algorithm is induced by the following two invariants maintained by the algorithm.

I1 During the execution of the algorithm no right-cycle conflict occurs.

I2 Let $\{a_1, \ldots, a_m\}$ be the arcs removed during the execution of the algorithm. Each graph $G_i = (V, A \setminus \{a_1, \ldots, a_i\})$, $1 \leq i \leq m$, still contains an optimal solution for G.

One very useful property of the Menger Algorithm is based on the following easy observation.

OBSERVATION 2.2. *At any stage of the algorithm, no arcs leave any path drawn so far to its right.*

Informally, we can say that the solution determined by the Menger Algorithm is a "rightmost" solution. This special solutions can be used to solve the two-face paths packing problem in linear time as well.

2.4. The Two-Face Paths Packing Problem

In [26], a method is presented for solving the two-face paths packing problem and some related problems in linear time. We sketch this approach here.

Obviously, the Menger Algorithm can be used to determine pairwise vertex-disjoint paths connecting specified terminals on one face to the boundary of the other face. Now, the Menger Algorithm is applied twice. On one hand, "inner" paths starting with the terminals s_i on the boundary of the inner face F^{i} and ending at some vertices on the boundary of the outer face F^{o} are determined. On the other hand, by a "left-first version" of the Menger Algorithm, "outer" paths starting with the terminals t_i on the boundary of the outer face F^{o} and ending at some vertices on the boundary of the inner face F^{i} are determined. Because of the special properties of the Menger Algorithm, the inner paths are in some sense rightmost and the outer paths are in some sense leftmost. Therefore, if the vertex-disjoint two-face paths packing problem is solvable, related inner and outer paths must intersect at least once. Then, a collection of paths connecting terminal s_i and t_i, for $1 \leq i \leq k$, can be constructed by concatenating appropriate segments of the inner paths with appropriate segments of the related outer paths. But, since "unrelated" inner and outer paths may intersect as well, not every choice of intersection vertices would yield a collection of pairwise vertex-disjoint paths. However, it can be proved that intersection vertices of related inner and outer paths exist which yield a vertex-disjoint solution (assumed a solution exists at all). These intersection vertices can be determined in linear time.

The method can be used to solve related vertex-disjoint paths packing problems, where the homotopies of the paths are given, in time $\mathcal{O}(n)$ as well [26].

3. The Edge-Disjoint Case

Most cases of the edge-disjoint Steiner trees packing problem are $\mathcal{NP}$-complete. Even the planar edge-disjoint paths packing problem, i.e., the N_i are all two-element sets and G is planar, is $\mathcal{NP}$-complete [14]. The problem is also $\mathcal{NP}$-complete if G is planar and $k = 2$ [13]. If $\sum_{i=1}^{k} |N_i|$ is fixed, the problem is polynomially solvable [29].

There are some special cases where the planar edge-disjoint paths packing problem turns out to be polynomially solvable. Typically, polynomial algorithms for such problems are based on the existence of certain duality results. That is, necessary and sufficient conditions for solvability are known. Obviously, a necessary condition for solvability is the *cut condition*.

A subset $X \subseteq V$ is called a *cut* of G. For a cut X the *capacity* of X, $cap(X)$, is the number of edges leaving X, and the *density* of X, $dens(X)$, is the number of nets leaving X, i.e.,

$$cap(X) := |\{\{u, v\} \in E : u \in X, v \in V \setminus X\}|,$$
$$dens(X) := |\{\{s_i, t_i\} \in \mathcal{N} : s_i \in X, t_i \in V \setminus X \text{ or } t_i \in X, s_i \in V \setminus X\}|.$$

The *free capacity* of X is defined as

$$fcap(X) := cap(X) - dens(X).$$

Cut Condition

A graph $G = (V, E)$ together with a set of terminal pairs $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ satisfies the *cut condition* if $fcap(X) \geq 0$ for all cuts $X \subseteq V$.The cut condition is a necessary condition for solvability, but in general it is not sufficient. An additional restriction on the problem, which sometimes makes things easier, is the *Eulerian condition* or *evenness condition*.

Eulerian Condition

A graph $G = (V, E)$ together with a set of terminal pairs $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ satisfies the *Eulerian condition* if and only if the graph $(V, E + \{s_1, t_1\} + \cdots + \{s_k, t_k\})$ is Eulerian. The two-terminal sets $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ are also called *demand edges*. Notice, that the terminals are not necessarily distinct.

The Eulerian condition is obviously equivalent to the condition that $fcap(v)$ is an even number, for all vertices $v \in V$. For instances where G together with the "demand" edges is Eulerian and planar, the cut condition is also sufficient for solvability. As a consequence, the problem is polynomially solvable [31]. It again becomes $\mathcal{NP}$-complete if either planarity or the Eulerian condition is dropped [18].

3.1. Solving the Eulerian One-Face Paths Packing Problem

A basic result of Okamura & Seymour [22] concerns the edge-disjoint one-face paths packing problem.

Edge-Disjoint One-Face Paths Packing Problem

Given: An instance $(G, \mathcal{N})$ consisting of a planar graph $G = (V, E)$ and a set of terminal pairs $\mathcal{N} = \{\{s_1, t_1\}, \ldots, \{s_k, t_k\}\}$. The graph G is embedded in the plane such that $s_1, \ldots, s_k, t_1, \ldots, t_k$ lie on the boundary of the outer face. (These vertices are not necessarily distinct.)

Problem: Find k edge-disjoint paths in G connecting s_i and t_i, for $1 \leq i \leq k$.

THEOREM 3.1 (OKAMURA & SEYMOUR) [22] *An instance $(G, \mathcal{N})$ of the edge-disjoint one-face paths packing problem that satisfies the Eulerian condition is solvable if and only if the cut condition is satisfied.*

The proof of Theorem 3.1 is constructive and yields an algorithm that preserves the cut condition and the Eulerian condition as invariants. The core of the algorithm can be formulated as follows:

"Algorithm of Okamura & Seymour"

while $E \neq \emptyset$ **do**
choose an edge $e = \{u, v\}$ on the outer face;
if there is a cut X with $u \in X, v \notin X$ and $fcap(X) < 0$ **then** stop;
(**cut condition is violated**)
if there is a cut X with $u \in X, v \notin X$ and $fcap(X) = 0$ **then**
choose an appropriate net $\{s, t\}$ with $s \in X, t \notin X$;
reserve e for $\{s, t\}$;
delete e;
replace $\{s, t\}$ by nets $\{s, u\}$, $\{v, t\}$; $(\star)$
else
delete e;
add a dummy net $\{u, v\}$; $(\star\star)$

The correctness of the algorithm is based on the following invariants.

I1 $(\star)$ and $(\star\star)$ preserve the cut condition.

I2 $(\star)$ and $(\star\star)$ preserve the Eulerian condition.

Invariant **I2** is obviously satisfied. Notice, that in $(\star\star)$ a dummy net is introduced in order to preserve the Eulerian condition. The proof of invariant **I1** consists of a case analysis, where the Eulerian condition is used several times.

Important for the efficiency of the algorithm is the following fact:

Fact *Restriction to connected cuts*
The cut condition is satisfied for all cuts $X \subseteq V$ if and only if the cut condition is satisfied for all *connected cuts* $X \subseteq V$, i.e., for all cuts $X \subseteq V$ where $G(X)$ and $G(V \setminus X)$ are connected graphs.

Only cuts whose density is at least one are considered during the algorithm of Okamura & Seymour. It is easy to prove that every connected cut X whose density is at least one cuts exactly two edges of the outer face boundary, i. e., there exist exactly two edges of the outer face boundary that are incident to both, a vertex in X and a vertex in $V \setminus X$. Thus, the conditions on cuts which are tested during the algorithm can be tested efficiently. Using the fact that minimum capacity cuts through two edges of the outer face boundary in G are equivalent to shortest paths between the corresponding vertices in the so-called *multiple source dual* [7, 33], of G leads to an $\mathcal{O}(n^2)$ implementation [1, 17]. The multiple source dual of G is obtained from the dual of G by introducing as many vertices corresponding to the outer face as there are edges on the boundary of the outer face. Then the edges dual to the edges on the boundary of the outer face are drawn such that the end-vertices corresponding to the outer face are distinct. In [10], the complexity of the algorithm has been improved to $\mathcal{O}(n^{\frac{5}{3}}(\log\log n)^{\frac{1}{3}})$ by using Frederickson's decomposition method for planar graphs [5].

Recently, WAGNER & WEIHE[34] introduced a new algorithm which solves the Eulerian edge-disjoint one-face paths packing problem in time $\mathcal{O}(n)$. In contrast to the algorithms mentioned above, it does not test cuts explicitly. The algorithm is, similar to the algorithms for the vertex-disjoint one-face paths packing problem and the vertex-disjoint Menger problem, based on "right-first search". We will explain the main ideas of this approach.

In the sequel we assume that, according to an anti-clockwise ordering starting with an arbitrary *start terminal* x, s_i precedes t_i for $i = 1, \ldots, k$, and t_i precedes t_{i+1} for $i = 1, \ldots, k-1$. All terminals have degree one and all other vertices have even degree. Obviously, a simple modification transforms any instance into a completely equivalent instance that fulfills this assumption.

Before we determine a solution for instance $(G, \mathcal{N})$, we will first consider an "easier" instance $(G, \mathcal{N}^{()})$ of *parenthesis structure*. That is, consider the $2k$-string of s-terminals and t-terminals on the outer face in anti-clockwise ordering, starting with x. The i^{th} terminal is assigned a *left parenthesis* if it is an s-terminal, and a *right parenthesis* otherwise. The resulting $2k$-string of parentheses is then a string of left and right parentheses that can be paired correctly, i. e., such that the pairs of parentheses are properly nested. The terminals are now newly paired according to this (unique) correct pairing of parentheses, i. e., an s-terminal and a t-terminal are paired if and only if the corresponding parentheses match. It is easy to see that $(G, \mathcal{N}^{()})$ is solvable, if $(G, \mathcal{N})$ is.

The following algorithm determines such a solution $(q_1, \ldots, q_k)$ for $(G, \mathcal{N}^{()})$. This solution will be used for the determination of the final solution. In contrast to the original nets, we denote the nets in $\mathcal{N}^{()}$ by $\{s_1^{()}, t_1^{()}\}, \ldots, \{s_k^{()}, t_k^{()}\}$, and we assume w. l. o. g. that $t_i = t_i^{()}$ for $i = 1, \ldots, k$. The paths q_i are determined by a right-first search. In principle, it proceeds in the same way as the "One-Face Layout Algorithm" introduced in Section 2.1. Let $v \in V$, and let e be incident

to v. We will say that the *next free edge after* e w. r. t. v is the first free edge to follow e in the adjacency list of v in reverse clockwise ordering.

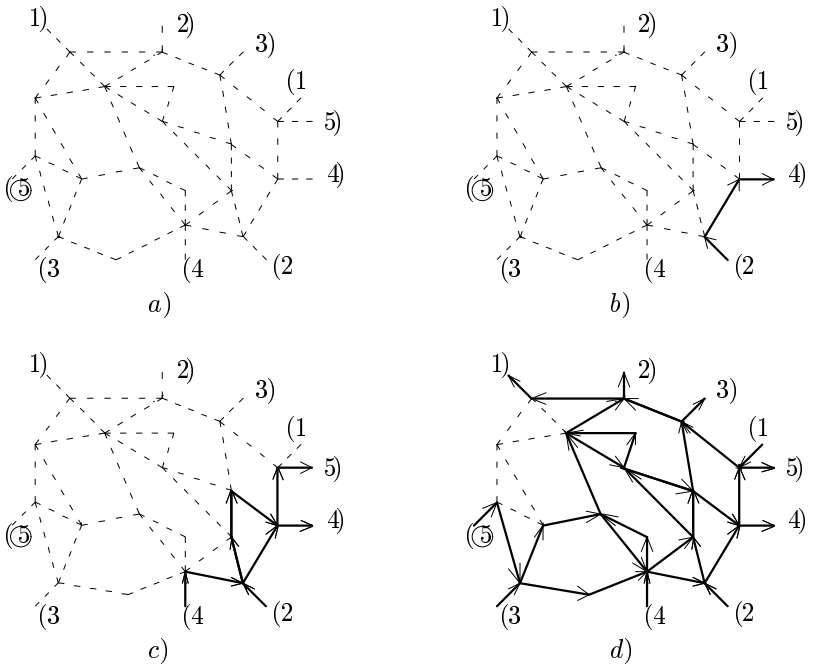

FIGURE 4. An example for the Algorithm for Parenthesis Problems: a) an Eulerian instance and the induced instance of parenthesis structure with respect to start terminal 5, b) the first path, c) first and second path, d) the auxiliary graph determined by the algorithm.

"Algorithm for Parenthesis Problems"

for $i := 1$ **to** k **do**
  let q_i initially consist of the unique edge incident to $s_i^{()}$;
  $v :=$ the unique vertex adjacent to $s_i^{()}$;
  while v is no terminal **do**
    let $\{v, w\}$ be the next free edge after the leading edge of q_i w. r. t. v;
    add $\{v, w\}$ to q_i;
    $v := w$;
  if $v \neq t_i^{()}$ **then** stop: **return** "unsolvable";
return $(q_1, \ldots, q_k)$;

An example illustrating the Algorithm for Parenthesis Problems is shown in Figure 4. The *auxiliary paths* $q_1, \ldots, q_k$ yield a directed *auxiliary graph* $A(G, \mathcal{N}, x)$ of instance $(G, \mathcal{N})$ w. r. t. start terminal x. Just orient all edges on the paths $q_1, \ldots, q_k$ according to the direction in which they are traversed

during the algorithm. Then $A(G,\mathcal{N},x)$ consists of all vertices of G and of all oriented edges. The following properties of the auxiliary paths and the auxiliary graph are easy to see.

LEMMA 3.2. *The auxiliary paths $q_1,\ldots,q_k$ neither cross themselves nor each other. In particular, the left and the right sides of all of them are well defined. All edges to the right of an auxiliary path q_i are contained in $A(G,\mathcal{N},x)$. The auxiliary graph $A(G,\mathcal{N},x)$ does not contain a right-cycle, i. e., a cycle whose interior is to its right.*

The paths $p_1,\ldots,p_k$ for the original instance $(G,\mathcal{N})$ are now determined in the auxiliary graph. That is, edges that are not contained in the auxiliary graph will not be occupied by a path $p_1,\ldots,p_k$ of the final solution. Even more, the edges occupied by the final solution are exactly the edges of the auxiliary graph. The solution paths p_i are determined by a "directed" right-first search. That is, edges that belong to $A(G,\mathcal{N},x)$ are used according to their orientations in $A(G,\mathcal{N},x)$.

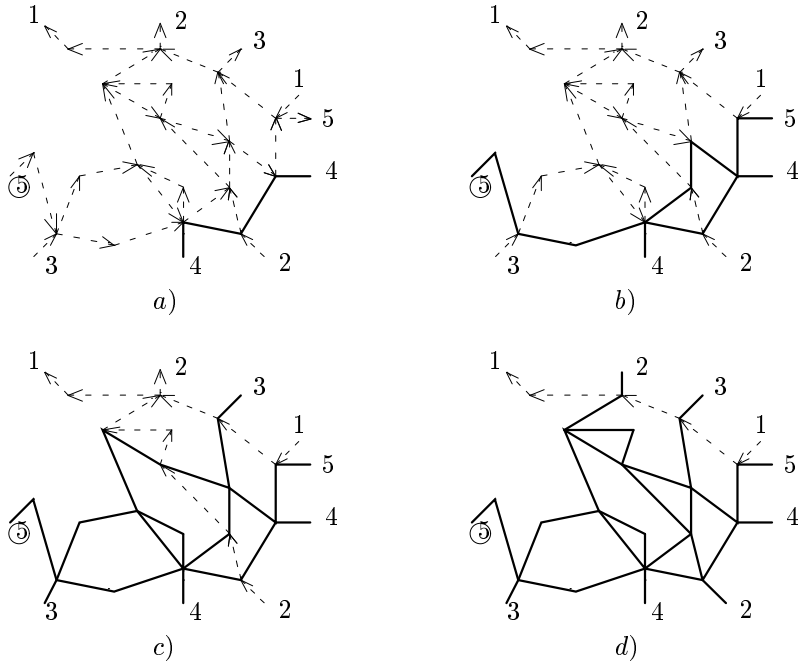

FIGURE 5. Edge-disjoint paths determined in the auxiliary graph shown in Figure 4: a) the first "final" path, b) the first and the second "final" paths, c) the first three "final" paths, d) the first four "final" paths determined by the algorithm.

"The Algorithm"

```
determine A(G, 𝒩, x) for an arbitrary start terminal x;
for i := 1 to k do
  let p_i initially consist of the unique edge leaving s_i in A(G, 𝒩, x);
  v := the head of this edge;
  while v is no terminal do
    let (v, w) be the next free edge leaving v after the leading edge of p_i w. r. t. v;
    add (v, w) to p_i;
    v := w;
  if v ≠ t_i then stop: return "unsolvable";
return (p_1, ..., p_k);
```

An example illustrating the algorithm is shown in Figure 5. The paths p_i determined by the algorithm have some nice properties similar to those of the auxiliary paths stated in Lemma 3.2. These properties are a consequence of the right-first search strategy.

LEMMA 3.3. *The paths $p_1, \ldots, p_k$ do not cross themselves. In particular, the left and the right side of each of them is well defined. All edges immediately to the right of a path p_i are either occupied by another path p_j, or contained in $A(G, \mathcal{N}, x)$ and directed towards p_i.*

The algorithm for parenthesis problems is just a right-first search in an undirected graph and is easily implemented to run in linear time. The main algorithm is a right-first search in a directed graph. For a linear-time implementation, a special case of Union-Find is used which also runs in linear time [6].

THEOREM 3.4 [34]. *A solution to a solvable Eulerian instance of the edge-disjoint one-face paths packing problem can be determined in time $\mathcal{O}(n)$.*

The correctness of the algorithm follows from two invariants maintained during the algorithm. For an instance $(G,\mathcal{N})$ and a path p_i determined by the algorithm, consider the induced *residual instance*. That is, the instance consisting of the subgraph of G induced by the edges that are not occupied by $p_1, \ldots, p_i$, and the set of nets $\{s_{i+1}, t_{i+1}\}, \ldots, \{s_k, t_k\}$. Then for a solvable instance $(G, \mathcal{N})$, the algorithm maintains the following invariants.

I1 For any path p_i, the induced residual instance is again solvable.

I2 For any path p_i, the subgraph of $A(G, \mathcal{N}, x)$ induced by the edges that are not occupied by $p_1, \ldots, p_i$ is equal to the auxiliary graph determined by the algorithm for parenthesis problems for the induced residual instance (resp. the corresponding instance of parenthesis structure).

To prove these invariants, it suffices to prove that path p_1 determined by the algorithm is correct, i.e., connects s_1 and t_1, and that **I1** and **I2** are correct for p_1. The main part of the correctness proof consists of the proof that **I1** is satisfied for p_1. The proof does not use the Theorem of Okamura & Seymour. Instead, it is proved that, for a solvable instance, there exists a solution

containing p_1. More precisely, it is shown that p_1 is just the unique *rightmost* path of all paths connecting s_1 and t_1 which are contained in a solution. The uniqueness of such a rightmost path follows from the Eulerian condition. In fact, this is the only statement in the proof of correctness where the Eulerian condition is necessary.

The correctness is based on the fact that the paths determined by the algorithm are in some sense "extremal". In fact, p_1 (resp. any path p_i) is the rightmost path connecting s_1 and t_1 (resp. s_i and t_i) with the property that the cut induced by the set of vertices lying to the right of p_1 (resp. p_i) is saturated (in the residual instance). This means that the paths determined by the algorithm "run along saturated cuts".

The approach of [34] does not only lead to an algorithm with linear running time to find a solution for any solvable instance, but also yields a linear-time algorithm to find a connected oversaturated cut for a non-solvable instance. It thus gives an alternative proof for the Theorem of Okamura & Seymour.

3.2. Solving the Weakly Even One-Face Paths Packing Problem

The complexity status of the non-even edge-disjoint one-face paths packing problem is open. If the Eulerian condition is relaxed to the *weak Eulerian* or *weak evenness condition*, there is again a necessary and sufficient cut condition for the solvability, the so-called *generalized cut condition* introduced by FRANK [4]. It again leads to a polynomial-time algorithm [1].

Weak Evenness Condition

An instance $(G,\mathcal{N})$ of the edge-disjoint one-face paths packing problem satisfies the *weak evenness* condition if and only if all interior vertices of G have even degree.

Generalized Cut Condition

A subset $Y \subseteq V$ is called *odd* if and only if the number of vertices of odd degree contained in Y is odd. A graph $G = (V, E)$, together with a set of terminal pairs $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$, satisfies the *generalized cut condition* if and only if $\sum_{i=1}^{l} fcap(X_i) \geq \frac{1}{2}q$ for all partitions $X_1, \ldots, X_l \subseteq V$, where q is the number of odd sets X_i, $1 \leq i \leq l$.

THEOREM 3.5 [4]. *An instance $(G,\mathcal{N})$ of the edge-disjoint one-face paths packing problem that satisfies the weak evenness condition is solvable if and only if the generalized cut condition is satisfied.*

Let us study the difference between even and weakly even solvable instances $(G,\mathcal{N})$. Consider a solution for $(G,\mathcal{N})$ and all edges of G that are not occupied by the solution. If $(G,\mathcal{N})$ is even, the edges not occupied by the solution induce an Eulerian graph, i.e., decompose into cycles. On the other hand, if the instance is weakly even, the edges not occupied by the solution induce a graph that decomposes into cycles and paths between the odd vertices. Thus, if a weakly even instance is solvable, it obviously satisfies the generalized cut condition.

The proof of the converse is constructive. It yields an efficient solution algorithm similar to the algorithm in [1]. The basic idea of this algorithm is to extend weakly even solvable instances into even solvable instances by introducing *dummy nets* between odd vertices on the outer face boundary. The existence of such an extension can be decided in time $\mathcal{O}(nu)$ (n number of vertices of G, u number of vertices on the outer face boundary). In case such an extension exists, it can be determined in time $\mathcal{O}(nu)$ as well. For the more restricted class of square grid graphs the running time reduces to $\mathcal{O}(n^{\frac{3}{2}})$ [1].

The complexity status of the general case, i.e., the case that not even the weak evenness condition holds, is open. If the graph together with the "demand" edges is planar, the non-even edge-disjoint one-face paths packing problem can be solved in linear time by a slightly modified version of the algorithm for parenthesis problems from Section 3.1. Observe that in this case the problem has parenthesis structure.

3.3. Further Versions of the Paths Packing Problem

There are several faster algorithms for special grid graphs, mostly based on the Okamura & Seymour approach. The improvement of the running time is based on the restricted shape of cuts to be considered. In [20] Nishizeki, Saito & Suzuki handle convex grids and achieve a running time of $\mathcal{O}(n)$. For more general grids, so-called general switchboxes, KAUFMANN & MEHLHORN [11] present a routing algorithm with running time $\mathcal{O}(n \log^2 n)$. If the class of generalized switchboxes with any horizontal or vertical cut crossing the boundary at most twice is considered, which also contains convex grids, the running time can be reduced to $\mathcal{O}(n)$ [9]. For grids of rectangular shape or channels, even a sublinear running time can be achieved. In these cases, the solution is specified by the positions of the bends performed by the paths. For further references on disjoint paths problems in grid graphs we also refer to [12] and [19].

There are some polynomially solvable variants of the edge-disjoint paths packing problem where the terminals are allowed to lie on two different face boundaries. Again, the existence of polynomial-time algorithms is based on the sufficiency of the cut condition.

THEOREM 3.6 (OKAMURA) [21]. *Consider an instance $(G, \mathcal{N})$ consisting of a planar graph $G = (V, E)$ and a set of terminal pairs $\mathcal{N} = \{\{s_1, t_1\}, \ldots, \{s_k, t_k\}\}$. The graph G is embedded in the plane such that $s_1, \ldots, s_l$, $t_1, \ldots, t_l$ lie on the boundary of one fixed inner face, and $s_{l+1}, \ldots, s_k, t_{l+1}, \ldots, t_k$ lie on the boundary of the outer face, for some l, $1 \leq l \leq k$. Let $(G, \mathcal{N})$ satisfy the Eulerian condition. There exist k edge-disjoint paths in G connecting s_i and t_i, for $1 \leq i \leq k$, if and only if the cut condition is satisfied.*

THEOREM 3.7 (OKAMURA) [21]. *Consider an instance $(G, \mathcal{N})$ consisting of a planar graph $G = (V, E)$ and a set of terminal pairs $\mathcal{N} = \{\{s_1, t_1\}, \ldots, \{s_k, t_k\}\}$, where $t_1 = t_2 = \cdots = t_l$ for some $l, 1 \leq l \leq k$. The graph G is embedded in the plane such that $s_{l+1}, \ldots, s_k, t_1, \ldots, t_k$ all lie on the boundary of the outer*

face. Let $(G,\mathcal{N})$ satisfy the Eulerian condition. There exist k edge-disjoint paths in G connecting s_i and t_i, for $1 \leq i \leq k$, if and only if the cut condition is satisfied.

The proofs of both theorems are constructive. In [33] a more general multicommodity flow problem is considered. The algorithms presented there lead to algorithms of time complexity $\mathcal{O}(n^2)$ for the cases described in Theorem 3.6 and Theorem 3.7.

Another case where the cut condition is also sufficient for the existence of edge-disjoint paths between terminals on at most two different faces is presented in [30]. But a (simple) algorithm with polynomial running time is not known for this problem.

THEOREM 3.8 (SCHRIJVER) [30]. *Consider an instance $(G,\mathcal{N})$ consisting of a planar graph $G = (V,E)$ and a set of terminal pairs $\mathcal{N} = \{\{s_1,t_1\},\ldots,\{s_k,t_k\}\}$. The graph G is embedded in the plane such that $s_1,\ldots,s_k$ lie on the boundary of one fixed inner face, and $t_1,\ldots,t_k$ lie on the boundary of the outer face, where the cyclic order of the $s_1,\ldots,s_k$ is converse to the order of the $t_1,\ldots,t_k$ is . Let $(G,\mathcal{N})$ satisfy the Eulerian condition. There exist k edge-disjoint paths in G connecting s_i and t_i, for $1 \leq i \leq k$, if and only if the cut condition is satisfied.*

REFERENCES

1. M. BECKER and K. MEHLHORN (1986). Algorithms for routing in planar graphs. *Acta Inform.*, **23**, 163–176.
2. S. EVEN and R. E. TARJAN (1975). Network flow and testing graph connectivity. *SIAM J. Comput.*, **4**, 507–518.
3. L. R. FORD and D. R. FULKERSON (1962). *Flows in networks.* Princeton University Press, Princeton.
4. A. FRANK (1985). Edge-disjoint paths in planar graphs. *J. Combin. Theory Ser. B*, **39**, 164–178.
5. G. N. FREDERICKSON (1987). Fast algorithms for shortest paths in planar graphs with applications. *SIAM J. Comput.* **16**.
6. H. N. GABOW and R. E. TARJAN (1985). A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.* **30**, 209–221.
7. R. HASSIN (1984). On multicommodity flows in planar graphs. *Networks* **14**, 225–235.
8. R. HASSIN and D. B. JOHNSON (1985). An $\mathcal{O}(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM J. Comput.* **14**, 612–624.
9. M. KAUFMANN (1990). A linear time algorithm for routing in a convex grid. *IEEE Trans. Comp.-Aided Design*, CAD-**9**, 180–184.
10. M. KAUFMANN and G. KLÄR (1991). A faster algorithm for edge-disjoint paths in planar graphs. In: W. L. HSU and R. C. T. LEE, editors, *ISA'91 Algorithms, 2nd International Symposium on Algorithms*, pages 336–348. Springer-Verlag, Lecture Notes in Computer Science **557**.

11. M. KAUFMANN and K. MEHLHORN (1985). Generalized switchbox routing. *J. Algorithms* **7**, 510–531.
12. M. KAUFMANN and K. MEHLHORN (1990). Routing problems in grid graphs. In: B. KORTE, L. LOVÁSZ, H. J. PRÖMEL and A. SCHRIJVER, editors, *Paths, Flows and VLSI-Layout*, pages 165–184. Springer-Verlag.
13. B. KORTE, H. J. PRÖMEL and A. STEGER (1990). Steiner trees in VLSI-layout. In: B. KORTE, L. LOVÁSZ, H. J. PRÖMEL and A. SCHRIJVER, editors, *Paths, Flows, and VLSI-Layout*, pages 185–214. Springer-Verlag.
14. M. R. KRAMER and J. VAN LEEUWEN (1984). The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI-circuits. In: F. P. PREPARATA, editor, *Advances in Computer Research, VOL 2: VLSI Theory*, pages 129–146. JAI Press Inc..
15. K.-F. LIAO and M. SARRAFZADEH (1990). Vertex-disjoint trees and boundary single-layer routing. In: R. H. MÖHRING, editor, *Proceedings 16th International Workshop on Graph-Theoretic Concepts in Computer Science, WG'90*, pages 99–108. Springer-Verlag, Lecture Notes in Computer Science, vol. 484.
16. J. F. LYNCH (1975). The equivalence of theorem proving and the interconnection problem. *ACM SIGDA Newsletter* **5**, 31–65.
17. K. MATSUMOTO, T. NISHIZEKI and N. SAITO (1985). An efficient algorithm for finding multicommodity flows in planar networks. *SIAM J. Comput.* **14**, 289–302.
18. M. MIDDENDORF and F. PFEIFFER (1993). On the complexity of the disjoint path problem. *Combinatorica* **13**, 97–107.
19. R. H. MÖHRING, D. WAGNER and F. WAGNER (1992). *VLSI network design: A survey.* Technical Report 323, Technische Universität Berlin.
20. T. NISHIZEKI, N. SAITO and K. SUZUKI (1985). A linear time routing algorithm for convex grids. *IEEE Trans. Comp.-Aided Design*, CAD-**4**, 68–76.
21. H. OKAMURA (1983). Multicommodity flows in graphs. *Discrete Appl. Math.* **6**, 55–62.
22. H. OKAMURA and PAUL D. SEYMOUR (1981). Multicommodity flows in planar graphs. *J. Combin. Theory Ser. B* **31**, 75–81.
23. B. REED, N. ROBERTSON, A. SCHRIJVER and P. D. SEYMOUR (1993). Finding disjoint trees in graphs on surfaces. Preprint.
24. J. H. REIF (1983). Minimum s-t-cut of a planar undirected network in $\mathcal{O}(n \log^2(n))$ time. *SIAM J. Comput.* **12**, 71–81.
25. H. RIPPHAUSEN-LIPA, D. WAGNER and K. WEIHE (1992). *The vertex-disjoint Menger-problem in planar graphs.* Technical Report 324, Technische Universität Berlin.
26. H. RIPPHAUSEN-LIPA, D. WAGNER and K. WEIHE (1993). Linear-time algorithms for disjoint two-face paths packing problems in planar graphs. To appear in *Proceedings of the 4th International Symposium on Algorithms and Computation, ISAAC'93.* Springer-Verlag, Lecture Notes in Computer Science.

27. H. Ripphausen-Lipa, D. Wagner and K. Weihe (1993). The vertex-disjoint Menger-problem in planar graphs (extended abstract). In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'93*, pages 112–119.
28. N. Robertson and P. D. Seymour (1986). Graph minors VI, disjoint paths across a disc. *J. Combin. Theory Ser. B* **41**, 115–138.
29. N. Robertson and P. D. Seymour (1990). An outline of a disjoint paths algorithm. In: B. Korte, L. Lovász, H. J. Prömel and A. Schrijver, editors, *Paths, Flows, and VLSI-Layout*, pages 267–292. Springer-Verlag, Berlin.
30. A. Schrijver (1989). The Klein bottle and multicommodity flow. *Combinatorica* **9**, 375–384.
31. P. D. Seymour (1981). On odd cuts and plane multicommodity flows. *Proc. London Math. Society* **42**, 178–192.
32. H. Suzuki, T. Akama and T. Nishizeki (1990). Finding Steiner forests in planar graphs. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'90*, pages 444–453.
33. H. Suzuki, T. Nishizeki and N. Saito (1989). Algorithms for multicommodity flows in planar graphs. *Algorithmica* **4**, 471–501.
34. D. Wagner and K. Weihe (1993). A linear time algorithm for edge-disjoint paths in planar graphs. In: T. Lengauer, editor, *First European Symposium on Algorithms, ESA'93*. Springer-Verlag, Lecture Notes in Computer Science.