Using Network Flows for Surface Modeling

by

Rolf H. Möhring
Matthias Müller–Hannemann
Karsten Weihe

No. 377/1994 Revised

Using Network Flows for Surface Modeling[§]

Rolf H. Möhring[*] Matthias Müller–Hannemann[*] Karsten Weihe[**]

Technische Universität Berlin

Revised version, December 1994

Abstract

We apply network flow techniques to a problem arising in the computer aided design of cars, planes, ships, trains, and the like: Refine a mesh of spheric polygons, which approximates the surface of a workpiece, so that the resulting mesh is suitable for a numerical analysis. In commercial CAD systems, this problem is usually solved in a greedy–like fashion, which leaves the user a lot of patchwork to do afterwards. The global approach introduced here avoids those local traps and yields solutions that require significantly less additional patchwork.

1 Introduction

In this paper, we consider a problem arising in the computer aided design of pieces of traffic vehicles, for example, bodyworks or chassis or components of them. This work was done in cooperation with a Berlin software house, the *Dr. Krause Software GmbH*, which sells a CAD preprocessor, ISAGEN, exactly for this problem [Kr]. Our part of this cooperation is to solve that particular problem and to implement code that can be inserted in ISAGEN.

In principle, computer aided design proceeds in several stages. In the beginning, the user, who is usually an expert in the application domain, interactively designs a model of the workpiece he has in mind. This model is formulated as a two–dimensional mesh of spheric, convex, openly disjoint polygons in the three–dimensional space. Typically, the polygons are triangles and quadrilaterals. See Figs. 7, 5, and 11 for examples taken from practice and Fig. 1 for an artificial instance, which might be more instructive.

For convenience, we will always identify a mesh like this with the undirected graph $G = (V, E)$ formed by the boundaries of the polygons. That is, V is the set of all vertices of polygons, and E consists of all subintervals of boundaries between two elements of V. In other words, an edge is an inclusion–maximal subinterval such that all internal points of that interval are shared by the same polygons. The polygons themselves are henceforth called the *holes* of G. Note that, depending on the shape of the workpiece, an edge may be incident to more than two holes. Those edges are called *folding edges*.

Once the mesh has been specified by the user, the next step is to apply a numerical analysis in order to learn about the physical properties of the workpiece. This includes, for example, vibration and acoustic characteristics, heat transfer, stability, and elasticity. The numerical analysis usually amounts to solving a system of partial differential equations by means of finite element methods.

This provides information about how to modify the input workpiece (i.e., the mesh) in order to improve its physical properties. After the user has modified the mesh accordingly, he may run another

[§]An extended abstract has appeared in the Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (1995).

[*]Technische Universität Berlin, Fachbereich Mathematik, Sekr. MA 6-1, Straße des 17. Juni 136, 10623 Berlin, Germany, e-mail: {moehring,mhannema}@math.tu-berlin.de;

[**]Universität Konstanz, Informatik, 78434 Konstanz, Germany, e-mail: karsten.weihe@uni-konstanz.de; communicate with the second author.

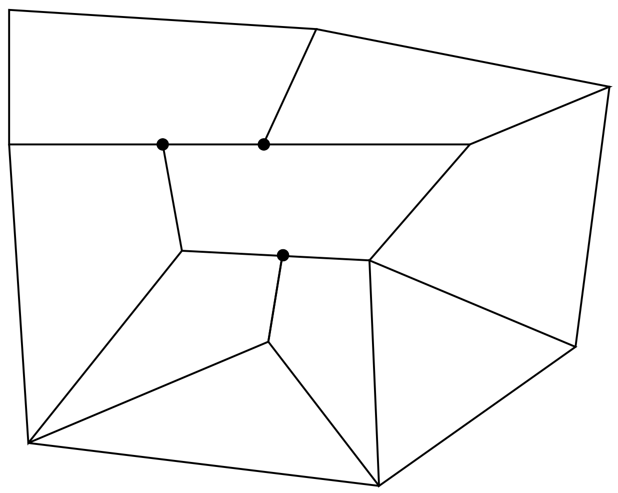

Figure 1: An artificial instance.

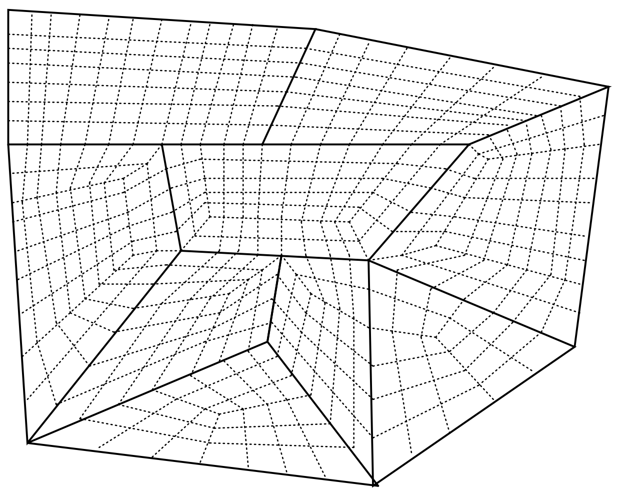

Figure 2: A feasible refinement: reasonably fine, quadrilaterals only, and each side of each polygon consists of a single edge (conforming mesh).

numerical analysis, which provides new insights, and so on. The whole process is repeated until the physical characteristics of the workpiece satisfy all requirements.

However, interactively generated meshes are, in general, not suitable for finite element methods, for three reasons.

1. Those meshes are not fine enough for the numerical analysis to achieve the required accuracy.
2. For numerical reasons, our partners regard triangles in a mesh highly unfavorable and prefer meshes consisting only of quadrilaterals. However, a CAD system must not leave this problem to the user, since modeling a workpiece with quadrilaterals only is difficult.
3. The mesh must be *conforming.* That is, any two polygons share either a whole side or a single vertex or are completely disjoint. In other words, each side of a polygon must consist of a single edge of G. This task cannot be left to the user either.

For these reasons, the CAD system must first refine the mesh according to these requirements, before the numerical analysis itself may take place. Figs. 2, 8, 6, and 12 show feasible refinements for the meshes in Figs. 1, 7, 5, and 11, respectively. This must be done automatically since refining a mesh by hand takes a lot of time and is prone to error.

Requirement 3 implies that the individual polygons must not be refined independently of each other, which could be done if only requirements 1 and 2 had to be satisfied. In fact, the refinement of one single polygon may restrict the possibilities for many other polygons, possibly for all. Hence, the refinements for the individual polygons have to fit together. More precisely, requirement 3 means that for each edge of the input mesh, all refinements of the incident polygons induce the same collection of additional mesh vertices to lie on, and subdivide, that edge.

This suggests a two–stage approach: First determine the additional mesh vertices located on the different edges of the input mesh. Then refine each polygon separately so that, for each incident edge, the induced additional mesh vertices are exactly the vertices determined in the first stage. Most systems pursue exactly this strategy, and so does ISAGEN. However, when doing so, one is faced with another problem yet: It does not suffice that the first stage provides a result induced by a feasible refinement with reasonable numerical properties; it is also necessary that the second stage be able to find one.

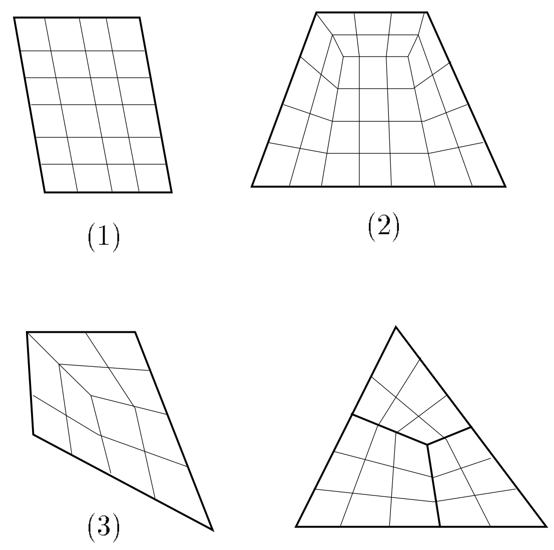

Figure 3: The standard templates for quadrilaterals and triangles. The central 'wye' of the triangle is emphasized.

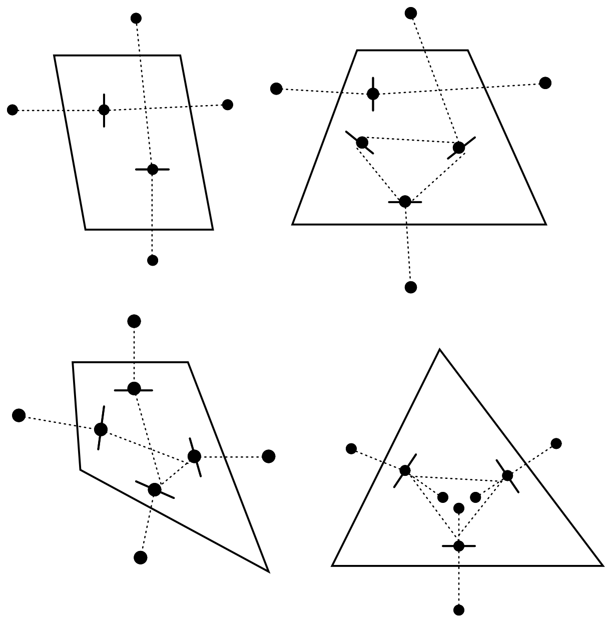

Figure 4: The auxiliary network $\tilde{G} = (\tilde{V}, \tilde{E})$ for a one–hole instance, induced by the corresponding template, respectively. The short solid line going through a transshipment vertex indicates the partition of its adjacency list into two parts.

For this purpose, each system provides a set of *templates* for refining a single polygon. The templates are designed so that each polygon is refined into quadrilaterals only. Fig. 3 shows the standard templates for triangles and quadrilaterals.

A quadrilateral can be refined according to Template 1 if and only if $N_1 = N_2$ and $N_3 = N_4$, where N_1 and N_2 denote the final numbers of mesh vertices on two opposite sides, and N_3 and N_4, on the other two sides. The second template (Template 2) means $N_1 = N_2$ and $N_3 \equiv N_4 \pmod 2$ or $N_3 = N_4$ and $N_1 \equiv N_2 \pmod 2$, and the third template (Template 3) means $|N_1 - N_2| = |N_3 - N_4|$.

Finally, a triangle can be refined to the standard template if and only if $N_1 = a+b+1$, $N_2 = a+c+1$, and $N_3 = b + c + 1$ for some nonnegative integers a, b, c, where N_1, N_2, and N_3 denote the numbers of final mesh vertices on the three sides.

The problem in the first stage is now to find subdivisions of all edges of the input mesh such that at least one template applies to each hole, respectively. The second stage, that is, refining the individual polygons separately according to these templates, is now a relatively easy task, and satisfactory algorithms have already been designed for that. It is the first stage that makes the problem hard, and exactly this is the problem we will consider in this paper.

Problem 1.1 (Preliminary formulation) *Given an input mesh consisting of triangles and quadrilaterals, and the set of templates shown in Fig. 3, find a subdivision of all edges such that each polygon can be refined by one of the templates, the resulting refinement is conforming and everywhere reasonably fine (to be specified later). In addition, numerical aspects, e.g. avoidance of pathological angles, induce an objective function which should be minimized by the solution (to be specified later).*

Little is known in general about this problem. Both scientific and commercial systems are based on heuristics. See [Ho, SNTM] for a survey of the whole domain. All these heuristics proceed as follows (to the best of our knowledge). All polygons are refined one at a time, where first the additional mesh vertices on the boundary are fixed, and then additional lines are drawn through the interior of the

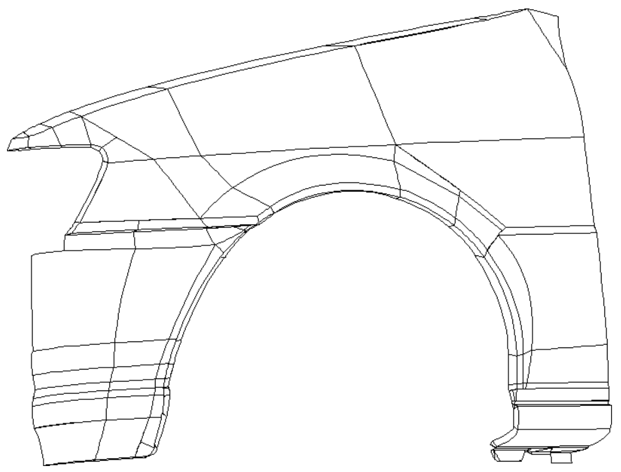

Figure 5: The left front wing of a car.

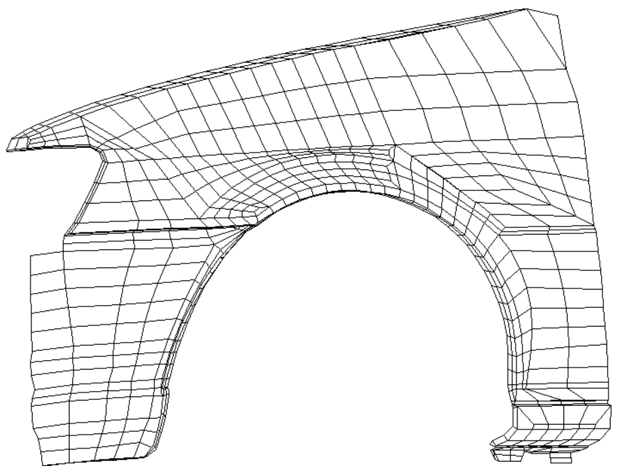

Figure 6: The refinement produced by our algorithm for the mesh in Fig. 5.

polygon. In this process, the (increasing) restrictions imposed by previously refined polygons have to be incorporated.

Quite often, a heuristic like this runs into a situation where the additional restrictions do not leave any possibility to refine the remaining polygons, although the input instance itself may be solvable. Some of the heuristics get stuck in this case, and other heuristics modify some of the previous refinements in order to resolve the conflict. Furthermore, some heuristics apply yet another strategy, namely, early enough before a conflict may occur, they apply some additional "emergency" templates to polygons, which have worse numerical properties, but do not impose further restrictions on the refinements of the remaining polygons.

However, when applying such an algorithm to an instance taken from practice, there typically remain polygons unrefined (or unacceptably badly refined). As a consequence, there still remains a lot of patchwork to do for the user. This usually takes very much time, because the (correct) refinements of other polygons must be changed, too, in order to make refinements of the unrefined polygons possible.

In retrospect, this is not surprising at all, since we have shown that the problem is highly intractable. All one can hope for is a heuristic which solves most relevant instances satisfactorily, and otherwise, leaves only little additional patchwork to do. Fortunately, this is not beyond all hope, because the instances used to prove intractability are highly pathological. In fact, instances from practice seem to be much easier and much better accessible in general.

In contrast to the greedy–like algorithms outlined above, we have developed a somewhat more *global* approach. This means that we do *not* refine one polygon at a time completely. Instead, we repeatedly perform partial refinements, one at a time, which may affect several polygons simultaneously, but do not necessarily complete the refinements of these polygons. More precisely, each partial refinement step affects a chain of polygons, where any two subsequent polygons coincide (repetitions allowed). Such a step does not necessarily make the refinement of a polygon finer; the effect may be a correction of the current partial refinement of that polygon. Therefore, no decision is fixed once and for all, although we do not apply expensive backtrack or "emergency" operations.

This basic idea leads to subproblems which can be stated as a certain generalization of the *feasible circulation problem* on undirected graphs and which are equivalent to feasible flow problems in bidirected graphs. See [AMO] for a survey of network flows and, in particular, for a discussion of the feasible circulation problem. In a sense, the notion of *flow augmenting path* can be generalized to our problem, and the task to refine the mesh step by step along a chain of polygons amounts to augmenting the flow along such a generalized augmenting path.

The bidirected flow problem can be reduced to a (capacitated) perfect b-matching problem [Ed, De].

Although this transformation is certainly interesting from the theoretical point of view, it does not seem appropriate for a practical work such as ours, for several reasons. Firstly, it necessarily involves complicated data structures and algorithmic techniques, which makes the maintenance hard for our partners. Secondly, a good asymptotic complexity does not guarantee a good practical performance. Thirdly, the direct solution allows a better use and understanding of some heuristics for the main problem.

Therefore, we decided to avoid treatment of blossoms and to implement an algorithm which is merely a heuristic, but is easy to implement, to maintain and to adapt, and, nonetheless, yields very good practical results.

This is not the first approach to this problem which goes beyond the usual greedy–like strategies. Tam and Armstrong have formulated a similar problem as an integer linear program and solved this using an improvement of Gomory's algorithm [TA]. However, there are several drawbacks to their approach, which are overcome by our method. Firstly, practical instances often consist of hundreds or thousands of polygons, and the resulting integer linear programs are quite large. The time needed to solve such a program might be unreasonable for an interactive system. Secondly, it is not possible to construct a suboptimal solution by stopping the program deliberately after some time, since no intermediate solution is feasible. Thirdly, it is not clear at all how to select a good template for each polygon. The authors do not consider this problem in detail, and incorporating the choice of the templates in the linear model might enlarge and complicate the program remarkably. Finally, in a linear programming approach, the objective function must be linear or at least allow a linear reformulation.

In contrast, our algorithm usually provides a (suboptimal) solution after a reasonable amount of time, the choice of the templates is an integral part of the algorithm, and the shape of the objective function does not matter. Our algorithm is a heuristic, but one can easily derive an exact algorithm from our approach as well, for example using branch–and–bound or dynamic programming techniques.

The paper is organized as follows. In Sect. 2, we formulate the problem exactly and state our main $\mathcal{NP}$–completeness results. In particular, we will explain what we mean by reasonably fine refinements, and we will introduce the objective function we are faced with. Proofs of our $\mathcal{NP}$-completeness theorems can be found in the appendix. Then, in Sect. 3, we will present our algorithm. In Sect. 4, computational experiences are discussed, and finally, in Sect. 5, we discuss variations on the problem.

2 Exact Formulation of the Problem

In many long discussions with the engineers, we have developed an exact model, which satisfies the three substantial requirements: It reflects all numerical needs, which are given only as rules of thumb; it is suitable for a purely discrete approach; and it can be easily adapted to changes of the numerical needs. (See Sect. 5 for the importance of the last requirement.) This model is the basis for everything to follow.

In this model, an instance of the problem is given as an undirected, biconnected graph $G = (V, E)$, and a set $H = T \cup Q$ of triangles and quadrilaterals, the *holes* of G. For each edge, we are given the set of all incident holes. Conversely, for each hole we are given the set of all incident edges. The latter set is subdivided into three or four subsets, namely, one for each side of the hole. Each of these subsets is sorted according to the order of the edges along the hole. Moreover, for each edge $e \in E$ we are given four numerical parameters, its *length* L_e, its *target length* L_e^{targ}, an *upper tolerance factor* α_e, and a *lower tolerance factor* β_e.

As explained in the introduction, the problem is to find, for each edge, a collection of subdivision points on that edge such that each hole can be refined according to some template, and such that the refinement is everywhere reasonably fine. This is what the four numerical values per edge are for. More precisely, the additional points on an edge $e \in E$ must subdivide e such that the length of each resulting segment is at most $L_e^{\mathrm{targ}} \cdot \alpha_e$ and at least $L_e^{\mathrm{targ}} \cdot \beta_e$. A subdivision close to L_e^{targ} is preferred. Without loss of generality, we henceforth restrict attention to subdivisions where each edge

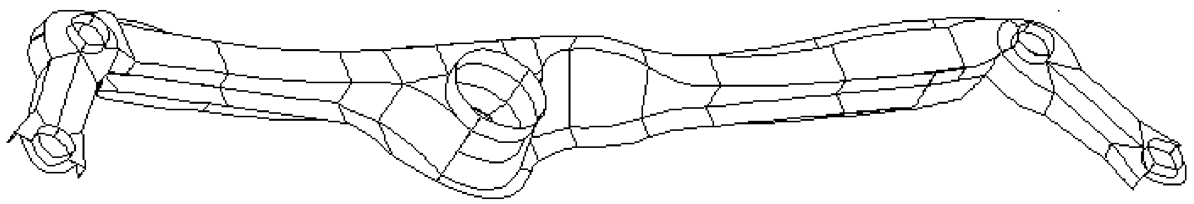
Figure 7: Part of the chassis of a car modeled by a mesh of spheric polygons.

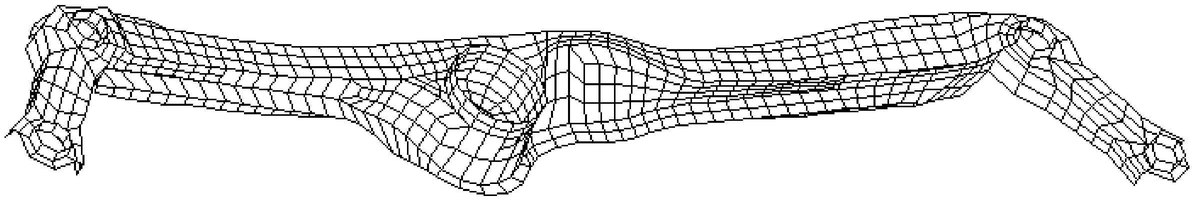
Figure 8: The refinement produced by our algorithm for the mesh in Fig. 7.

is subdivided into segments with uniform length.

The parameter L_e^{targ} defines what "reasonably fine" means for a single edge and is the result of a compromise between exactness and running time of the numerical analysis. The finer the finite–element mesh is, the more exact will be the results of the numerical analysis, but the more time it will take. However, some regions of the input mesh will require a finer approximation than others for reasonably exact results. For example, this concerns regions where the absolute value of the curvature is very high. In those regions, the target lengths of the edges are fairly small. On the other hand, in regions where no difficulties occur, the target lengths will be rather large in order not to waste running time unnecessarily. In general, it is not possible to subdivide each edge according to its target length. But this is not necessary. It suffices to differ not too much from the optimal length. This is what the tolerance factors are for. Typical values are $\alpha_e = 2$ and $\beta_e = 1/2$.

This concludes the definition of the feasibility version of our problem, regardless of any optimality criterion. This problem is already highly intractable.

Theorem 2.1 *The problem of finding a feasible solution is strongly $\mathcal{NP}$–hard. This is true even if we impose two restrictions simultaneously: firstly, we do not allow folding edges, and secondly, each of L_e^{targ}, α_e, and β_e is a global value independent of $e \in E$.*

Proof: Reduction from a variant on SATISFIABILITY, see Appendix. □

Now we turn attention to the objective function. The objective function should punish bad choices of templates for quadrilaterals. Whether a template is regarded as being good or bad for a single quadrilateral mainly depends on its shape. But depending on the application some other criteria might be important, too (for example, small warp coefficients). Thus, we give a very general formulation of the optimization problem in the following.

For a feasible solution x and quadrilateral $q \in Q$ let $t_q(x) \in \{1, 2, 3\}$ be the realized template of q given x (notice that Template 1 is a proper and unique special case of both Template 2 and 3; hence in case of ambiguity Template 1 will be assumed). With each quadrilateral $q \in Q$ the engineer associates weight functions $f_q(t_q(x), x)$ which express his preferences. A smaller weight means a better solution. Then the overall quality of a feasible solution x is measured by $\sum_{q \in Q} f_q(t_q(x), x)$ and has to be minimized in the optimization problem. The engineer is provided with a carefully chosen default objective function which reflects the basic criterion of well shaped elements. But he may alter this proposal according to his actual needs.

In our implementation, we were given the following rule of thumb as the main objective: The more a quadrilateral looks like a parallelogram, the better will be Template 1. The more a quadrilateral looks like a trapezoid, the better will be Template 2. Finally, if neither is true, Template 3 will be the best. This intuitive rule is modeled using two parameters, λ_1^q and λ_2^q, for each quadrilateral $q \in Q$, which describe the shape of q. That is, λ_1^q is the ratio of the lengths of two opposite sides of q, and λ_2^q, that of the other two sides. Without loss of generality, we assume $\lambda_1^q \geq \lambda_2^q \geq 1$. Now if both λ_1^q and λ_2^q are close to 1, Template 1 is favorable. If only λ_2^q is close to 1, Template 2 is, otherwise Template 3. For $i = 1, 2, 3$, we punish the choice of template i for q with $f_i(\lambda_1^q, \lambda_2^q)$, where f_1 is monotonously increasing in λ_1^q, f_2 in λ_2^q/λ_1^q, and f_3 in $1/\lambda_2^q$. Consider a feasible solution where template $i_q \in \{1, 2, 3\}$ is chosen for each quadrilateral $q \in Q$. Then the value of the objective function for this solution is $\sum_{q \in Q} f_{i_q}(\lambda_1^q, \lambda_2^q)$. This model reflects the above rule of thumb for good choices of templates and leaves enough degrees of freedom for fine tuning. The latter is important in view of Sect. 5.

The exact formulations of the three functions are subject to still on–going computational experiments. But this is no problem, because we modeled the objective function so that modifications do not affect the formulation of the rest of the algorithm (this in contrast to a linear programming approach, for example). For our preliminary computational results, we used linear functions f_i (which is, however, not likely to be the best choice).

In view of Theorem 2.1 there is not much hope to have an efficient algorithm for the optimization problem. Even worse, Theorem 2.2 shows that its associated recognition version is strongly $\mathcal{NP}$-complete even if at least one feasible solution is easy to obtain. Furthermore, this remains valid for the special case where no folding edges occur.

Theorem 2.2 *Given a feasible instance, a solution for this instance, and a value k, it is still strongly $\mathcal{NP}$-complete to determine whether or not there is a solution having a value of at most k. This remains true for instances without folding edges and where L_e^{targ}, α_e, and β_e are global values independent of $e \in E$.*

Proof: Reduction from 3-EXACT-COVER, see Appendix. $\square$

3 The Algorithm

The following description of the algorithm does not completely correspond to our actual implementation, but might better reveal the underlying ideas and principles.

The basic idea underlying our discrete model is a reduction to a certain *network flow* problem by fixing all templates of quadrilaterals and all subdivision points of all folding edges. To this end, we construct an auxiliary undirected network $\tilde{G} = (\tilde{V}, \tilde{E})$, which depends on given choices of templates and subdivisions of folding edges. A vertex in $\tilde{V}$ is called a *terminal*, if it has degree 1, and a *transshipment vertex*, otherwise. All possible auxiliary graphs of one–hole meshes are depicted in Fig. 4, and the auxiliary network for the instance of Figs. 1 and 2 is shown in Fig. 9.

Reformulation as a network flow problem.

Any feasible refinement of G corresponds to a certain weighting $(x(\tilde{e}))_{e \in \tilde{E}}$ of the edges of $\tilde{G}$. To make this relation one–to–one, we impose two restrictions on these weightings. First of all, each edge $\tilde{e} \in \tilde{E}$ is given an *upper capacity* $u(\tilde{e})$ and a *lower capacity* $l(\tilde{e})$. The first restriction on x is $l(\tilde{e}) \leq x(\tilde{e}) \leq u(\tilde{e})$ for all $\tilde{e} \in \tilde{E}$. An edge $\tilde{e} \in \tilde{E}$ that crosses a non–folding edge $e \in E$ is given the following capacities: $l(\tilde{e}) = \lceil L_e/L_e^{\text{targ}} \cdot \alpha_e \rceil - 1$ and $u(\tilde{e}) = \lfloor L_e/L_e^{\text{targ}} \cdot \beta_e \rfloor - 1$. (Folding edges are handled separately below.) For every edge $\tilde{e} \in \tilde{E}$ that crosses no edge in E, we set $l(\tilde{e}) = u(\tilde{e}) = 1$, if e is incident to a terminal, and $l(\tilde{e}) = 0$ and $u(\tilde{e}) = +\infty$ otherwise.

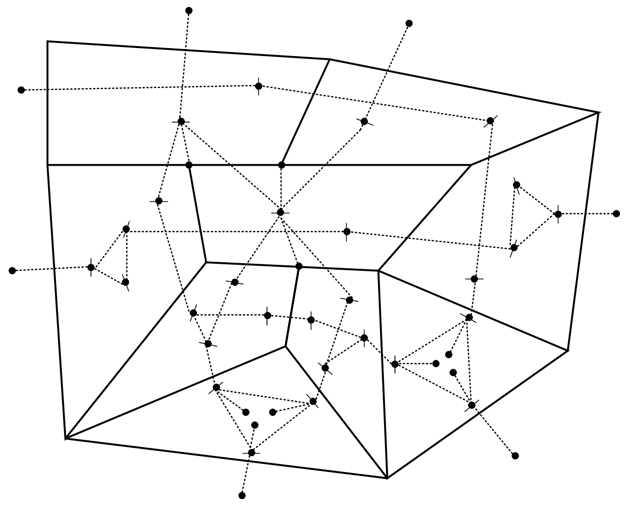

Figure 9: The auxiliary graph $\tilde{G} = (\tilde{V}, \tilde{E})$ for the instance of Fig. 1, induced by a heuristically good choice of templates for the quadrilaterals as shown in Fig. 2.

For the second restriction, the adjacency list of each transshipment vertex is partitioned into two parts, and the second restriction on x is that, for any such vertex, both partition sets, say P_1 and P_2, are equally weighted: $\sum_{\tilde{e} \in P_1} x(\tilde{e}) = \sum_{\tilde{e} \in P_2} x(\tilde{e})$. This set of restrictions enforces that no additional line of the refined mesh may end somewhere in the middle of a hole, except where this is explicitly desired (and modeled by an additional terminal inside the hole).

The latter class of restrictions may be seen as a generalization of the usual flow conservation conditions in directed graphs: For any vertex of a directed graph, consider the partition of the adjacency list into the set P_1 of all arcs *leaving* the vertex and the set P_2 of all arcs *pointing to* the vertex. Then the usual flow conservation condition is just a special case of the above equality. (It might be worth to notice that this is a *proper* special case.)

To go more into details, first consider a mesh consisting of a single triangle, see Fig. 4. In any feasible refinement, there is one additional line for each side which starts on that side and ends in the interior of the triangle. All further additional lines go from one side to another side. We may translate such a feasible refinement straightforwardly into a feasible weighting x of $\tilde{E}$ (and vice versa):

Let $\tilde{e}_1$, $\tilde{e}_2$, and $\tilde{e}_3$ be the edges of $\tilde{E}$ crossing the three sides of the triangle. We define $x(\tilde{e}_1)$, $x(\tilde{e}_2)$, and $x(\tilde{e}_3)$ to be the number of additional lines in the refinement which meet the corresponding side e_i, respectively. In other words, $x(\tilde{e}_1)$, $x(\tilde{e}_2)$, and $x(\tilde{e}_3)$ are the numbers of additional mesh vertices placed on the sides of the triangle. The x–value of the edge incident to $\tilde{e}_1$ and $\tilde{e}_2$ is the number of additional lines going from the side corresponding to $\tilde{e}_1$ to the side corresponding to $\tilde{e}_2$, and the x–values of the other two edges on the 3–cycle are defined analogously. Finally, we set the x–values of the three internal terminals equal to 1, which reflects the three additional lines that end in the interior of the triangle and form a 'wye' (emphasized in Fig. 3).

The reader may verify that the three possible auxiliary networks for a single quadrilateral (one for each template) actually model feasible refinements in the same manner. The auxiliary graph for a mesh consisting of several holes is constructed from the auxiliary graphs for the individual holes as follows (see Fig. 9). Let $h \in H$ be a hole of G, let s be one side of h, and let $e_1, \ldots, e_k \in E$ be the edges forming s. In the one–hole case, there is a single edge $\tilde{e}$ crossing s in the auxiliary network. Let v be the transshipment vertex inside h that is incident to $\tilde{e}$ and let P be the partition set in the adjacency list of v to which $\tilde{e}$ belongs. Then the edge $\tilde{e}$ is replaced by $2k - 1$ edges, all of which are incident to v and belong to P.

Among these $2k-1$ edges, there are k edges, $\tilde{e}_1, \ldots, \tilde{e}_k \in \tilde{E}$, such that $\tilde{e}_i$ crosses e_i, respectively. If e_i is incident to exactly one hole, $\tilde{e}_i$ connects v with a new terminal, and the upper and lower capacity of $\tilde{e}_i$ are determined by the length of e_i as described above. If e_i is incident to exactly two holes, then the two edges crossing e_i (one for each hole) are identified with each other, and the upper and lower capacities are again defined as described above. Finally, if e_i is a folding edge, $\tilde{e}_i$ connects v with a new terminal, and the upper and the lower capacity of $\tilde{e}_i$ are set to the value chosen in advance for that folding edge.

Each of the remaining $k-1$ edges corresponding to s connects v with a new terminal, and as usual, the upper and the lower capacity of such an edge are defined to be 1. These $k-1$ edges are necessary to ensure that each of the original mesh vertices on side s, that is, each vertex between two subsequent edges e_i and e_{i+1}, will be considered in the flow conservation conditions. In other words, this ensures that in any refinement each of these vertices meets exactly one additional line through h. This will make the refined mesh conforming.

The reader may verify that any feasible refinement of G induces a feasible weighting of $\tilde{E}$ and vice versa. This completes the reformulation as a network flow problem.

Outline of the algorithm.

We are now going to formulate the algorithm on a high level. In the beginning, we "guess" a heuristically good choice of templates and of subdivisions of all folding edges. This choice defines a network flow instance as described above. We try to solve this problem, and if we succeed, we are done. Otherwise we try to find the "bottlenecks" that cause the trouble. We try to get rid of these bottlenecks by slightly modifying our initial choices.

This defines another network flow instance, which (this is our hope) induces a better solution. We try to solve the new instance, and so forth. This is repeated until a good solution is found or a fixed time limit is exceeded. In the latter case, the algorithm returns the "least bad" solution seen during the algorithm. In this case, the user will still have to do some patchwork. But the better the output is, the less work is left to do.

Somewhat more formally, our algorithm looks as follows.

High level description of the mesh refinement algorithm:

```
1. Guess a good initial choice of all templates of quadrilaterals and of the subdivisions of all folding
   edges ;
2. repeat
   (a) construct the auxiliary flow network corresponding to the current choices of templates and
       subdivisions of folding edges ;
   (b) try to solve this network flow instance ;
   (c) if not solution OK then modify the current choices slightly to get rid of bottlenecks in
       the auxiliary flow network ;
   until solution OK or time over ;
3. if solution OK then return solution
                  else return least infeasible
                       "solution" seen .
```

This completes the high–level description. We are now going to fill in the details.

Solving the network flow problem.

Our algorithm for the auxiliary network flow problem generalizes a common approach to the feasible circulation problem. We start with a weighting $x(\cdot)$ of $\tilde{E}$ which satisfies the capacity constraints but not necessarily the flow conservation conditions at the transshipment vertices. Clearly, for each edge $\tilde{e} \in \tilde{E}$ with $l(\tilde{e}) = u(\tilde{e})$, we set $x(\tilde{e})$ to this value. Any other edge $\tilde{e} \in \tilde{E}$ which crosses an edge $e \in E$ is given the weight that approximates the target length of e best: $x(\tilde{e}) := \mathsf{round}\,(L_e/L_e^{\mathrm{targ}}) - 1$.

After that, we repeatedly modify x such that, after each iteration, the capacity constraints are still satisfied, the imbalance of no transshipment vertex is increased, and the imbalance of at least one transshipment vertex is strictly reduced. As mentioned in the introduction, the notion of *augmenting path* generalizes to this network flow problem. In each iteration, we will modify the flow along such an augmenting path. This ensures that the balance of no internal vertex on this path is changed by this augmentation. Moreover, in each case we will select the path such that the balance of neither endvertex increases and the balance of at least one endvertex decreases.

However, unlike in usual flow problems, the augmenting paths are not proper paths, because they may contain vertices and even edges more than once. Therefore, we define augmenting "paths" only as sequences of vertices and edges. Each edge in such a sequence is associated a sign, that is $'+'$ or $'-'$. If the sign of an edge is $'+'$, we increase the weight on this edge, else we decrease it (all by the same amount, of course).

Definition 3.1 *An augmenting path P in $\tilde{G}$ is a sequence $(\tilde{v}_0 - \tilde{e}_1 - \tilde{v}_1 - \cdots - \tilde{v}_{k-1} - \tilde{e}_k - \tilde{v}_k)$, with associated signs $\sigma_1, \ldots, \sigma_k \in \{'+', '-'\}$, which fulfills the following properties:*

1. *Each edge $\tilde{e}_i$ is incident to $\tilde{v}_{i-1}$ and $\tilde{v}_i$, $i = 1, \ldots, k$.*
2. *For an edge $\tilde{e}$, let $\Sigma_{\tilde{e}}$ denote the number of indices i with $\tilde{e} = \tilde{e}_i$ and $\sigma_i = '+'$* **minus** *the number of indices i with $\tilde{e} = \tilde{e}_i$ and $\sigma_i = '-'$. If $\Sigma_e > 0$, we have $x(\tilde{e}) + \Sigma_{\tilde{e}} \leq u(\tilde{e})$, else we have $x(\tilde{e}) + \Sigma_{\tilde{e}} \geq l(\tilde{e})$.*
3. *a) Vertex $\tilde{v}_0$ is an imbalanced transshipment vertex, and "augmenting" the flow along the path reduces the imbalance of $\tilde{v}_0$.*

   b) If the end-vertex $\tilde{v}_k$ is a transshipment vertex, too, then $\tilde{v}_k$ is imbalanced, and the augmentation along the path reduces the imbalance of $\tilde{v}_k$ as well.
4. *Each of the vertices $\tilde{v}_1, \ldots, \tilde{v}_{k-1}$ is a balanced transshipment vertex. For $i = 1, \ldots, k-1$, $\tilde{e}_i$ and $\tilde{e}_{i+1}$ belong to the same partition set in the adjacency list of v_i if and only if $\sigma_i \neq \sigma_{i+1}$.*

Our network flow algorithm may now be formalized as follows.

Network flow algorithm:

```
1. for $\tilde{e} \in \tilde{E}$ do if $\tilde{e}$ crosses $e \in E$
     then $x(\tilde{e}) :=$ round $(L_e / L^{\text{targ}}) - 1$
     else  $x(\tilde{e}) := l(\tilde{e})$ ;
2. failed := FALSE ;
3. $L :=$ the list of all imbalanced transshipment vertices ;
4. while $L \neq \emptyset$ do
   (a) let $v$ be an imbalanced vertex ;
   (b) try to find an augmenting path starting with $v$ ;
   (c) if failure then
         i. remove $v$ from $L$ ;
        ii. failed := TRUE ;
       else
         i. augment flow along $p$ until at least one edge is saturated ;
        ii. if $v$ is now balanced
              then remove $v$ from $L$ ;
       iii. if the terminal of $p$ is a transshipment vertex and now balanced
              then remove it from $L$ ;
```

The list L is roughly sorted according to increasing values of the total imbalances. That is, first we try to balance the almost balanced transshipment vertices, and the heavily imbalanced vertices only afterwards. This means that we first consider the imbalanced vertices that can, probably, be balanced. And the vertices that are likely to be the true bottlenecks remain imbalanced throughout the network flow algorithm and can, hence, be easily identified later on.

Heuristic search for augmenting paths.

As mentioned in the introduction, we look for augmenting paths only by a fast heuristic and therefore may miss some of them. Our search for augmenting paths is a variation on depth-first search. At first, it initializes for all edges and nodes the information that they have not been examined in the search process so far, and creates an empty path P. Then, an augmenting path is constructed incrementally by adding one edge at a time. In the meantime, the procedure maintains a *partial augmenting* $(\tilde{v}_0, n)$*-path.* This is a path from the starting-point $\tilde{v}_0$ to some current node n which is admissible with the exception that the current node may violate condition 3b) in Definition 3.1. The path search is succsessfully finished if and only if condition 3b) is also fulfilled. We say that an *edge* $e\_next = (n, n\_next)$ is *admissible* relative to a partial augmenting $(\tilde{v}_0, n)$-path in $\tilde{G}$ if the $(\tilde{v}_0, n)$-path together with the new edge e_next forms a valid partial augmenting $(\tilde{v}_0, n\_next)$-path. If the current node n has an admissible edge *e_next = (n, n_next)* (returned by *find_first_edge_on_-path* or *find_next_edge_to_add*, resp.), we add this edge to the partial augmenting path *(push_on_path)*. With each edge we add to the current path, we also store its associated sign and a *change_value*. As an invariant, the change_value of an edge e denotes the residual capacity on the path from the starting-point to e.

```
procedure augmenting_path_search($\tilde{G}$, $x$, $\tilde{v}_0$, $path$)

 1. create a path with starting-point $\tilde{v}_0$, but no edges;
 2. initialize node and edge labels (no node is on the current path,
    no edge has been checked for admissibility);
 3. while e := find_first_edge_on_path ($\tilde{v}_0$, $change\_value$) $\neq NIL$ do
     (a) push_on_path( $e$, $change\_value$);
     (b) let $n$ be the current node; { assumed $e = (\tilde{v}_0, n)$;}
     (c) while path is not empty do
          i. if  node_is_end-vertex($n$) then
                 return path;
         ii. if $e$ := find_next_edge_to_add( $n$, $change\_value$) $= NIL$ then
                 $e$ := pop_from_path( $change\_value$);
                 $n = n\_prev$; { assumed $e = (n\_prev, n)$;}
             else
                 push_on_path( $e$, $change\_value$);
                 $n = n\_next$; { assumed $e = (n, n\_next)$;}
        end while
    end while { there is no first edge $e$ }
 4. return "no path";
```

Figure 10: Heuristic search for augmenting paths

Then we check whether the new current node *n_next* may serve as an end-vertex *(node_is_end-vertex)*. As soon as an end-vertex is reached, the procedure stops and returns the path found. If we are not able to find a first edge, we return the information that no admissible augmenting path from $\tilde{v}_0$ exists. If no next edge can be found, we backtrack by deleting the last edge from the path *(pop_from_path)*.

Whenever we check if an edge $e = (n_1, n_2)$ is admissible for increase (decrease) or not, we afterwards label this edge as being "inadmissible for increase (resp. decrease) coming from endpoint n_1 (resp. n_2)", and so touch each edge at most four times: at most once in each direction, and in each case at most once for increasing and decreasing the value, resp. This restriction of edge examinations makes the algorithm fast but non-exact, because in certain (very unlikely) situations one of the paths missed may be crucial.

A pseudocode version of our procedure for augmenting paths is given in Fig. 10.

Filling in all remaining gaps.

It remains to show how to realize steps 1 and 2c of the mesh refinement algorithm of page 9. In step 1, we simply choose for each quadrilateral the template that fits its shape best (with a slight overemphasis on the first template), and for each folding edge, we choose the weight that approximates its target length best.

In step 2c, we look for quadrilaterals that currently have to satisfy Template 1 and have an imbalanced transshipment vertex inside. For a few of them, Template 1 is then relaxed to Template 2 or Template 3. Focusing on Template 1 is justified by the observation that for most quadrilaterals the initial choice is usually Template 1; because Template 1 is overemphasized, Template 2 or Template 3 is chosen as the initial template only if a quadrilateral is an extreme trapezoid or kite, and in this case switching the template is not likely to help much. Moreover, focusing on *imbalanced* quadrilaterals is justified by our rule to balance first the vertices that are likely to be no real bottlenecks. The vertices that could not be balanced are probably bottlenecks, and so changing templates of those quadrilaterals will probably remove bottlenecks and result in a "less unsolvable" network flow instance.
For changing values of folding edges, only an experimental heuristic has been implemented so far.

This completes the description of the algorithm. We shall note that all parts of the algorithm except the formal descriptions of the main algorithm and the network flow algorithm are still subject to on–going research.

4 Computational Experiences

We have implemented our algorithm on a Sun Sparc station under SunOS 5.1. The programming language is C++, and we used the GNU compiler by the Free Software Foundation, Inc. Some basic data structures of LEDA have been used for implementing the algorithm [MN]. Our front end is ISAGEN.

In some points, our implementation differs from the description in Sect. 3 in order to speed up the algorithm. For example, we do not have terminals in our auxiliary network, but work with modified flow conservation conditions. To give yet another example, we do not realize Template 2 for quadrilaterals as shown in Fig. 4, but we replace the triangle inside the quadrilateral by a single vertex with a completely different kind of flow conservation condition, which does not fit into the flow model in Sect. 2.

Besides a number of artificial instances, we have applied the algorithm to six *benchmark instances*, which stem from practice and are used by German car manufacturers for testing CAD systems offered by software houses. The sizes of the instances range from 150 up to 500 polygons.

Although only a preliminary version of the algorithm has been implemented so far, the results are very encouraging. Five out of the six examples could be refined completely, that is, our algorithm does not cause any additional patchwork (not even for "beautifying"). For the last instance, our algorithm determines a refinement for all polygons but two. This seems to be caused by a complicated folding structure of the sixth instance. It might be worth to notice that, in addition to the templates in Fig. 4, ISAGEN supports an emergency template, which could be immediately applied to these two holes without changing our solution any further. (As mentioned in the introduction, the patch is usually not that easy.)

The first five instances required less than one minute each, and the last instance, less than two minutes. This includes reading all data from a file in a certain standard format (200–600K size) and writing the result into another file. This will not be necessary any longer, once the main code is inserted in ISAGEN. However, distinguishing between the time for I/O and the running time of the algorithm does not make sense so far, since in our preliminary implementation the time for I/O cannot be distinguished from the time for building up all data structures. Clearly, the latter should be taken into account.

5 Variations on the Problem

It has been important for us to consider variations on the problem at each stage of our research, because the problem definition we are faced with is subject to on–going research, too.

First of all, this includes the objective function and the set of templates. However, if the objective function changes, we do not have to change anything else but just the objective function (again, this in contrast to a "more mathematical" approach such as linear programming).

On the other hand, if the set of templates changes, we only have to extend steps 1, 2a, and 2c. However, this affects only minor details of these three steps, not the general procedure, and our partners can do that themselves without knowing much about discrete mathematics (even long time after the cooperation is over).

At least, all this remains true as long as the modified problem can still be formulated as the network flow problem of Sect. 3. But the flow conservation conditions are a more or less natural reformulation, in terms of graph theory, of the requirement that the resulting mesh be conforming, and this is a condition the engineers will never drop.

Besides that, it might be promising to develop new variations on the problem that can even be better solved than the original problem by a discrete approach. The problem formulation we are faced with is not god–given, but the — necessarily imperfect — formalization of informal rules of thumb. This leaves many degrees of freedom, which can be used to find a problem definition that reflects all numerical requirements at least as well, but is more suitable for a network flow approach.

In future, it might be interesting to see whether or not our approach extends to other models. For example, in computer graphics, meshes that approximate surfaces usually have to consist of triangles rather than quadrilaterals. Another example is the approximation of compact bodies using fully three–dimensional meshes.

It is not clear that those variations can still be solved by network flow techniques. But it seems that, in any case, there is at least a discrete core problem, which can be isolated and solved disregarding all numerical and geometrical aspects, and which is crucial for the original, non–discrete, problem itself.

References

[AMO] R.K. Ahuja, T.L. Magnanti and J.B. Orlin: *Network flows.* Prentice Hall (1993).

[Ho] K. Ho–Le: *Finite element mesh generation methods: a review and classification.* Computer–Aided Design **20** (1988), 27–38.

[Kr] G. Krause: *Interactive finite element preprocessing with ISAGEN.* In: J. Robinson, ed., Finite Element News **15** (1991).

[De] U. Derigs: *Programming in Networks and Graphs.* Lecture Notes in Economics and Mathematical Systems, vol. 300, Springer-Verlag, Berlin (1988).

[Ed] J. Edmonds: *An introduction to matching.* Lecture Notes. The University of Michigan, Ann Arbor (1967).

[MN] K. Mehlhorn and S. Näher: *LEDA, a library of efficient data types and algorithms.* Technical report TR A 04/93, FB 10, Universität des Saarlandes, Saarbrücken (1989).

[PS] C.H. Papadimitriou and K. Steiglitz: *Combinatorial optimization — algorithms and complexity.* Prentice Hall (1982).

[SNTM] V. Srinivasan, L.R. Nackman, J.–M. Tang and S.N. Meshkat: *Automatic mesh generation using the symmetric axis transformation of polygonal domains.* Proceedings of the IEEE **80** (1992), 1485–1501.

[TA] T.K.H. Tam and C.G. Armstrong: *Finite element mesh control by integer programming.* International Journal for Numerical Methods in Engineering **36** (1993), 2581–2605.

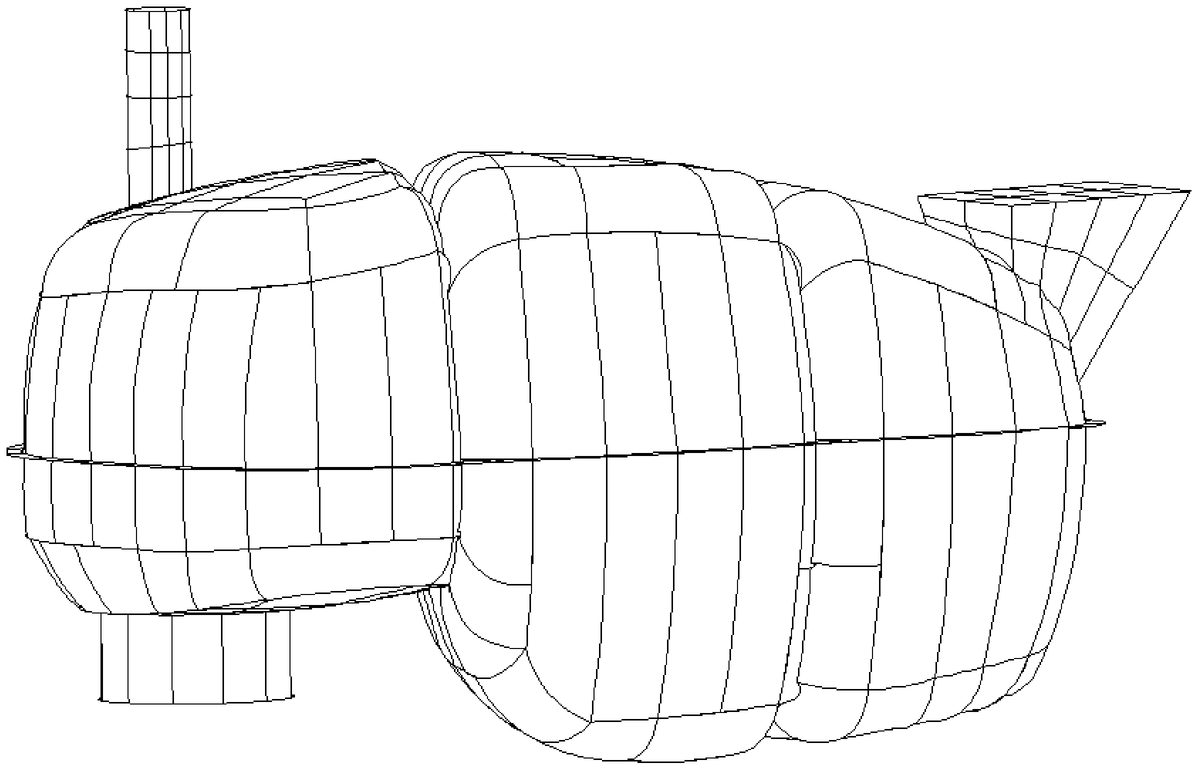

Figure 11: Part of the cooling system of an engine.

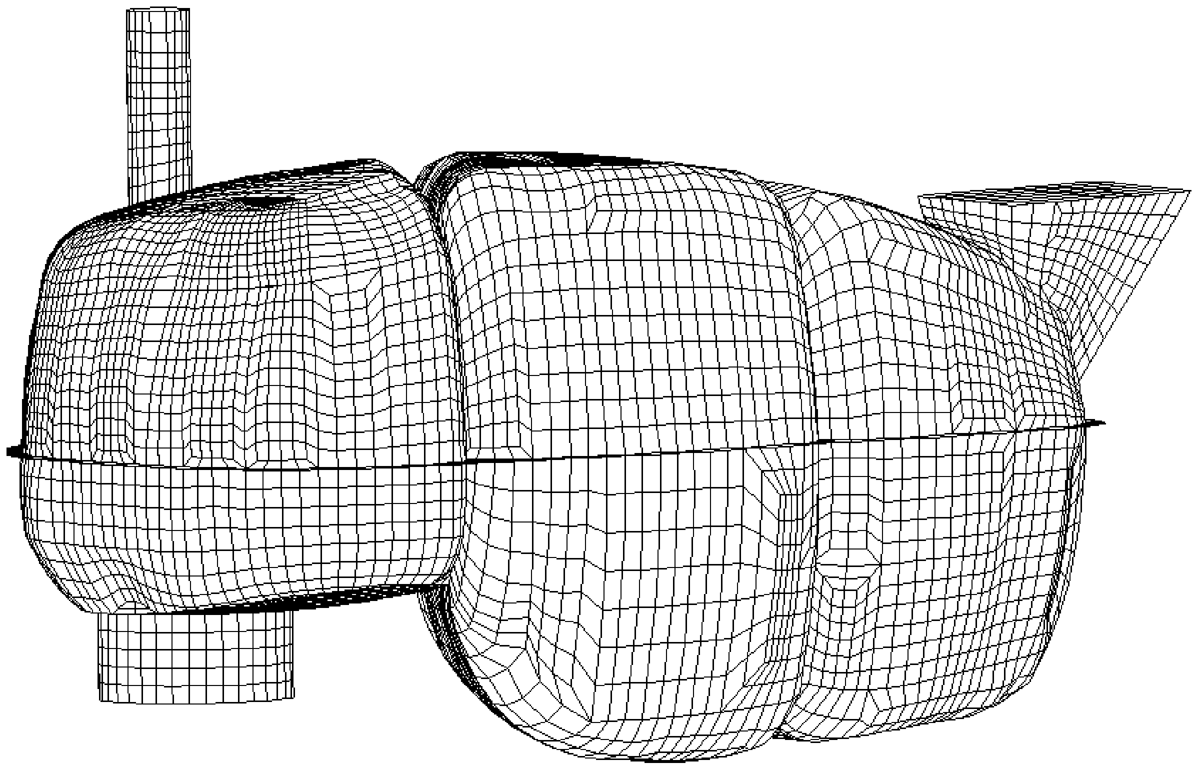

Figure 12: The refinement produced by our algorithm for the mesh in Fig. 11.

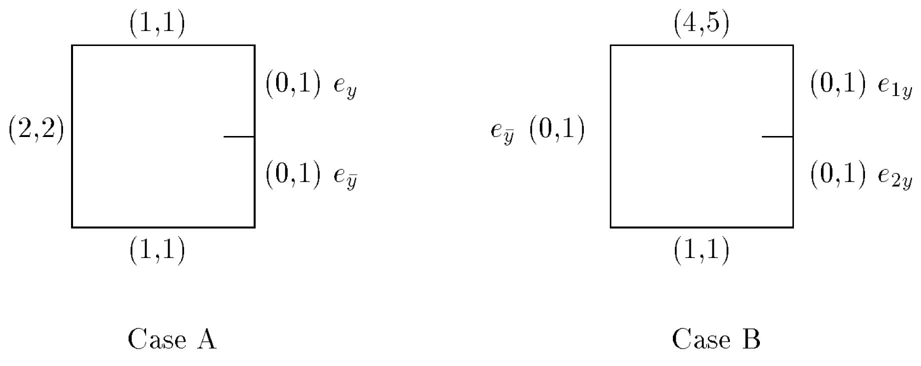

Figure 13: Representation of variable y

Appendix

Theorem 2.1 *The problem of finding a feasible solution (as defined in Sects. 1 and 2) is strongly $\mathcal{NP}$–hard. This is true even if we impose two restrictions simultaneously: firstly, we do not allow folding edges, and secondly, each of L_e^{targ}, α_e, and β_e is a global value independent of $e \in E$.*

Proof: The problem is in $\mathcal{NP}$ because a feasible solution, if it exists, can be checked for feasibility quite easily. To show $\mathcal{NP}$-completeness we shall give a reduction from the following variant of SATISFIABILITY:
We are given a Boolean formula consisting of m clauses $C_1, ..., C_m$ (in conjunctive normal form) and involving the variables $y_1, ..., y_n$, each of which appears only once or twice unnegated and exactly once negated. Is the formula $C_1 \cdot C_2 \cdot \ldots \cdot C_m$ satisfiable ?
Even with this restriction SATISFIABILITY remains $\mathcal{NP}$-complete, as was shown in [PS, p. 394 ff.].

Let $F = C_1 \cdot C_2 \cdot \ldots \cdot C_m$ be a formula of that type. As in many $\mathcal{NP}$-completeness proofs, the construction of an instance of our problem needs the design of special-purpose components:

For each variable $y \in \{y_1, ..., y_n\}$ we define a corresponding quadrangle. If the variable y appears exactly once negated and once unnegated, we create a quadrangle with capacity constraints (the pair of lower and upper capacities is indicated in brackets for each edge) as shown in Fig. 13A. The negation of a variable is indicated by a bar. Here and in the following, we use "short lines" inside a quadrangle to mark endpoints of edges.

We observe that the only feasible solutions for this type of quadrangle meet the condition of Template 1, namely with $x_{e_y} = 1$, $x_{e_{\bar{y}}} = 0$ or $x_{e_y} = 0$, $x_{e_{\bar{y}}} = 1$. This ensures that exactly one of e_y and $e_{\bar{y}}$ has value 1, and the other one, value 0. Hence, this quadrangle encodes the truth assignment to y or $\bar{y}$.

If the variable y appears three times, we need a quadrangle with capacity constraints as shown in Fig. 13B. Here, the intended interpretation of the edges e_{1y} and e_{2y} and their values $x_{e_{1y}}$, $x_{e_{2y}}$ is the truth assignment to y, and edge $e_{\bar{y}}$ stands for $\bar{y}$. There are exactly two possibilities to find a feasible subdivision of such a quadrangle, one which meets the condition of Template 2 and one with the condition of Template 3 respectively, see Fig. 14.

In the case that Template 2 is met, we have the truth assignment $\bar{y} = 1$ and $y = 0$, and if Template 3 is met, we have $\bar{y} = 0$ and $y = 1$. Hence, the only feasible assignments to the quadrangle correspond to possible truth settings of y.

Clearly, we may assume that each clause in our formula has at least two variables, because clauses with only one variable are satisfied if and only if this single variable is set to $TRUE$ if unnegated and to $FALSE$ if negated.

The clauses are represented by quadrangles with certain capacity constraints. For a clause $C \in$

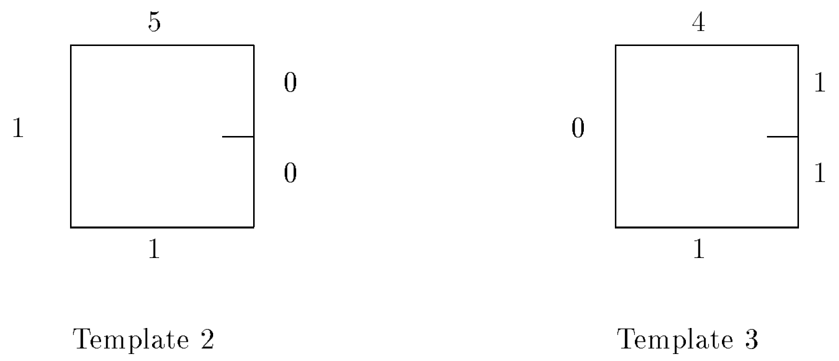

Figure 14: Feasible subdivisions in Case B

$\{C_1, ..., C_m\}$ with $k \geq 2$ variables we define a quadrangle as shown in Fig. 15.

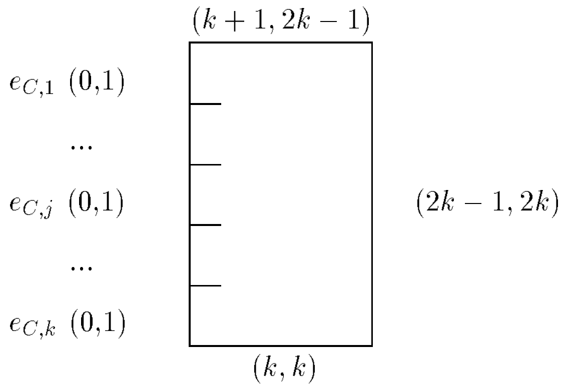

Figure 15: Representation of clause C with k literals, $k \geq 2$

This encodes the truth value of the corresponding clause, namely, such a quadrangle has a solution if and only if at least one edge $e_{C,j}$ has value $x_e = 1$. To see this, assume first that all edges $e_{C,j}$ for $1 \leq j \leq k$ have the assignment $x_e = 0$. Then we have exactly $k-1$ subdivision points on this quadrangle side, but on the opposite side at least $2k-1$. Hence, this pair of sides has a difference of subdivision points of at least k, whereas the other pair of sides has a difference of at most $k-1$ and of at least 1. Thus, none of the three template conditions Template 1 - Template 3 can be fulfilled in this case. For the other direction, assume now that l edges $e_{C,j}$ $(1 \leq l \leq k)$ have value $x_e = 1$. Then we can always meet condition Template 3 by the choice of capacities as indicated in Fig. 16.

Finally, we connect variable quadrangles with clause quadrangles in order to make the truth values of all variables and clauses consistent. To this end, edge $e_{C,k}$ of clause C is connected by a quadrangle with e_y if the k-th literal of C is y, and with $e_{\bar{y}}$ if it is $\bar{y}$, cf. Fig. 17. A feasible subdivision of such a quadrangle always meets the condition of Template 1. More precisely, the edges of the clauses and corresponding variables have to obtain the same subdivision values.

This completes the construction of the instance. If local target edge lengths and tolerance factors are used, it is immediate that all capacity constraints which arise in the designed quadrangles of this proof could have been derived from real geometric instances. For global parameters L_e^{targ}, α_e, and β_e it is possible to scale the edge lengths of the quadrangles in such a way that the same capacity

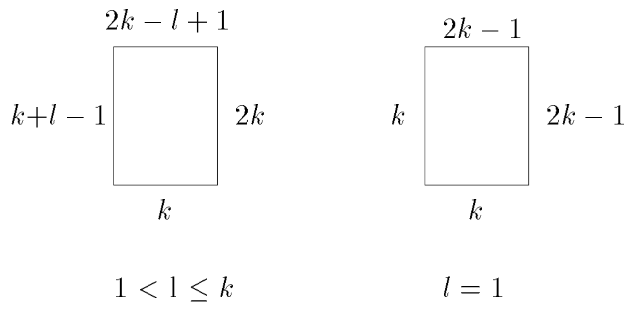

Figure 16: Quadrangles of a clause with k variables, and the choice of subdivision numbers to meet condition Template 3 if l edges are subdivided.

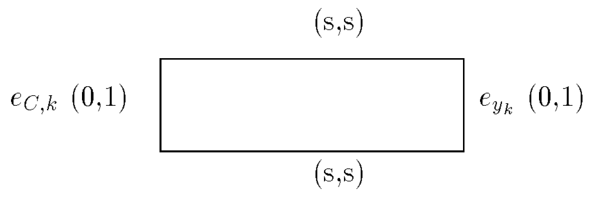

Figure 17: Connection between variables and clauses

constraints are obtained. However, these technical details are left to the reader.

Observe that no folding edge occurs. It is clear that there exists a feasible solution if and only if F is satisfiable, and the transformation is done in polynomial time. The problem is *strongly* $\mathcal{NP}$-complete, since all numbers we had to use in our reduction are small in the input length.

□

For the proof of Theorem 2.2 we need the definition of the problem 3-EXACT-COVER which is well-known to be strongly $\mathcal{NP}$-complete:
Given a family $F = \{X_1, ..., X_n\}$ of n subsets of $S = \{u_1, ..., u_{3k}\}$, each of cardinality three, is there a subfamily of k subsets that covers S?

Theorem 2.2 *Given a feasible instance, a solution for this instance, and a value k, it is still strongly $\mathcal{NP}$-complete to determine whether or not there is a solution having a value of at most k. This remains true for instances without folding edges and where L_e^{targ}, α_e, and β_e are global values independent of $e \in E$.*

Proof: Checking a given solution vector x for feasibility and summing up the realized template weights can be done quite easily. Thus, this problem is clearly in $\mathcal{NP}$. In this proof we use special template weights. We take $f_q(1, x) = 0$ for all $q \in Q$ and x with $t_q(x) = 1$ and $f_q = 1$ in all other cases (using the notation of Sect. 2, page 6). In other words, we simply want to maximize the number of quadrilaterals with realization of Template 1.

The reduction from 3-EXACT-COVER is as follows: We create a quadrangle for each subset X_j as in Fig. 18. For simplicity we will refer to it as quadrangle X_j. Edge $e_{X_j l}$ with $l \in \{1, 2, 3\}$ is to correspond with the l-th element of X_j. If one of the edges $e_{x_j l}$ gets value one, then we need Template 3 for X_j, otherwise Template 1 is possible. We embed the quadrangles X_j in the plane along a thought

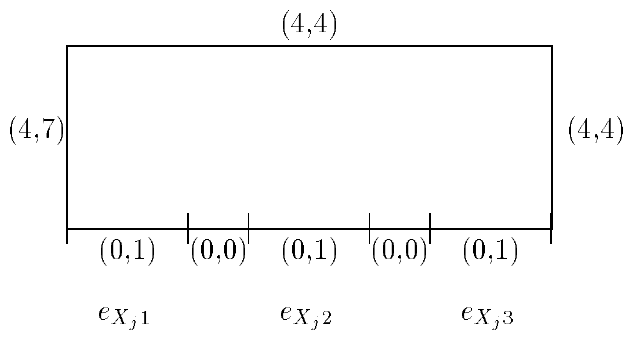

Figure 18: Subset X_j

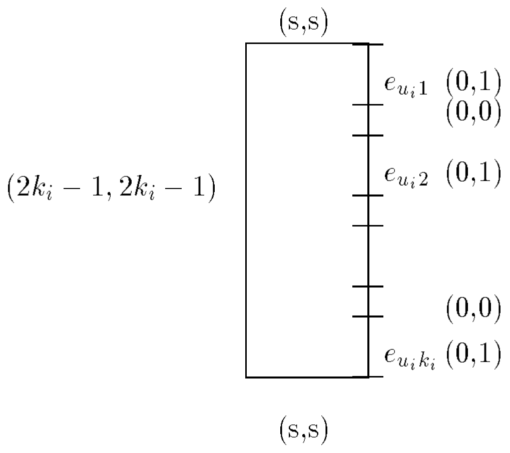

Figure 19: Element u_i

horizontal axis in such a way that they do not touch.

Let k_i denote the total number of appearances of u_i in the subsets X_j. Define for each element u_i a quadrangle as in Fig. 19. Exactly one of the k_i edges e_{u_il} has to get a subdivision of one in order to meet the condition of Template 1. All these "element" quadrangles u_i are embedded in the plane along a thought vertical axis.

Now we establish the relationship between elements u_i and subsets X_j which encodes that u_i is an element of X_j. To this end, we connect each edge e_{X_jl} with a corresponding edge of element u_i by quadrangles which we call *channels* with capacities as in Fig. 20, where the s_i are some small constants. If all quadrangles of such a channel meet Template 1, then both edges obtain the same value.

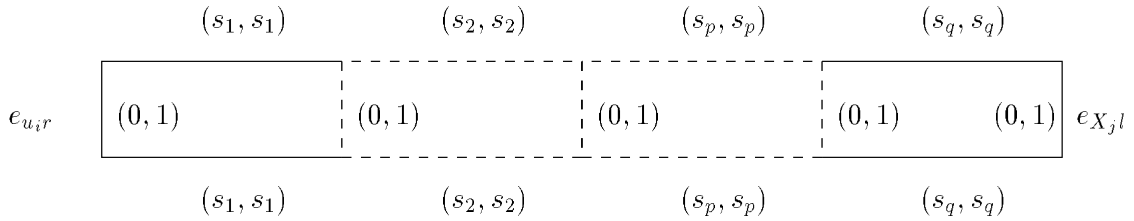

Figure 20: Channel which connects element u_i and subset X_j (with $u_i \in X_j$)

The purpose of these connections is to make sure that each element u_i may be covered by any subset X_j it belongs to. Notice that element u_i is covered in any feasible subdivision by (at least) one subset, which is connected via a channel to the chosen edge e_{u_il} with value one of element u_i.

The length of each channel, i.e. the number of its quadrangles, depends on the position of u_i and X_j in the embedding, but is at most $O(n)$. To embed the channels in the plane, we need crossings between certain channels. The corresponding quadrangles are referred to as *crossings*. Crossings have (0,1) - capacities for all edges. These capacities ensure that for all quadrangles u_k and all the channels and crossings Template 1 is always possible, but for channels and crossings also Template 3. We give a sketch of the complete construction in Fig. 21.

Given global parameters L_e^{targ}, α_e, and β_e, the edge lengths of all quadrangles can be chosen such that we get the capacities of our construction. As in the previous proof, these details are omitted. The transformation is clearly done in polynomial time, since the number of quadrangles used is at most $O(nk)$ and all capacities are small. No folding edge occurs in our construction. It is easy to see

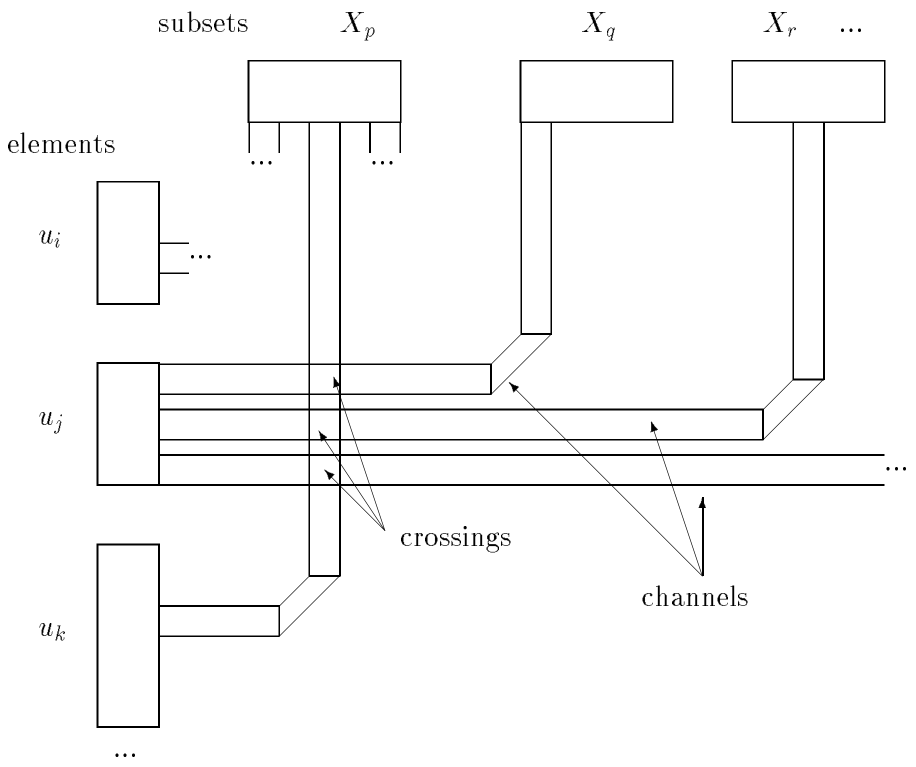

Figure 21: Sketch: Transformation of a 3-Exact-Cover instance

that there are many feasible solutions for the transformed mesh instance. (For example, choose for each element u_i exactly one subset X_{i_l} in which u_i is contained.)

If an exact cover of S exists, say with $\{X_{i_1}, ..., X_{i_k}\}$, one needs exactly these k quadrangles to fulfill Template 3 but not Template 1. Otherwise, at least one more X_i is used in the cover of S or certain crossings do not meet the condition of Template 1. In both cases this results in more than k quadrangles with Template 3 but not Template 1.

□

Reports from the group

"Algorithmic Discrete Mathematics"

of the Department of Mathematics, TU Berlin

408/1994 *Maurice Queyranne, Andreas S. Schulz:* Polyhedral Approaches to Machine Scheduling

407/1994 *Andreas Parra, Petra Scheffler:* How to Use the Minimal Separators of a Graph for Its Chordal Triangulation

401/1994 *Rudolf Müller, Andreas S. Schulz:* The Interval Order Polytope of a Digraph

396/1994 *Petra Scheffler:* A Practical Linear Time Algorithm for Disjoint Paths in Graphs with Bounded Tree-width

394/1994 *Jens Gustedt:* The General Two-Path Problem in time $O(m \log n)$, extended abstract

393/1994 *Maurice Queyranne:* A Combinatorial Algorithm for Minimizing Symmetric Submodular Functions

392/1994 *Andreas Parra:* Triangulating Multitolerance Graphs

390/1994 *Karsten Weihe:* Maximum (s,t)–Flows in Planar Networks in $\mathcal{O}(|V| \log |V|)$ Time

386/1994 *Annelie von Arnim, Andreas S. Schulz:* Facets of the Generalized Permutahedron of a Poset

383/1994 *Karsten Weihe:* Kurzeinführung in C++

377/1994 *Rolf H. Möhring, Matthias Müller-Hannemann, Karsten Weihe:* Using Network Flows for Surface Modeling

376/1994 *Valeska Naumann:* Measuring the Distance to Series-Parallelity by Path Expressions

375/1994 *Christophe Fiorio, Jens Gustedt:* Two Linear Time Union-Find Strategies for Image Processing

374/1994 *Karsten Weihe:* Edge–Disjoint (s,t)–Paths in Undirected Planar graphs in Linear Time

373/1994 *Andreas S. Schulz:* A Note on the Permutahedron of Series–Parallel Posets

371/1994 *Heike Ripphausen–Lipa, Dorothea Wagner, Karsten Weihe:* Efficient Algorithms for Disjoint Paths in Planar Graphs

368/1993 *Stefan Felsner, Rudolf Müller, Lorenz Wernisch:* Optimal Algorithms for Trapezoid Graphs

367/1993 *Dorothea Wagner:* Simple Algorithms for Steiner Trees and Paths Packing Problems in Planar Graphs, erscheint in *"Special Issue on Disjoint Paths" (eds. B. Gerards, A. Schrijver) in CWI Quarterly*

365/1993 *Rolf H. Möhring:* Triangulating Graphs without Asteroidal Triples

359/1993 *Karsten Weihe:* Multicommodity Flows in Even, Planar Networks

358/1993 *Karsten Weihe:* Non-Crossing Path Packings in Planar Graphs with Applications

357/1993 *Heike Ripphausen-Lipa, Dorothea Wagner, Karsten Weihe:* Linear-Time Algorithms for Disjoint Two-Face Paths Problems in Planar Graphs