Discrete Applied Mathematics 92 (1999) 91–110

DISCRETE APPLIED MATHEMATICS

PlaNet
A software package of algorithms and heuristics for disjoint paths in *Pla*nar *Net*works ☆

Ulrik Brandes[a], Wolfram Schlickenrieder[b], Gabriele Neyer[c], Dorothea Wagner[a, *], Karsten Weihe[a]

[a] *Universität Konstanz, Fakultät für Mathematik und Informatik, D 188, 78457 Konstanz, Germany*
[b] *ETH Zürich, Institute for Operations Research, ETH Zentrum–CLV B3, 8092 Zürich, Switzerland*
[c] *ETH Zürich, Institute for Theoretical Computer Science, IFW B27.1, 8092 Zürich, Switzerland*

Received 23 June 1997; revised 1 September 1998; accepted 9 November 1998

Abstract

We present a package for algorithms on planar networks. This package comes with a graphical user interface, which may be used for demonstrating and animating algorithms. Our focus so far has been on disjoint path problems. However, the package is intended to serve as a general framework, wherein algorithms for various problems on planar networks may be integrated and visualized. For this aim, the structure of the package is designed so that integration of new algorithms and even new algorithmic problems amounts to applying a short "recipe." The package has been used to develop new variations of well-known disjoint path algorithms, which heuristically optimize additional $\mathcal{NP}$-hard objectives such as the total length of all paths. We will prove that the problem of finding edge-disjoint paths of minimum total length in a planar graph is $\mathcal{NP}$-hard, even if all terminals lie on the outer face, the Eulerian condition is fulfilled, and the maximum degree is four. Finally, as a demonstration how *PlaNet* can be used as a tool for developing new heuristics for $\mathcal{NP}$-hard problems, we will report on results of experimental studies on efficient heuristics for this problem. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Algorithms engineering; Planar graphs; Disjoint paths; Length minimization

☆ This work was supported by the Technische Universität Berlin under grant FIP 1/3 and by the Deutsche Forschungsgemeinschaft under grant Wa 654/10–2.

* Corresponding author. Fax: + 49 7531 88 3577

E-mail addresses: {ulrik.brandes, dorothea.wagner, karsten.weihe} @uni-konstanz.d, WWW http://www.informatik.uni–konstanz.de/ {brandes, wagner,weihe} (U. Brandes, D. Wagner, K. Weihe), neyer@inf.ethz.ch, WWW http://www.inf.ethz.ch/ personal/neyer (G. Neyer), schlicke@ifor.math.ethz.ch, WWW, http://www.ifor.math.ethz.ch/staff/schlicke (W. Schlickenrieder).

1. Introduction

We present a package to display and animate algorithms that work on planar networks. So far, we have been concentrating on disjoint path algorithms, because this is the field of our theoretical interest. See [19] for a survey. On one hand, the aim of this package is to demonstrate these algorithms by means of a graphical user interface (*GUI*). On the other hand, it can be used as a tool for developing new heuristics for $\mathcal{NP}$-hard problems and for experimental studies of such heuristics. The package is implemented in C++ and runs on Sun Sparc stations, and the graphics are based on the X11 Window System. A restricted demo version, the sources, and some additional information are available in the World Wide Web (URL http://www.informatik.uni-konstanz.de/Forschung/Projekte/PlaNet/). In this paper, we shall describe the full version.

Instances for the different algorithmic problems may be generated either interactively or by a random generator, stored externally in files, and read from the respective file again. All instances are strongly typed, that is, each instance is assigned a (unique) algorithmic problem to which it belongs. This classification of instances is reflected by the GUI: the user cannot invoke an algorithm with an instance of the wrong type. Another advantage of this classification is that data structures for different algorithmic problems may be implemented differently. For example, if an algorithmic problem is restricted to grid graphs, instances of this problem should be implemented as two-dimensional arrays, whereas general planar graphs should usually be implemented by adjacency lists.

Like instances, all algorithms are classified according to the problems they solve, and this classification is also reflected by the GUI. For each algorithmic problem there may be an arbitrary number of algorithms solving this problem. For two algorithmic problems, P_1 and P_2, let $P_1 \preccurlyeq P_2$ indicate that P_1 is a special case of P_2. Such relations of problem classes can be integrated into the package and are then also reflected by the GUI. More specifically, an algorithm solving problem P may be applied not only to instances of type P, but also to instances of any type P' such that $P' \preccurlyeq P$. In other words, although all instances and algorithms are strongly typed by the corresponding algorithmic problems, an algorithm may be applied to any instance for which it is suitable from a theoretical point of view, even if the types of the algorithm and the instance are different.

In Section 2, we describe the GUI in greater detail. The internal structure of the package is designed to support the integration of new algorithms. As a consequence, developers may insert new algorithms and algorithmic problems by applying a short “recipe”, which requires no knowledge about the internals. For this aim, the internal structure of the package has been designed in a framework-like manner. In Section 3, we introduce our overall design and explain this recipe. In Section 4, we will describe the algorithms that have been integrated so far. To our surprise, it has turned out that generating suitable probability distributions for special planar problem classes is not straightforward, and designing random generators took us a lot of time.

In Section 5 we will report on new, heuristic, variations of known algorithms. The problem of finding edge-disjoint paths in a planar graph such that each path connects two specified vertices on the outer face of the graph is well studied. The “classical” Eulerian case introduced by Okamura and Seymour [16] is solvable in linear time [26]. So far, the length of the paths were not considered. In this paper, we prove that the problem of finding edge-disjoint paths of minimum total length in a planar graph is $\mathcal{NP}$-hard, even if the graph fullfills the Eulerian condition and the maximum degree is four. Minimizing the length of the longest path is $\mathcal{NP}$-hard as well. Efficient heuristics based on the algorithm from [26] are presented that determine edge-disjoint paths of small total length. We have studied their behavior, and it turns out that some of the heuristics are empirically quite successful. The report on heuristics in Section 5 will serve as an example how *PlaNet* can be used as a tool for developing new heuristics for $\mathcal{NP}$-hard problems and for experimental studies of such heuristics.

2. From a user’s perspective

At every stage of a session with PlaNet, there is a *current algorithmic problem* P. The problem P indicates the node in the problem class hierarchy on which the user currently concentrates. In the beginning, P is void, and the user always has the opportunity to replace P by another problem in the hierarchy. A new current problem may be chosen directly from the list of all problems or by navigating over the problem class hierarchy. In the latter case, a single navigation step amounts to replacing P either by one of its immediate descendants or by one of its immediate ancestors. To initialize P in the beginning, each of the topmost (i.e., most general) problems may be used as an entry point for navigation.

Once the current algorithmic problem is initialized, the user may construct a *current instance* and choose one or two *current algorithms*. Afterwards, the user may apply these two algorithms simultaneously to the current instance in order to compare their results.

An instance may be generated or read from a file only if the type P' of the instance satisfies $P' \preccurlyeq P$. Analogously, an algorithm may be chosen only if the type P'' of the algorithm satisfies $P \preccurlyeq P''$. In particular, this guarantees that the algorithm is suitable for the instance. These restrictions on instances and algorithms are enforced by the GUI: to select an algorithm, the user must pick it out of a list, which is collected and offered by the GUI on demand. This list only contains algorithms of appropriate types (namely types P'' such that $P'' \succcurlyeq P$). Analogously, the lists of random instance generators and of externally stored instances only contain items of appropriate types (types P' such that $P' \preccurlyeq P$).

When applying one or two algorithms to an instance, the GUI not only shows the final results, it also displays the execution of an algorithm step-by-step. After each modification of the display, the algorithm stops for a prescribed *wait time*. By default, this wait time is zero, and the user only observes the final result, because the display of the procedure is too fast. The user may change this wait time to an arbitrary multiple of 1/1000 sec. in order to observe the procedure step-by-step.

Many algorithms consist of a small number of major steps, for example, preprocessing and core procedure. For each major step of an algorithm, a separate window is opened. The initial display of such a window is the result of the former major step or – if it is the very first major step – the plain input instance. The procedure of this major step is displayed in the associated window, and afterwards the window remains open showing the final result of the major step. Therefore, on termination of the whole algorithm, all intermediate stages are shown simultaneously and may be compared with each other.

The GUI offers a feature to print the contents of a window or to dump it to a file in PostScript format.

We refer to [8] for a tutorial, a more detailed overview, and a reference manual.

3. From a developer's perspective

The implementation heavily relies on the object-oriented features of C++. Here we do not say much about object-oriented programming; see for example [13] for a thorough description of object-oriented programming, and see [21] for a description of C++. For further details of our internal design, we again refer to [8]. Nonetheless, in order to make this section self-contained, we first introduce all terminology that we need to describe the internal design.

Classes. Classes are a means of modeling *abstract data types*. For example, the abstract data type "stack" is essentially defined by the subroutines "push", "pop", and "top". A stack class wraps an interface around a concrete implementation of stacks (*encapsulation*), which consists solely of such subroutines (usually called the *methods* of the stack class). Consider an algorithm which works on stacks and whose implementation uses this stack class. In such an implementation, only the interface may be accessed; the concrete implementation behind the interface is hidden from the rest of the code. This allows software developers to adopt a higher, more abstract point of view, simply by disregarding all technical details of the implementation of abstract data types.

Inheritance. A class A may be *derived* from another class B. This means that the interface of A is the same as or an extension of the interface of B, even if the concrete implementation behind the interface is completely different for A and B. Moreover, an object of type A may be used wherever an object of type B is appropriate. For example, a formal parameter of type B may be instantiated by an actual parameter of type A. A class may be derived from several classes (*multiple inheritance*). Therefore, inheritance may be used to model relationships between special cases and general cases. Moreover, inheritance may be used for "code sharing," that is, all methods of class B may be called by the methods of class A to perform central tasks.

Dynamic polymorphism. This is another application of inheritance. Dynamic polymorphism means that classes $A_1, \ldots, A_k$ are derived from a common "*polymorphic*" class B, but the access methods of B are only declared, not implemented. Here inheritance is simply used to make $A_1, \ldots, A_k$ exchangeable: a formal parameter of type B is used whereever it does not matter which of $A_1, \ldots, A_k$ is the type of the actual parameter.

This concludes the introduction into object-oriented terminology. In PlaNet, each algorithmic problem is modeled as a class, and inheritance is used to implement the relation "$\prec$". The topmost element of the inheritance hierarchy is the basic LEDA graph class [11,12]. Consequently, each problem class offers all features of the LEDA class by code sharing.

We paid particular attention to the problem of integrating new algorithms and new problem classes into the package afterwards. The problem classes and their inheritance hierarchy are described by a file named *classes.def*, in a high-level, descriptive language, which is much simpler than C++. Each problem class is represented by an item within *classes.def*, which consists of one clause for each piece of information: the name of the problem class, the classes from which it is derived, a default directory for instances of the new problem class, and the name of a documentation file ("help file") for this problem class. Moreover, the item of a problem class contains an arbitrary number of subitems for algorithms, which solve this problem. The project makefile scans *classes.def* and integrates all problem classes and their inheritance relations, and all solvers, into the package.

Therefore, integration of a new problem class amounts to inserting a new item in *classes.def*. In addition, a file named *Config* must be changed slightly. This file consists of definitions of two string variables and a line that executes the project makefile depending on the contents of these two strings. These two strings define all additional object files and their directories, respectively. The project makefile searches each such directory for a subproject makefile. If there is one, it is executed in order to generate the object files, otherwise a default rule is applied to generate all object files in this directory. After that, these object files are collected in a library, and this library is linked to the rest of the package. When several developers work on the integration of different new problem classes and algorithms in the package simultaneously, each developer needs to maintain his/her own local copy of *classes.def* and of this small file *Config*; all other stuff may be shared.

Besides the advantages of encapsulation discussed above, we use encapsulation for several further important design goals, notably the task of separating the code for graphics from the code for the algorithms. Since the GUI displays not only the output of an algorithm but also its procedure, graphics and algorithms are strongly coupled. However, it is highly desirable to separate algorithms and graphics strictly from each other. In fact, otherwise algorithms and graphics cannot be modified independently of each other, and changing a small part of the package may result in a chain of modifications spread all over the package. This means that maintaining and modifying the package is simply not feasible. Several authors of this paper had discouragingly bad experiences with packages wherein the algorithms were “messed” with the graphics.

In PlaNet, the graphical display is delegated to the underlying problem class. This means the following: the most general problem class, called *net_graph*, encapsulates a reference to an additional object, which serves as a connection to the underlying graphic system. Clearly, this object is inherited by all problem classes. This object is of a polymorphic class type, and from this class type another class is derived, which realizes the graphical display under the X11 Window System. To run the package under another graphical system, it is only necessary to derive yet another class from this polymorphic class.[1] If one or more algorithms shall be extracted and run without any graphics, a dummy class must be derived, whose methods are void.

In a preceding project, *CRoP* [23,25], which solves combinatorial VLSI routing problems, graphics and algorithms were separated from each other as follows. An algorithm produces not only a final result, but also records all actions that may be relevant for the graphical display in an additional output data structure. Afterwards, these recordings are handed over to the implementation of the GUI, which passes over them to display the procedure of the algorithm. In our experience, the object-oriented solution applied in PlaNet is much more appropriate.

4. Algorithms

So far, the algorithmic problems modeled with PlaNet mainly reflect our theoretical research interests.

- Algorithms to compute a random triangulation and a Delaunay triangulation, respectively ([7], cf. [2,17]). These algorithms are mainly intended to serve as a basis for random generation of instances.
- Various smaller algorithms such as removing a random couple of edges, random paths, and the like. These procedures are mainly intended to support flexible random instance generation, too.
- The vertex-disjoint Menger problem, that is, find the maximum number of internally vertex-disjoint paths such that all paths connect the same pair $\{s,t\}$ of vertices. The linear-time algorithm from [18,20] has been integrated.
- An algorithm for computing a minimum (s,t)-separator given a maximum number of vertex-disjoint (s,t)-paths.
- The edge-disjoint version of the former problem. The linear-time algorithm from [28,29] has been integrated.
- Further versions of the vertex- and edge-disjoint Menger problems, respectively, where the task is to find the maximum number of vertex-disjoint paths such that each

[1] In terms of *design patterns*, this concept implements the *observer* and the *bridge* pattern [5].

path connects two vertices out of a given triple $\{s,t,u\}$. New linear-time algorithms for these two problems have been integrated [14].

- The Okamura–Seymour problem [16]: Given a planar graph and pairs of vertices $\{s_1,t_1\},\dots,\{s_k,t_k\}$ on the outer face (*terminals*) such that the so-called *Eulerian condition* is satisfied, find edge-disjoint paths $p_1,\dots,p_k$ such that each path p_i connects $\{s_i,t_i\}$. In that, an instance is said to fulfill the Eulerian condition if, for each vertex, the degree plus the number of terminals placed on this vertex sum up to an even number. The linear-time algorithm from [24,26] has been integrated.
- Several variations of the former algorithm, which heuristically minimize the total length of all paths. We report on these in Section 5.

All of our random instance generators apply the following two steps. First a triangulation is constructed (note that the triangulations are exactly the maximal planar graphs with respect to insertion of edges). Afterwards, a couple of edges, paths, or whatsoever are removed. Our experience is that suitable random distributions can be implemented this way much more easily than, for example, by constructing random graphs incrementally. For example, a class of suitable random distributions for graphs subject to the Eulerian condition can be relatively easily implemented by removing a couple of edge-disjoint paths from a triangulation, namely such that the number of paths meeting a vertex is even if and only if this vertex already fulfills the Eulerian condition in the initial triangulation. In contrast, an incremental approach causes a lot of difficult technical problems (e.g., maintenance of planarity throughout the procedure).

5. Edge-disjoint paths with short length

In this section now, we consider the problem of finding edge-disjoint paths of short length in a planar graph. We prove that the problem of finding edge-disjoint paths of minimum total length in a planar graph is $\mathcal{NP}$-hard, even if all terminals lie on the boundary of the outer face, the graph fulfills the Eulerian condition, and the maximum degree is four. Minimizing the length of the longest path is $\mathcal{NP}$-hard as well. Efficient heuristics based on the algorithm from [26] are presented that determine edge-disjoint paths of small total length.

Let $G=(V,E)$ be a simple, undirected, planar graph 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*. Consider a set $N=\{\{s_1,t_1\},\dots,\{s_k,t_k\}\}$, where $s_1,t_1,\dots,s_k,t_k$ are vertices of G on the boundary of the outer face. The elements of N are called *nets* and the s_i,t_i are called *terminals*. Notice that the terminals are not necessarily different. A graph $G=(V,E)$ together with a set of nets $N=\{\{s_1,t_1\},\dots,\{s_k,t_k\}\}$ satisfies the *Eulerian condition* if and only if the graph $(V,E+\{s_1,t_1\}+\cdots+\{s_k,t_k\})$ is Eulerian. The problem is to determine edge-disjoint paths $p_1,\dots,p_k$, such that p_i connects s_i with t_i for $i=1,\dots,k$. The basic result due to Okamura and Seymour is a theorem that gives a necessary and sufficient condition for solvability [16]. Efficient algorithms based on the proof of this theorem are given in [1,9,10]. An algorithm solving the problem in linear time is presented in [26]. The complexity status is open for the case when the Eulerian condition is dropped.

So far, the length of the paths were only considered for very restricted cases, i.e., where the graph is a rectilinear grid [3,22,27]. The only case known, where edge-disjoint paths of minimum total length can be determined in polynomial time is when the instance is a dense channel [3]. In this case, the problem is even solvable in time linear in the number of nets [22]. For convex grids, an efficient heuristic to determine edge-disjoint paths of small total length is presented in [27]. In this paper, we prove that finding edge-disjoint paths of minimum total length in planar Eulerian instances of maximum degree four is $\mathcal{NP}$-hard, and minimizing the length of the longest path is $\mathcal{NP}$-hard as well. We present efficient heuristics based on the algorithm from [26] that determine edge-disjoint paths of small total length. The heuristics have been implemented and included in PlaNet. They are studied empirically, and for not too large instances the results are compared to the solutions determined by an exact approach[2] with exponential worst-case time complexity.

5.1. Preliminaries

Definition 5.1. An *optimization instance* is a pair (G,N) consisting of a planar graph $G=(V,E)$ and a set of terminal pairs $N=\{\{s_1,t_1\},\dots,\{s_k,t_k\}\}$. The graph G is embedded in the plane such that the terminals $s_1,\dots,s_k,t_1,\dots,t_k$ lie on the boundary of the outer face. (G,N) satisfies the Eulerian condition. A *decision instance* consists of an optimization instance (G,N) and a nonnegative integer K.

Problem 5.2. Edge-Disjoint Paths Problem

Given: *An optimization instance* (G,N).

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

In the following, we assume G to be biconnected. For a non-biconnected graph the problem can be easily solved by considering its biconnected components separately [16]. We first outline the algorithm from [26], which solves the edge-disjoint paths problem in linear time. For technical reasons, G is modified such that all terminals have degree 1 and all other vertices have even degree. Obviously, an instance can easily be transformed into a completely equivalent instance that fulfills this assumption. Now, let x be an arbitrary terminal, called the *start terminal*. Without loss of generality, according to a counterclockwise ordering of all terminals starting with x, s_i precedes t_i for $i=1,\dots,k$, and t_i precedes t_{i+1} for $i=1,\dots,k-1$. The latter clearly means that in a sense all t-terminals are sorted in increasing order. The algorithm is based on “right-first

[2] The authors want to thank Martin Oellrich and Andreas S. Schulz for making the implementation of an exact method using CPLEX available.

search", i.e., a depth-first search where in each search step the edges are searched from right to left. It consists of two phases. In the first phase, an "easier" instance $(G,N^{()})$ of *parenthesis structure* is solved. Then in the second phase, a solution to the instance (G,N) is determined based on the solution for $(G,N^{()})$.

For the first phase, consider the $2k$-string of s-terminals and t-terminals on the outer face in counterclockwise ordering, starting with x. The ith 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 $(G,N^{()})$ is solvable, if (G,N) is. Procedure 5.3 determines such a solution $(q_1,\dots,q_k)$ for $(G,N^{()})$. This solution will be used to determine the final solution. In contrast to the original nets, we denote the nets in $N^{()}$ by $\{s_1^{()},t_1^{()}\},\dots,\{s_k^{()},t_k^{()}\}$, and we assume without loss of generality that $t_i = t_i^{()}$ for $i=1,\dots,k$. The paths q_i are determined by a right-first search. Let $v \in V$, and let e be incident to v. We will say that the *next free edge after* e with respect to v is the first free edge to follow e in the adjacency list of v in counterclockwise ordering.

Procedure 5.3.

```
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 not a terminal do
      let $\{v,w\}$ be the next free edge after the leading edge of $q_i$ with respect
      to $v$;
      add $\{v,w\}$ to $q_i$;
      $v$:=$w$;
  if $v \neq t_i^{()}$ then stop: return "unsolvable";
return $(q_1,\dots,q_k)$;
```

The *auxiliary paths* $q_1,\dots,q_k$ yield a directed *auxiliary graph* $A(G,N,x)$ of instance (G,N) with respect to start terminal x. Just orient all edges on the paths $q_1,\dots,q_k$ according to the direction in which they are traversed during the procedure. Then $A(G,N,x)$ consists of all vertices of G and of all oriented edges. The solution $p_1,\dots,p_k$ for the original instance (G,N) is now determined in the auxiliary graph. That is, edges that are not contained in the auxiliary graph will not be occupied by a path $p_1,\dots,p_k$ 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 $A(G,N,x)$ are used according to their orientations in $A(G,N,x)$.

Algorithm 5.4.

```
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 $p_i$ with
        respect to $v$;
        add $(v,w)$ to $p_i$;
        $v{:=}w$;
    if $v \neq t_i$ then stop: return "unsolvable";
return $(p_1,\dots,p_k)$;
```

Algorithm 5.4 can be implemented to run in linear time using a special case of Union-Find [4]. For a proof of correctness see [26]. We will focus on the following optimization problem.

Problem 5.5. Minimum Edge-Disjoint Paths Problem.

Instance: *An optimization instance* (G,N).

Problem: *Find k edge-disjoint paths* $p_1,\dots,p_k$ *in G connecting* s_i *and* t_i, *for* $1 \leqslant i \leqslant k$, *such that* $\sum_{i=1}^{k}$ *length*(p_i) *is minimum.* (*The length of a path p is the number of edges on p.*)

5.2. $\mathscr{NP}$-completeness proof

We prove that the decision problem corresponding to Problem 5.5 is $\mathscr{NP}$-complete, even if the maximum degree of the graph is four.

Problem 5.6. Minimum Edge-Disjoint Paths Decision Problem.

Instance: *A decision instance* (G,N,K).

Question: *Are there edge-disjoint paths* $p_1,\dots,p_k$ *in G connecting* s_i *and* t_i, *for* $1 \leqslant i \leqslant k$, *such that* $\sum_{i=1}^{k}$ *length*$(p_i) \leqslant K$?

Theorem 5.7. *The minimum edge-disjoint paths decision problem stated as Problem* 5.6 *is* $\mathscr{NP}$ *-complete, even if the maximum degree of G is four.*

Proof. It is easy to see that Problem 5.6 is in $\mathscr{NP}$. For the $\mathscr{NP}$-completeness proof we use a transformation from $3SAT$ [6]. An instance of $3SAT$ is given by a set U of variables, $|U| = n$, and a set C of clauses over U, $|C| = m$, such that $|c| = 3$ for $c \in C$. The problem is to decide if there is a satisfying truth assignment for C. We

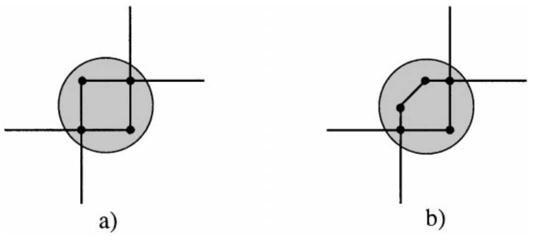

Fig. 1. (a) Super-vertex of type 1. (b) Super-vertex of type 0.

can assume without loss of generality that n is an even number (otherwise, we can obviously add a “dummy variable”).

Let $K{:=}n(9m+1)+m(6n+1)$. We construct an even instance (G,N) consisting of a planar graph G with maximum degree four, and a set of nets whose terminals lie on the boundary of the outer face of G. G is a grid-like graph consisting of $2n$ horizontal lines $i_1, i_2, i=1,\dots,n$ and $3m$ vertical lines $j_1, j_2, j_3,\ j=1,\dots,m$. An horizontal line $i_l,\ l \in \{1,2\}$ and a vertical line $j_r,\ r \in \{1,2,3\}$ meet in a “super-vertex” v_{i_l,j_r} of *type* 1 or *type* 0. A super-vertex of *type* 1 consists of four vertices and a super-vertex of *type* 0 consists of five vertices. See Fig. 1. Variable $u_i \in U$ corresponds to two subsequent horizontal lines i_1, i_2, where u_i corresponds to i_1 and u_i corresponds to i_2. The clause $c_j \in C$ corresponds to the vertical lines j_1, j_2, j_3, where j_r corresponds to the rth literal in c_j. For every variable $u_i \in U$ and every clause $c_j \in C$ we have a net $\{s_{u_i}, t_{u_i}\}$ and a net $\{s_{c_j}, t_{c_j}\}$, respectively. Terminal s_{u_i} lies on a vertex connected to the two leftmost super-vertices on horizontal lines i_1 and i_2, while terminal t_{u_i} lies on a vertex connected to the two rightmost super-vertices on horizontal lines i_1 and i_2. Analogously, terminal s_{c_j} lies on a vertex connected to the three uppermost super-vertices on vertical lines j_1, j_2 and j_3, and terminal t_{c_j} lies on a vertex connected to the three lowermost super-vertices on vertical lines j_1, j_2 and j_3. In order to guarantee that the instance is Eulerian, vertices corresponding to s_{u_i} (resp. t_{u_i}) are pairwise connected by an edge, i.e., edges $\{s_{u_i}, s_{u_{i+1}}\}$ and $\{t_{u_i}, t_{u_{i+1}}\}$ are added for $i=1,\dots,k-1$. Observe that a shortest path connecting s_{u_i} and t_{u_i} (resp. s_{c_j} and t_{c_j}) has length $9m+1$ $(6n+1)$. See Fig. 2.

For super-vertex v_{i_l,j_r}, where horizontal line $i_l,\ l \in \{1,2\}$ and vertical line $j_r,\ r \in \{1,2,3\}$ meet, we have

$$type(v_{i_l,j_r}){:=}\begin{cases} 0 & \text{if } l=1 \text{ and } \neg u_i \text{ occurs in clause } c_j \text{ as } r\text{th literal} \\ & \text{or } l=2 \text{ and } u_i \text{ occurs in clause } c_j \text{ as } r\text{th literal;} \\ 1 & \text{otherwise.} \end{cases}$$

By definition, a super-vertex v_{i_l,j_r} is of type 0 if and only if setting the corresponding literal $\neg u_i$ (resp. u_i) to false does not satisfy the corresponding clause c_j. Obviously, the two paths connecting s_{u_i} and t_{u_i} (resp. s_{c_j} and t_{c_j}) can be both shortest only if they meet in a super-vertex of type 1. See Fig. 3 for an example.

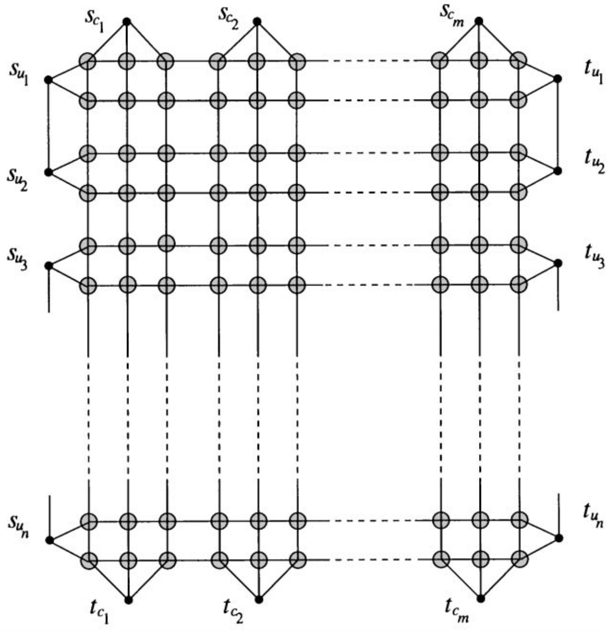

Fig. 2. Generic example of an instance of Problem 5.6 corresponding to an instance of 3*SAT*.

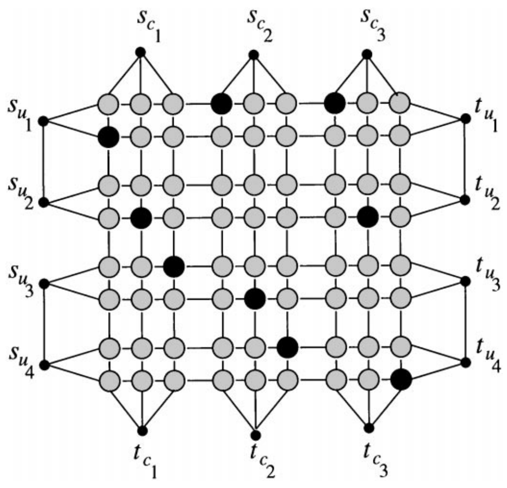

Fig. 3. The instance of Problem 5.6 corresponding to the clauses $c_1 = u_1 \vee u_2 \vee \neg u_3, c_2 = \neg u_1 \vee u_3 \vee \neg u_4, c_3 = \neg u_1 \vee u_2 \vee u_4$. The shaded super-vertices are of type 1, the black super-vertices of type 0.

Now, a satisfying truth assignment for an instance of 3*SAT* induces a solution to the corresponding instance of Problem 5.6 as follows. Net $\{s_{u_i}, t_{u_i}\}$ is routed along horizontal line i_1 if and only if u_i is set *true*. Net $\{s_{c_j}, t_{c_j}\}$ is routed along the leftmost vertical line j_r, $r \in \{1,2,3\}$ corresponding to a literal satisfying c_j. Then nets $\{s_{u_i}, t_{u_i}\}$

and $\{s_{c_j}, t_{c_j}\}$ meet only in super-vertices of type 1. Consequently, the total length of the solution is K.

On the other hand, in a solution of length at most K to the instance of Problem 5.6 any net $\{s_{u_i}, t_{u_i}\}$ has length $9m+1$ and any net $\{s_{c_j}, t_{c_j}\}$ has length $6n+1$. Therefore, nets only meet at super-vertices of type 1. This induces a truth assignment for the instance of $3SAT$ where u_i is set true if and only if $\{s_{u_i}, t_{u_i}\}$ is routed along horizontal line i_1. If for a clause c_j the corresponding net is routed along vertical line j_r, then the net corresponding to the rth literal of c_j passes j_r through a super-vertex of type 1. Consequently, this literal must be true, and c_j is satisfied. □

Problem 5.8. Minimum Longest Path Problem.

Instance: *A decision instance* (G,N,K).

Question: *Are there edge-disjoint paths* $p_1, \dots, p_k$ *in* G *connecting* s_i *and* t_i, *for* $1 \leqslant i \leqslant k$, *such that the length of the longest path* p_i *is at most* K?

Corollary 5.9. *Problem* 5.8 *is* $\mathscr{NP}$*-complete, even if the maximum degree of* G *is four.*

Proof. We use a slight modification of the reduction from Theorem 5.7. The instance for $3SAT$ is modified by adding "dummy variables" and "dummy clauses", such that for the corresponding graph G the number of vertical lines is equal to the number of horizontal lines, and K is set to the shortest length of a path connecting the two terminals of a net. Then in a set of paths corresponding to a satisfying truth assignment every path has length K. □

5.3. Heuristics

In this section, we describe several heuristics for Problem 5.5 that are based on Algorithm 5.4. We present an extensive experimental study comparing these heuristics and an exact method on more than 1800 instances. However, the exact method has exponential running time in the worst–case. For larger instances (more than 100 vertices and 20 nets) this method delivered no solution in more that 90% of the instances.

In principle, the heuristics pursue three different ideas. An obvious way to improve Algorithm 5.4 heuristically is to choose the start terminal x best possible. Second, we can use the fact that only edges occupied by the auxiliary graph determined in Procedure 5.3 belong to the final solution. Third, shortest paths computations may be included into the computation of the edge-disjoint paths. Observe that in general a collection of shortest paths connecting the terminals of the nets are not a feasible solution, because they are not pairwise edge-disjoint.

The first three heuristics use a sort of preprocessing for Algorithm 5.4 where the start terminal is chosen heuristically. Then the basic algorithm is called. The crucial fact used by the heuristics is that the $2k$-string of s-terminals and t-terminals on the outer face in counterclockwise ordering can be shifted cyclically without influencing the solvability of the instance. Possibly, s_i has to be exchanged with t_i to maintain the property that s_i occurs before t_i in the string. Now, an interval of length l_i is associated with every net n_i, and the list is shifted cyclically until the total interval length is minimal. After that, Algorithm 5.4 is called with the first terminal of the $2k$-string as the start terminal x. These heuristic algorithms also have linear running time.

Heuristic 5.10. Edge-Disjoint Min Interval Path Algorithm I. *The interval length l_i is defined as the number of terminals between s_i and t_i in the sequence.*

Heuristic 5.11. Edge-Disjoint Min Interval Path Algorithm II. *The length l_i is defined as the number of edges on the outer face between s_i and t_i.*

Heuristic 5.12. Edge-Disjoint Min Interval Path Algorithm III. *The length l_i is defined as the number of edges on the outer face between s_i and t_i minus the edges incident to the terminals s_i and t_i.*

The next three heuristics also use a sort of preprocessing for Algorithm 5.4 where the start terminal is chosen heuristically. These heuristics start again at the ordering of the s- and t-terminals, in the first phase of the basic algorithm. But here the nets of the problem with parenthesis structure are considered. An interval length l_i is associated with every net $n_i^{()}$, $i \in \{1 \dots k\}$ of instance $(G, \mathcal{N}^{()})$. For the definition of l_i see below. Then, the $2k$-string of terminals with minimum total interval length is determined and Algorithm 5.4 is called with the first terminal as start terminal of that string. The running time is $\mathcal{O}(n + k^2)$.

Heuristic 5.13. Edge-Disjoint Min Parenthesis Interval Path Algorithm I. *The interval length l_i is defined as the number of terminals between $s_i^{()}$ and $t_i^{()}$ in the sequence.*

Heuristic 5.14. Edge-Disjoint Min Parenthesis Interval Path Algorithm II. *The length l_i is defined as the number of edges on the outer face between $s_i^{()}$ and $t_i^{()}$.*

Heuristic 5.15. Edge-Disjoint Min Parenthesis Interval Path Algorithm III. *The length l_i is defined as the number of edges on the outer face between $s_i^{()}$ and $t_i^{()}$ minus the number of edges incident to the terminals s_i and t_i.*

The next two heuristics use the following observation. In the second phase of Algorithm 5.4 the paths are constructed using only edges from the auxiliary graph. The running time is linear respectively $\mathcal{O}(nk)$.

Heuristic 5.16. Reduced Edge-Disjoint Path Algorithm. *The heuristic now uses the observation above by invoking Algorithm* 5.4 *first and then removing all edges, which are not considered. Then, a new start terminal is chosen randomly and the algorithm is called again with this reduced instance.*

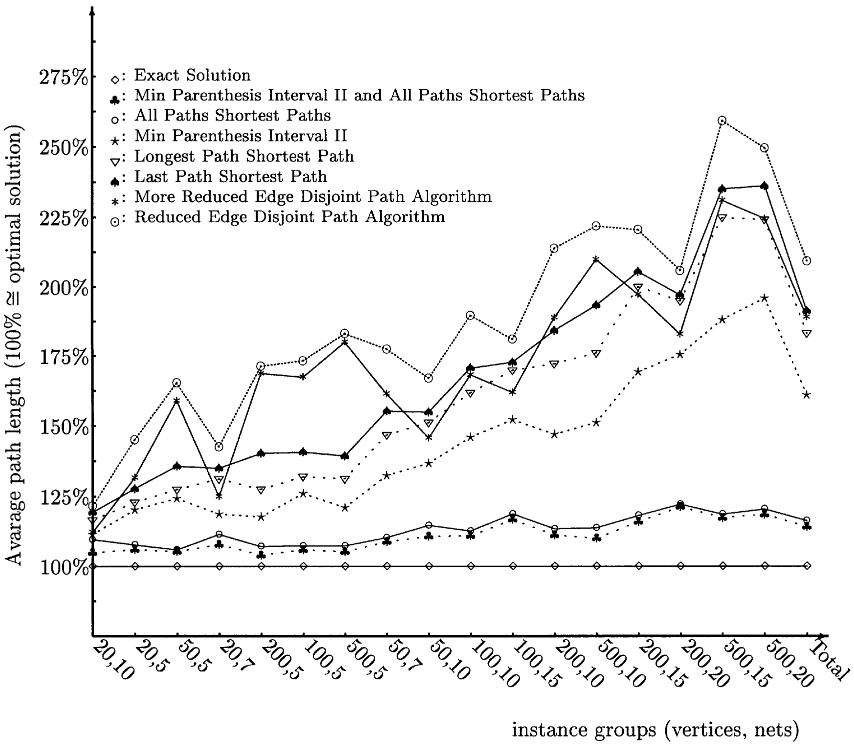

Fig. 4. Representation of the avarage difference between exact and heuritic solutions.

Heuristic 5.17. More Reduced Edge-Disjoint Path Algorithm. *Heuristic* 5.16 *is applied for every terminal as the start terminal in the second call of the basic algorithm. This is done in the heuristic algorithm and the resulting solution is shown.*

The following heuristics use a sort of postprocessing for the basic algorithm. Algorithm 5.4 is called first and then the constructed paths are reconsidered. The running time is linear respectively $\mathcal{O}(nk)$.

Heuristic 5.18. Edge-Disjoint Path Algorithm – Last Path Shortest Path. *The aim of this heuristic algorithm is to shorten the length of the last path of the solution determined by Algorithm* 5.4. *Therefore, the last path is determined by an algorithm to compute shortest paths using only edges of the input instance which are not occupied by the other paths.*

Heuristic 5.19. Edge-Disjoint Path Algorithm – Longest Path Shortest Path. *Analogously, this heuristic algorithm shortens the longest path of the solution determined by Algorithm* 5.4. *The longest path is determined by an algorithm to compute*

Table 1
The average path length of the heuristics in percent in respect to the exact solutions

Vertices, nets	Exact results	Min Parenthesis Interval II and All Paths Shortest	All Paths Shortest Paths	Min Paranthesis Interval II	Min Paranthesis Interval I	Min Interval II	Min Interval I	Min Paranthesis Interval III	Min Interval III	Longest Path Shortest Path	More Reduced	Last Path Shortest Path	Reduced
20,5	100.00	106.1	107.8	120.4	122.2	120.4	122.6	121.3	120.9	123.0	131.7	127.8	145.2
20,7	100.00	107.9	111.5	118.8	119.7	118.8	120.3	121.2	120.3	131.2	125.0	135.0	142.6
20,10	100.00	105.0	109.7	111.5	111.3	111.5	111.9	112.1	113.3	116.7	112.3	119.5	121.7
50,5	100.00	105.3	106.0	124.4	124.4	124.0	125.1	124.7	126.9	127.6	159.0	135.7	165.4
50,7	100.00	108.9	110.3	132.4	133.1	132.6	133.6	132.9	132.6	146.8	161.4	155.2	177.5
50,10	100.00	110.8	114.7	136.7	137.3	136.7	138.1	137.0	137.2	151.2	145.7	154.9	167.0
100,5	100.00	105.9	107.4	126.1	128.5	125.2	129.1	125.5	125.5	132.0	167.4	140.7	173.3
100,10	100.00	111.0	112.7	146.0	146.0	146.5	146.9	145.9	145.8	161.8	168.2	170.7	189.6
100,15	100.00	116.8	118.8	152.2	152.3	154.2	152.7	155.0	156.4	169.8	161.9	172.7	181.0
200,5	100.00	104.0	107.2	117.8	121.8	116.4	122.5	118.0	118.3	127.6	168.7	140.3	171.4
200,10	100.00	111.2	113.4	147.0	150.3	147.4	148.8	150.5	148.8	172.2	188.8	184.2	213.8
200,15	100.00	116.0	118.2	169.3	170.3	169.9	170.2	171.0	171.2	200.0	197.2	205.4	220.5
200,20	100.00[a]	121.3	122.2	175.5	175.5	175.5	176.1	177.1	177.1	194.8	182.8	197.0	205.8
500,5	100.00	105.2	107.4	121.1	127.2	121.6	123.4	120.2	121.1	131.2	180.0	139.3	183.1
500,10	100.00	110.1	113.8	151.2	151.8	151.7	151.8	152.9	153.6	176.0	209.8	193.3	221.8
500,15	100.00	117.4	118.7	188.1	187.0	189.8	190.0	190.9	192.1	224.9	231.0	235.1	259.3
500,20	100.00[b]	118.6	120.5	195.8	195.8	197.9	197.3	198.1	198.5	224.1	224.5	236.2	249.6
Total	100.00	114.2	116.3	161.0	161.6	161.8	162.4	162.7	163.1	183.1	189.0	191.2	209.2

[a] Only 45% of all solvable instances were solved.
[b] Only 14% of all solvable instances were solved.

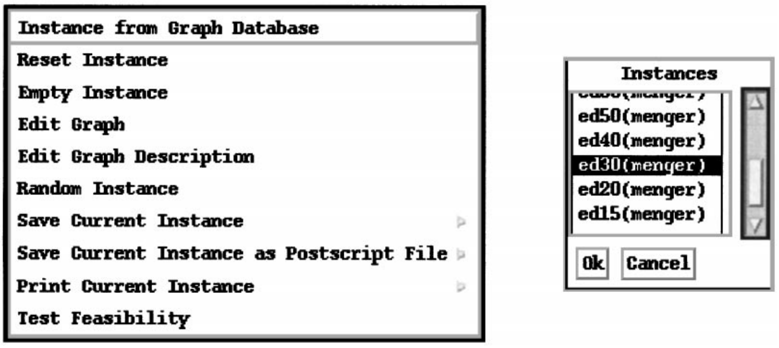

Fig. 5. The main menu for handling instances and the listing of all instances of type "vertex-disjoint Menger problem".

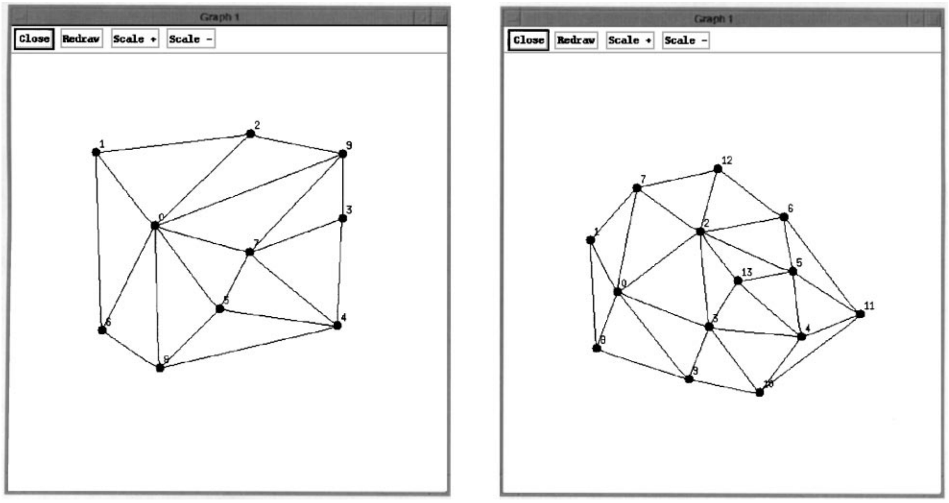

Fig. 6. Small randomly generated, triangulated planar graphs.

shortest paths using only edges of the input instance which are not occupied by the other paths.

Heuristic 5.20. Edge-Disjoint Path Algorithm – All Paths Shortest Paths. *This method uses a sort of postprocessing for the basic algorithm. Algorithm* 5.4 *is called first and then the constructed paths are reconsidered one after the other. Let* $p_1, \ldots, p_k$ *be the constructed paths. For* p_k *to* p_1 *all paths are removed from the graph and redetermined by an algorithm which determines shortest paths using only edges of the input instance which are not occupied by the other paths.*

The last heuristic is a combination of two of the previously described methods. Its running time is $\mathcal{O}(nk)$.

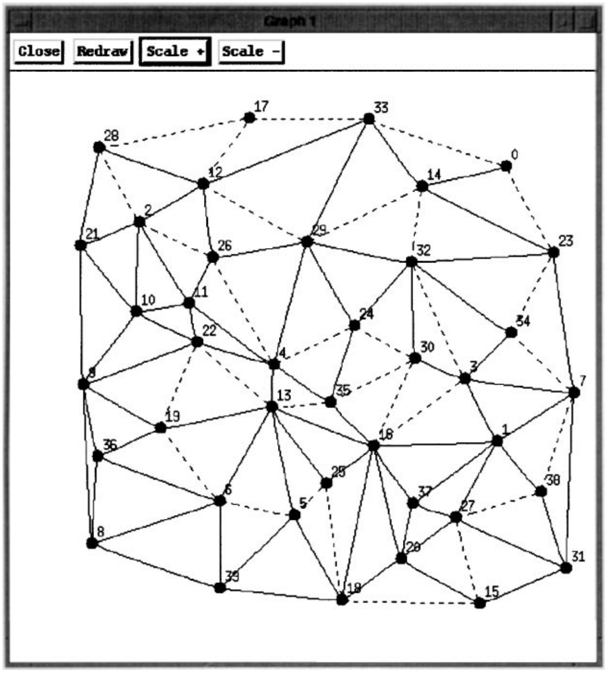

Fig. 7. An instance of type “vertex-disjoint Menger problem”, with source 30 and target 17, and the solution constructed by the algorithm from [18,20]. In black-and-white mode, the vertex-disjoint paths from the source to the target are dashed.

Heuristic 5.21. Min Parenthesis Interval II and All Paths Shortest Paths. *This method calls as a preprocessing Heuristic* 5.14. *As a postprocessing Heuristic* 5.20 *is executed.*

The heuristics have been implemented and included in PlaNet. They were tested on randomly generated instances. The random instance generator applies the following two steps. First a triangulation is constructed. (Recall that the triangulations are exactly the maximal planar graphs with respect to insertion of edges inside the outer face.) Afterwards, a couple of edges, paths, and cycles is removed. The Eulerian condition is guaranteed by removing a couple of edge-disjoint paths from the triangulation, namely such that the number of paths meeting a vertex is even if and only if this vertex already fulfills the Eulerian condition in the initial triangulation. Our experience is that suitable random distributions can be implemented this way much more easily than, for example, by constructing random graphs incrementally. In contrast, an incremental approach causes a lot of difficult technical problems (e.g., maintenance of planarity throughout the procedure).

We have studied the heuristics on more than 1800 instances in total. The results of our experimental studies are shown in Fig. 4 and Table 1. We tested 100 instances of each pair (n,k) in Table 1, where n is the number of vertices of the graph and k is the number of nets. The heuristics are compared to an exact approach [15] that is

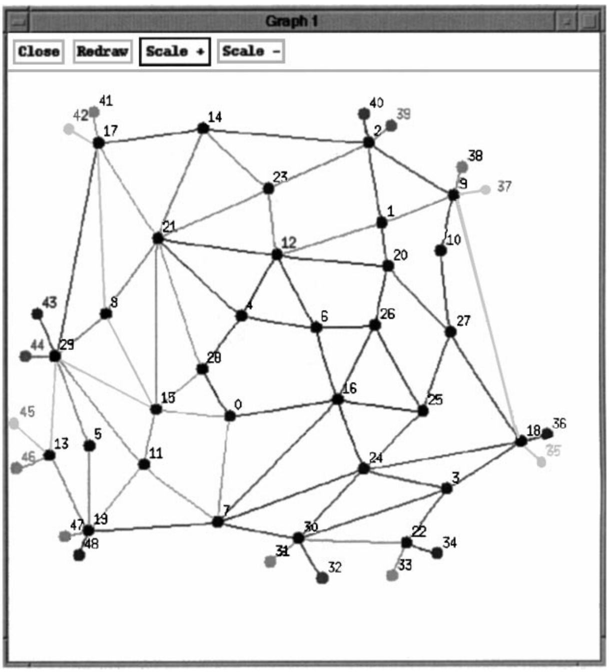

Fig. 8. An instance of type “Okamura–Seymour problem”, i.e., all terminals are on the boundary, and for each vertex the degree plus the number of teminals placed on it sum up to an even number. The dashed lines are the edges occupied by the algorithm from [24,26] (drawn in different colors in color mode). The degree-one nodes on the boundary are auxiliary nodes and have been added by the algorithm. Each terminal has been moved by the algorithm to one of these auxiliary nodes.

based on branch-and-bound. Of course, the exact method has exponential running time in the worst-case.

Acknowledgements

We thank Dagmar Handke for proof-reading this paper and for valuable comments.

References

[1] M. Becker, K. Mehlhorn, Algorithms for routing in planar graphs, Acta Inform. 23 (1986) 163–176.
[2] H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer, Berlin, 1987.
[3] M. Formann, D. Wagner, F. Wagner, Routing through a dense channel with minimum total wire length, J. Algorithms 15 (1993) 267–283.
[4] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1985) 209–221.
[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, Reading, MA, 1995.
[6] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness, Freeman, San Francisco, 1979.

[7] L. Guibas, J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams, ACM Trans. Graphics 4 (1985) 75–123.
[8] D. Handke, G. Neyer, PlatNet Tutorial and Reference Manual, 1996.
[9] M. Kaufmann, G. Klär, A faster algorithm for edge-disjoint paths in planar graphs, in: W.-L. Hsu, R. Lee (Eds.), ISA'91 Algorithms, 2nd Internat. Symp. on Algorithms, Lecture Notes in Computer Science, vol. 557, Springer, Berlin, 1991, pp. 336–348.
[10] K. Matsumoto, T. Nishizeki, N. Saito, An efficient algorithm for finding multicommodity flows in planar networks, SIAM J. Comput. 14 (1985) 289–302.
[11] K. Mehlhorn, S. Näher, LEDA: a library of efficient data structures and algorithms, Comm. ACM 38 (1995) 96–102.
[12] K. Mehlhorn, S. Näher, C. Uhrig, The LEDA User Manual, Version r 3.4, 1996.
[13] B. Meyer, Object-oriented Software Construction, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[14] G. Neyer, Optimierung von Wegpackungen in planaren Graphen, Master's thesis, 1996.
[15] M. Oellrich, Master's thesis, 1996 (in German).
[16] H. Okamura, P. Seymour, Multicommodity flows in planar graphs, J. Combin. Theory Ser. B 31 (1981) 75–81.
[17] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Springer, Berlin, 1993.
[18] H. Ripphausen-Lipa, D. Wagner, K. Weihe, The vertex-disjoint Menger-problem in planar graphs, in: Proc. 4th Annual ACM–SIAM Symp. on Discrete Algorithms, SODA'93 1993, pp. 112–119.
[19] H. Rippahausen-Lipa, D. Wagner, K. Weihe, Efficient algorithms for disjoint paths in planar graphs, in: W. Cook, L. Lovász, P. Seymour (Eds.), DIMACS Series in Discrete Mathematics and Computer Science, vol. 20, American Mathematical Society, Providence, RI, 1995, pp. 295–354.
[20] H. Ripphausen-Lipa, D. Wagner, K. Weihe, The vertex-disjoint Menger problem in planar graphs, SIAM J. Comput. 26 (1997) 331–349.
[21] B. Stroustrup, The C++ Programming Language, 2nd ed., Addison-Wesley, Reading, MA, 1994.
[22] D. Wagner, Optimal routing through dense channels, CWI Quarterly 3 (1993) 269–289.
[23] D. Wagner, K. Weihe, An animated library for combinatorial VLSI routing algorithms, Technical report, Fachbereich Mathematik, Technische Universität Berlin, 1993. Full version from URL http://informatik.unikonstanz.de/w̃eihe/manuscripts.html#paper10.
[24] D. Wagner, K. Weihe, A linear time algorithm for edge-disjoint paths in planar graphs, in: T. Lengauer (Ed.), 1st European Symp. on Algorithms, ESA'93, Lecture Notes in Computer Science, vol. 726, Springer, Berlin, 1993, pp. 384–395.
[25] D. Wagner, K. Weihe, An animated library for combinatorial VLSI routing (communications), in: Proc. 11th ACM Symp. on Computational Geometry, SCG'95, Vancouver, British Columbia, Canada, June 5–7, 1995.
[26] D. Wagner, K. Weihe, A linear time algorithm for edge-disjoint paths in planar graphs, Combinatorica 15 (1995) 135–150.
[27] F. Wagner, B. Wolfers, Short wire routing in convex grids, in: W.-L. Hsu, R. Lee (Eds.), ISA'91 Algorithms, 2nd Internat. Symp. on Algorithms, Lecture Notes in Computer Science, vol. 557, Springer, Berlin, 1991, pp. 72–83.
[28] K. Weihe, Edge-disjoint (s,t)-paths in undirected planar graphs in linear time, in: J. Leeuwen (Ed.), 2nd European Symp. on Algorithms, ESA'94, Lecture Notes in Computer Science, vol. 855, Springer, Berlin, 1994, pp. 130–140.
[29] K. Weihe, Generische Programmierung: Polymorphie jensiets von isolierten Objekten, Konstanzer Schriften in Mathematik und Informatik 54, Universität Konstanz, 1998.