ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report No. 97.271

Practical Performance of Efficient Minimum Cut Algorithms

by

Michael Jünger
Giovanni Rinaldi
Stefan Thienel

1997

Revised version of July 1998

Institut für Informatik
UNIVERSITÄT ZU KÖLN
Pohligstraße 1
D-50969 Köln

Addresses of the authors:

Michael Jünger
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln
Germany
E-mail: mjuenger@informatik.uni-koeln.de
Telephone: +49-221-470-5313

Giovanni Rinaldi
Istituto di Analisi dei Sistemi ed Informatica del CNR
Viale Manzoni, 30
I-00185 Roma
Italy
E-mail: rinaldi@iasi.rm.cnr.it
Telephone: +39-6-77161

Stefan Thienel
Almenrauschstraße 2
D-82110 Germering
Germany
E-mail: Stefan.Thienel@sdm.de
Telephone: +49-89-8942-92-60

Partially supported by ESPRIT Long Term Research Project Nr. 20244 (ALCOM-IT).

Abstract

In the late eighties and early nineties, three major exciting new developments (and some ramifications) in the computation of minimum capacity cuts occurred and these developments motivated us to evaluate the old and new methods experimentally. We provide a brief overview of the most important algorithms for the minimum capacity cut problem and compare these methods both on problem instances from the literature and on problem instances originating from the solution of the traveling salesman problem by branch-and-cut.

1. Introduction

Computer programs that compute minimum capacity cuts in undirected graphs with nonnegative edge capacities belong to the most intensively used basic tools in optimization. Besides the obvious direct application of deciding the degree of connectivity of a given network, the main reason is that various separation routines in cutting plane algorithms depend on the practically efficient computation of minimum capacity cuts. The most prominent example is the separation of subtour elimination constraints for the traveling salesman polytope. The efficient solvability of the minimum capacity cut problem instances arising in separation is one of the reasons why instances with up to a few thousand cities can be solved to optimality or at least provably very close to optimality. However, the traveling salesman problem (TSP) (see, e.g., [JRR95]) is not the only such example: efficient minimum capacity cut computations also play an important role in, e.g., network reliability [Sto92], automatic graph drawing [Mut95], or sequential ordering [AJR98].

In this paper, we compare experimentally the most important methods for the solution of the minimum capacity cut problem. We conduct computational experiments on problem instance families from the literature and on problem instances originating from the solution of the traveling salesman problem by branch-and-cut. We give statistics based on almost 20 000 individual runs of the minimum capacity cut algorithms we implemented for this study. The selection of these runs was influenced by experience with more than 100 000 runs that we performed while we prepared the final versions of the implementations and the experiments. Recently, a similar study has been published [CGKLS97]. Whereas in [CGKLS97] fast hybrid algorithms combining various minimum capacity cut algorithms are presented, we compare in this paper the "pure" versions of the algorithms. In particular, we do not propose any new algorithms. Rather, we have put a lot of effort in good implementations of published algorithms in order to make comparisons as fair as we can.

For an undirected graph $G = (V, E)$ and $W \subseteq V$ we let $\delta(W) := \{\{u, v\} \in E \mid u \in W, v \in V \setminus W\}$ denote the *cut* in G induced by W, write $\delta(v)$ instead of $\delta(\{v\})$ for $v \in V$ and call $\delta(v)$ the *star* of v. For a pair of distinct nodes $u, v \in V$, a (u, v)-*cut* is a cut $\delta(W)$ separating u from v, i.e., with $u \in W$ and $v \notin W$. For edge capacities c_e $(e \in E)$ and $F \subseteq E$ we denote by $c(F) = \sum_{e \in F} c_e$ the sum of the capacities of the edges in F. For a node $v \in V$, we call $c(\delta(v))$ the *star capacity* of v. For a graph G with nonnegative edge capacities c_e $(e \in E)$ the *minimum capacity cut problem* consists of finding a nonempty node set $W^* \subset V$ such that for the capacity of the cut $\delta(W^*)$ we have $c(\delta(W^*)) \leq c(\delta(W))$ for all nonempty $W \subset V$. For ease of exposition, we assume that $G = (V, E)$ is a connected graph (otherwise a cut of capacity zero can be easily found by, say, depth first search), but in all our implementations, we drop this assumption.

An important notion in the following is the concept of *shrinkable node pairs.* Let $\lambda^* = \lambda^*(G) = c(\delta(W^*))$ denote the minimum capacity of a cut in G, let $\lambda^*_{uv} = \lambda^*_{vu}$ denote the minimum capacity of a (u, v)-cut of G for a pair of distinct nodes $u, v \in V$, and let $\overline{\lambda} = c(\delta(\overline{W}))$ for some nonempty $\overline{W} \subset V$ be an upper bound for λ^*. Shrinking a pair of distinct nodes $u, v \in V$ results in an edge weighted graph $G_{\{u,v\}} = (V_{\{u,v\}}, E_{\{u,v\}})$ that is obtained from $G = (V, E)$ by deleting the edge $\{u, v\}$ if it exists in E, identifying the nodes u and v, and then replacing each pair of parallel edges that may result from the identification by a single edge whose capacity is the sum of the capacities of the two edges (see Figure 1). The node that results from the identification of u and v is called the *supernode* composed of u and v. If G is given in the form of adjacency lists, which we assume throughout this article, this operation can clearly be performed in time proportional to the sum of the node degrees of u and v. In our implementations, we maintain the invariant that the nodes of the current graph are the first consecutive entries of an array of "active" nodes. With each active node, we associate a list of original nodes that it represents. Any sequence of shrinking operations can be undone in the reverse order. Each unshrinking operation takes no more time than the corresponding previous shrinking operation.

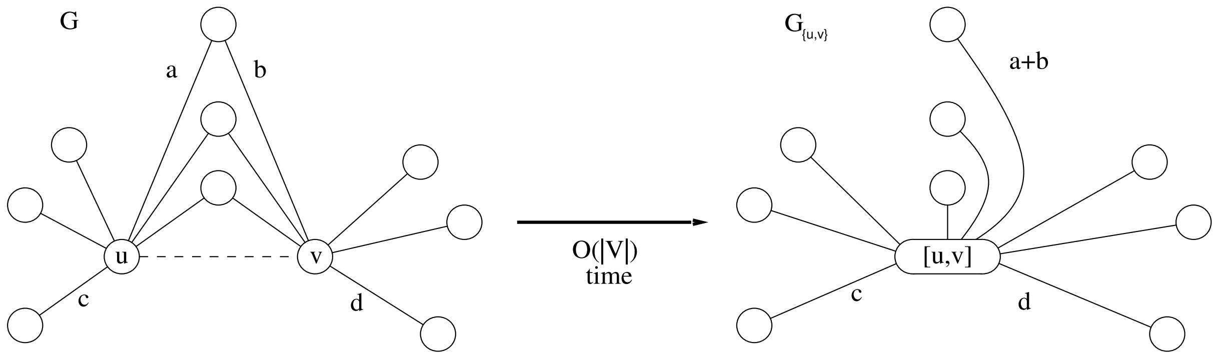


Figure 1. Shrinking a pair of nodes

From the definition of $G_{\{u,v\}}$ it follows that $\lambda^*(G) = \min\{\lambda^*_{uv}(G), \lambda^*(G_{\{u,v\}})\}$. Therefore, in the context of solving the minimum capacity cut problem for G we call a pair of nodes $u, v \in V$ *shrinkable* if either $\lambda^*_{uv}(G) \geq \lambda^*(G_{\{u,v\}})$ or $\lambda^*_{uv}(G) \geq \overline{\lambda}$. The motivation for this definition is the following. Suppose that either $\overline{W}$ or some cut in $G_{\{u,v\}}$ is at least as good as the minimum cut separating u from v. Then we can safely shrink u and v. In particular, suppose we have computed a minimum capacity cut in G that separates u and v. (This can be done by an (u, v)-max-flow-min-cut computation.) If $\overline{\lambda}$ is bigger than the capacity of the (u, v)-minimum capacity cut we can reset $\overline{\lambda}$ to λ^*_{uv}. Then u and v are shrinkable. Most of the algorithms that will be discussed in the following use more sophisticated (and computationally less expensive) conditions that guarantee that a given pair of nodes is shrinkable.

2. The Algorithms

For all minimum capacity cut algorithms that will be surveyed here and experimentally evaluated in Section 4, we will give only an account of the essential ideas and the essential features. For details and proofs we refer to the original articles that will be cited throughout.

A minimum capacity (s, t)-cut in G can be found by various (s, t)-max-flow-min-cut techniques. For example, the algorithm of King, Rao, and Tarjan [KRT94] runs in

$O(|V||E|\log_{|E|/|V|\log|V|}|V|)$ time. In practical experiments it turned out that a variant of the Goldberg-Tarjan preflow-push algorithm [GT88] using highest-level selection, global relabeling, and gap relabeling is very efficient [DM89, CG95]. When we have to solve an (s,t)-max-flow-min-cut problem in our computational experiments, we use this algorithm that runs in $O(|V|^2\sqrt{|E|})$ time. In the following, by T_{st} we denote the time it takes to compute an (s,t)-max-flow-min-cut.

Up to around 1986*, the best known way to compute a minimum capacity cut consisted of a sequence of arbitrarily choosing two distinct nodes s and t, computing an (s,t)-max-flow-min-cut, shrinking s and t, and iterating until a graph with only one node results. The cut of minimum capacity found in the course of this computation gives rise to a minimum capacity cut in G after the expansion of the supernodes in $O(|V|)$ time. We refer to this as to the "naive" method. This algorithm performs $|V|-1$ times an (s,t)-max-flow-min-cut computation followed by a shrinking operation for the nodes s and t, which are a shrinkable pair with respect to the previously computed capacity of the (s,t)-max-flow-min-cut. The running time is $O(|V|T_{st})$. In our setting, a running time of $O(|V|^3\sqrt{|E|})$ and $O(|E|)$ space results.

In the same asymptotic running time, we can as well compute a minimum capacity cut tree that gives more information than we want here, namely, a tree data structure that allows us to compute a minimum capacity (s,t)-cut for any pair $s,t \in V$ in $O(|V|)$ time. An algorithm that computes a minimum capacity cut tree was given by Gomory and Hu [GH61]. It uses complicated shrinking and unshrinking operations, and this is one of the reasons why most researchers interested in computing a minimum capacity cut efficiently in practice preferred the above "naive" method that is much easier to implement. However, in 1986, Gusfield [Gus90] showed how the same data structure can be computed in the same asymptotic time without any shrinking/unshrinking operations.

A leap in performance, but not in asymptotic worst case running time, resulted from the introduction of the Padberg-Rinaldi algorithm in 1986 [PR90]. Like in the previously described "naive" algorithm, an upper bound $\overline{\lambda}$ for a minimum capacity cut and a cut $\delta(\overline{W})$ realizing this bound via $\overline{\lambda} = c(\delta(\overline{W}))$ is always readily at hand. It can be initialized, say, with the minimum star capacity $c(\delta(v))$ among the nodes $v \in V$, and it can be updated during the computation. Padberg and Rinaldi gave sufficient conditions for a given pair of nodes to be shrinkable with respect to $\overline{\lambda}$. They also extracted a set of such conditions that can be tested in $O(|V|)$ time.

For later reference, we list these *shrinking conditions* here: For two nodes u and v, we let $c(\{u,v\}) = c(\{v,u\})$ denote the capacity of the edge linking u and v, if there is such an edge, 0 otherwise. Nodes u and v are shrinkable with respect to $\overline{\lambda}$ if

(PRCOND1) $\quad c(\{u,v\}) \geq \overline{\lambda}$

or

(PRCOND2) $\quad 2c(\{u,v\}) \geq \min\{(c(\delta(u)), c(\delta(v))\}$

* For chronological consistency we refer to the technical reports when we specify the "date of birth" of an algorithm.

or

(PRCOND3) There is a node $w \in V \setminus \{u,v\}$, such that
$\max\{(c(\delta(u)) - c(\{u,w\}), c(\delta(v)) - c(\{v,w\}) \leq 2c(\{u,v\})$

or

(PRCOND4) There is a set of nodes $S \subseteq V \setminus \{u,v\}$, such that
$c(\{u,v\}) + \sum_{w \in S} \min(c(\{u,w\}), c(\{v,w\})) \geq \overline{\lambda}$.

Whenever such a condition applies to a given pair of nodes, they can be shrunk to a supernode, and the star capacity of the resulting supernode possibly updates the incumbent pair $\overline{\lambda}, \overline{W}$. If none of the conditions applies to a certain set of trial node pairs, a "naive" step consisting of choosing some pair s and t and performing an (s,t)-max-flow-min-cut computation is taken.

This general scheme leaves some freedom to the implementor of a Padberg-Rinaldi type algorithm. In particular, it must be decided

1.) how many and which node pairs are tested in a testing step, before resorting to an (s,t)-max-flow-min-cut computation,

2.) how s and t are chosen for an (s,t)-max-flow-min-cut computation.

These two decisions have a great impact on the practical performance of the resulting algorithm. After some experimentation, we decided as follows:

1.) We initialize and maintain a priority queue of edges according to their capacities. A testing step consists of testing all four conditions for the end nodes of at most $|V|$ edges taken from the priority queue. After each shrinking operation, all edges in the star of the resulting supernode are inserted, respectively requeued, in the priority queue.

2.) In the pursuit of efficiently finding a pair s and t such that the supernode resulting from shrinking s and t has large star capacity $c(\delta(\{s,t\}))$, we choose a node s with maximum star capacity. The node t is one among the nodes in $V \setminus \{s\}$ for which the quantity $c(\delta(s)) + c(\delta(t)) - 2c(\{s,t\})$ is maximum. This strategy aims at producing a new neighbor for some node pairs, thus making condition PRCOND3 more likely to apply. An (s,t)-max-flow-min-cut computation is only performed if s and t do not happen to satisfy any shrinking criteria. In actual computation it has turned out that (s,t)-max-flow-min-cut computations are only very rarely needed.

If during the whole computation none of the shrinking criteria applies, then the Padberg-Rinaldi algorithm behaves essentially like the "naive" algorithm so that no improvement in worst case running time with respect to the naive algorithm can be achieved. However, we will see below how this technique has a tremendous impact on experimentally observed practical performance. In a way, this is not surprising as it can be interpreted as "heuristic problem size reduction", a technique that is also successfully applied in many other contexts, e.g., linear and (mixed) integer programming. A Fortran implementation by Padberg and Rinaldi has been (and still is) in widespread use. Our new implementation performs much better in practice.

In the late eighties and early nineties, three major exciting new developments (and some ramifications) in the computation of minimum capacity cuts occurred and these developments motivated us to evaluate the old and new methods experimentally.

A breakthrough in efficient minimum capacity cut computation was achieved by Nagamochi and Ibaraki in 1989 [NI92]. Slightly later, Nagamochi, Ono and Ibaraki [NOI94] published the description of an efficient implementation of this algorithmic idea, and we refer to this publication, which includes several clever algorithmic enhancements. At about the same time when the paper by Nagamochi, Ono and Ibaraki circulated as a preprint (and we started our experimental investigations) several simplifications of Nagamochi's and Ibaraki's original ideas were found by various authors. We will comment on this as well, and even include computational results for one of these variants.

The new quality of this algorithm is that it computes a minimum capacity cut without any (s,t)-max-flow-min-cut computations. In each major iteration, a shrinkable pair of nodes (here always the endnodes of an edge of G) is computed as follows: The current set of nodes V is partitioned into the *visited nodes* W and the *unvisited nodes* $\overline{W}$; initially $W = \emptyset$. For each unvisited node $u \in \overline{W}$ we maintain its *degree of connection* $c(u : W) = \sum_{v \in W} c(\{u,v\})$ to the visited nodes (initialized to 0). Until all nodes are visited, an unvisited node $u \in \overline{W}$ is chosen such that it has the maximum degree of connection $c(u : W)$ to the visited nodes among all unvisited nodes. This node is removed from $\overline{W}$ and added to W. For all remaining unvisited nodes w the quantity $c(w : W)$ is updated to $c(w : W) := c(w : W) + c(\{w,u\})$. Then the quantity $q(w,u) = c(w : W)$ is associated with the edge $\{w,u\}$, if it exists. It can be shown that $q(w,u)$ is a lower bound on the capacity of a (w,u)-cut, and that the endnodes of the last edge $\{w,u\}$ encountered in the process is indeed shrinkable. Shrinking $|V|-1$ times like this, the algorithm can be implemented to run in $\mathrm{O}(|V||E| + |V|^2 \log |V|)$ time and $\mathrm{O}(|E|)$ space (using Fibonacci heaps). Nagamochi, Ono and Ibaraki show how the quantities $q(w,u)$ can be used to determine in each major iteration not just a single shrinkable edge, but actually a shrinkable forest of edges. This technique, whose explanation goes beyond this exposition, but is precisely described in [NOI94], is a key factor in making an efficient implementation of the algorithm. Our implementation literally follows the description in [NOI94].

In the computational experiments in [NOI94] the authors observe that combining the strengths of their new algorithm and the Padberg-Rinaldi algorithm leads to a hybrid algorithm with very interesting performance. This algorithm applies the tests for the Padberg-Rinaldi conditions to all edges in the star of the last supernode resulting from contracting all edges in the shrinkable forest.

Of the variations published in the literature, we mention the one of Stoer and Wagner [SW94], who observe that the last two visited nodes in each major iteration are shrinkable (rather than the last edge in the Nagamochi and Ibaraki algorithm). They do not use the concept of shrinkable forests, but obtain the same asymptotic running time. For comparison, we also include this variant in our computational study. Similar variants have been proposed by Frank [Fra94] and Fujishige [Fuj94].

In 1992, Hao and Orlin [HO92, HO94] published a variant of the Goldberg-Tarjan (s,t)-max-flow-min-cut algorithm that computes a minimum capacity cut in the same asymptotic running time $\mathrm{O}(|V||E|\log(|V|^2/|E|))$ and space $\mathrm{O}(|E|)$ as the original. This is a gain of $\Theta(|V|)$ in terms of running time. The basic concept of the Hao-Orlin algorithm is very similar to the naive algorithm. An important difference is that instead of computing an (s,t)-max-flow-min-cut between two arbitrary nodes s and t in each iteration, except the first one, the node s is the supernode originating from the shrinking operation of the previous iteration. The $|V|-1$ (s,t)-max-flow-min-cuts are computed by a modification of the Goldberg-Tarjan algorithm. For every application of the Goldberg-Tarjan algorithm, the distance labels are initialized with the distance labels of the previous run. Moreover, Hao and Orlin also specify the way the node t has to be selected in every iteration by splitting the node set in *awake* and several classes of *dormant* nodes. These modifications yield an algorithm with the same asymptotic running time as a single call of the Goldberg-Tarjan algorithm. It should be noted that the shrinking of the nodes in every iteration does not have to be done explicitly. To achieve a running time of $\mathrm{O}(|V||E|\log(|V|^2/|E|))$, a dynamic tree data structure would have to be implemented. To our knowledge, it has not been investigated so far if the use of this data structure in the Hao-Orlin algorithm also pays in practical computations. It has turned out in [BB93] that the Goldberg-Tarjan maximum flow algorithm can benefit from the dynamic tree data structure for certain specially created instance types, but in general it seems to be better to use highest-level pushing. Our implementation of the Hao-Orlin algorithm uses highest-level pushing and requires $\mathrm{O}(|V|^2\sqrt{|E|})$ running time.

A randomized algorithm that finds a minimum capacity cut with high probability was introduced in 1993 by Karger and Stein [KS96]. This algorithm can be concisely specified as follows:

```
algorithm Karger-Stein $(G,\overline{\lambda},\overline{W})$
if $|V|=2$ ($V=\{u,v\}$, $E=\{\{u,v\}\}$) and $c(\{u,v\})<\overline{\lambda}$ then
  $\overline{\lambda}=c(\{u,v\})$;
  $\overline{W}$ = set of original nodes in supernode $u$;
else
  repeat twice
    $threshold$ = $\left\lceil\frac{|V|}{\sqrt{2}}\right\rceil$;
    while $|V|>$ $threshold$ do
      choose edge $\{u,v\}$ with probability proportional to $c(\{u,v\})$;
      shrink $u$ and $v$ into supernode $s$;
      if $c(\delta(s))<\overline{\lambda}$ then update
        $\overline{\lambda}=c(\delta(s))$;
        $\overline{W}$ = set of the original nodes in supernode $s$;
    Karger-Stein $(G,\overline{\lambda},\overline{W})$;
```

The possible update after the shrinking operation is not part of the original algorithm, but added in our implementation. We initialize $\overline{\lambda}$ and $\overline{W}$ like in the Padberg-Rinaldi implementation. Shrinking operations are interpreted as changing G in the way specified in Figure 1. In 1993, Karger and Stein showed that this algorithm can be implemented to run in $O(|V|^2 \log |V|)$ time and $O(|V|^2)$ space. A particular minimum capacity cut is found by this "recursive randomized shrinking algorithm" with probability $\Omega(1/\log |V|)$. Karger and Stein propose to make $\log^2 |V|$ independent runs of this algorithm in order to find a (all) minimum capacity cut(s) with high probability, such that the overall procedure runs in $O(|V|^2 \log^3 |V|)$ time and $O(|V|^2)$ space.

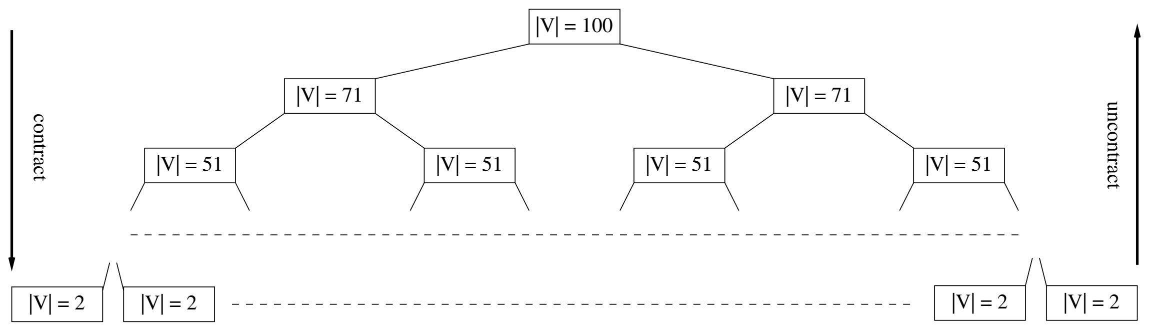


Figure 2. Contraction and uncontraction in the Karger-Stein algorithm

Figure 2 visualizes a run on a graph G with 100 vertices. One way of avoiding the $\Theta(|V|^2|$ space requirements in the conference version of [KS96] of 1993 is to traverse the recursion tree "from left to right", recording and undoing the shrinking operations using a stack. As we have noted in Section 1, undoing a shrinking operation takes no more time than performing it, and this is possible with our data structure, since unshrinking operations occur in reverse order with respect to the corresponding shrinking operations. Thus, our implementation of the Karger-Stein algorithm takes only $O(|E|)$ space. Independently, Karger and Stein also give a linear space variant of their algorithm in the journal version of their article [KS96]. Karger gave another randomized minimum capacity cut algorithm in [Kar96] that is not included in our study. It was implemented and tested in [CGKLS97].

3. The implementations

Asymptotic worst case running times suggest the superiority of all three main algorithmic ideas that have been surveyed in Section 2 in comparison to the Padberg-Rinaldi approach. It was quite a surprise that our experimental results give a rather different picture.

Before we discuss our experimental results in detail, we would like to discuss the possible shortcomings of our experimental evaluation. It is a matter of active discussion in the scientific community, equally in mathematics, computer science, and operations research, how fair computational experiments can be conducted. How much freedom does an implementor of an algorithm have, for better or for worse? Who guarantees that the implementor is not biased, putting a lot of effort into the implementation of his or her "favorite" algorithms, but little in the less "beloved"?

We believe that the only way out is the publication of the computer program implementing the algorithm. In a way, this is, in fact, the only reasonable way to specify an algorithm precisely, anyway. For a good reason, the scientific literature does not go that way, because of immense space consumption and, often, boring contents. As algorithm designers, we like to be given algorithmic ideas and not the details. Sometimes, though, details are interesting enough to be published anyway, because many people actually *have* problem instances and want to *use* the algorithms in practical computation. Even the traditional way of publishing in the scientific literature (books and journal articles) have made this possible.

One of the leading advocates of this form of publishing algorithms as well documented computer programs is Donald E. Knuth, who did so, among other algorithms, for the TEX typesetting system [Knu86] and a collection of combinatorial algorithms in the Stanford GraphBase [Knu93]. We adopted his "literate programming" style for our "minimum capacity cut" project. All our algorithms were written in CWEB [KL93] and will be made available to the scientific community. The reason for not doing this right now is solely a matter of approvable documentation, not the algorithms themselves which we believe to have been implemented well enough to make the computational comparison fair. This claim can be disputed by anyone by reading our programs that will be published in a suitable form [JRT98b]. The `C`-code of our programs (extracted from [JRT98b]) is available in [JRT98a]. With this article we hope to attract the reader's attention to our implementations.

The literate programming style realized in CWEB makes it easy to implement crucial parts in a hierarchy of macros that can be reused whenever appropriate. In our project, such crucial parts include the graph data structure and code fragments for manipulating it. Especially important are the shrinking and unshrinking operations, that are specified once and then used whenever such an operation is performed in any of the implemented algorithms. Thus the running time of the particular algorithm is not influenced by a specific implementation of such a basic step. Moreover, all algorithms use the same priority queue implementation (if they do), the same update procedure for the incumbent solution, etc.

In order to give a flavor of our implementations, we display verbatim some examples of sections in our CWEB-implementation. Figure 3 shows the main loop of the naive algorithm, Figure 4 shows the main loop of the Padberg-Rinaldi algorithm, and Figure 5 shows the main loop of the Nagamochi-Ono-Ibaraki algorithm.

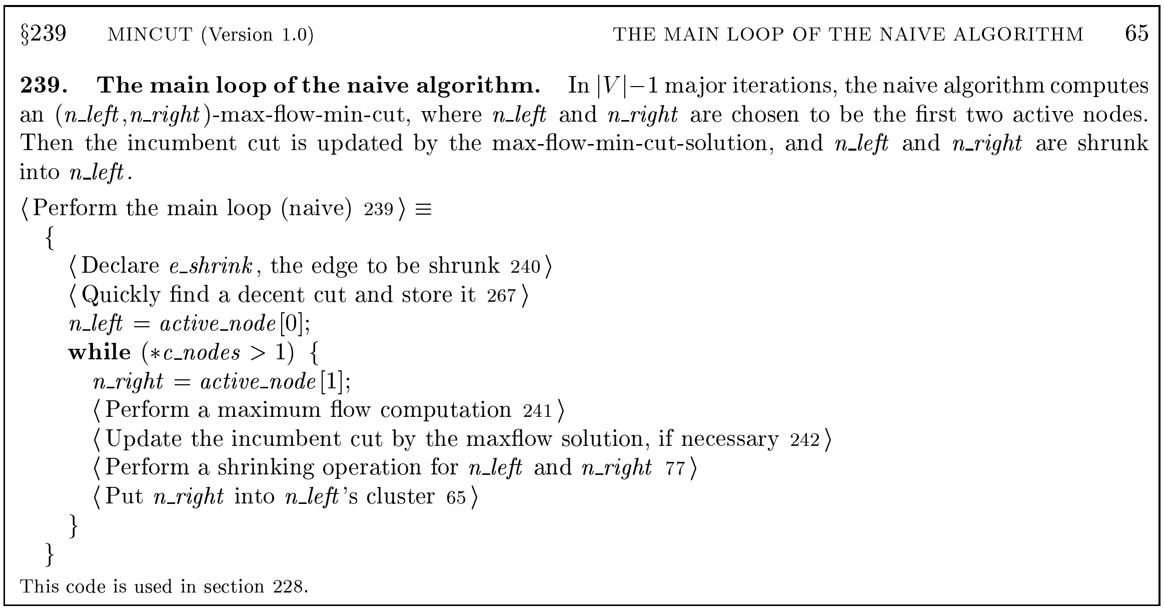
The figures suggest that it is not hard to engineer new algorithms using the pieces of our software package. For example, implementing the hybrid version of the Nagamochi-Ono-Ibaraki algorithm was an easy task, once the Nagamochi-Ono-Ibaraki algorithm and the Padberg-Rinaldi algorithm had been implemented.

The following implementations, whose high level descriptions we gave in Section 2, have been tested in our computational experiments:

NA: the naive algorithm

GH: the Gomory-Hu algorithm in Gusfield's implementation

PR: the Padberg-Rinaldi algorithm


§239 MINCUT (Version 1.0) THE MAIN LOOP OF THE NAIVE ALGORITHM 65

239. The main loop of the naive algorithm. In $|V|-1$ major iterations, the naive algorithm computes an (*n_left*,*n_right*)-max-flow-min-cut, where *n_left* and *n_right* are chosen to be the first two active nodes. Then the incumbent cut is updated by the max-flow-min-cut-solution, and *n_left* and *n_right* are shrunk into *n_left*.

```
⟨Perform the main loop (naive) 239⟩ ≡
  {
    ⟨Declare e_shrink, the edge to be shrunk 240⟩
    ⟨Quickly find a decent cut and store it 267⟩
    n_left = active_node[0];
    while (*c_nodes > 1) {
      n_right = active_node[1];
      ⟨Perform a maximum flow computation 241⟩
      ⟨Update the incumbent cut by the maxflow solution, if necessary 242⟩
      ⟨Perform a shrinking operation for n_left and n_right 77⟩
      ⟨Put n_right into n_left's cluster 65⟩
    }
  }
```

This code is used in section 228.

Figure 3. The main loop of the naive algorithm

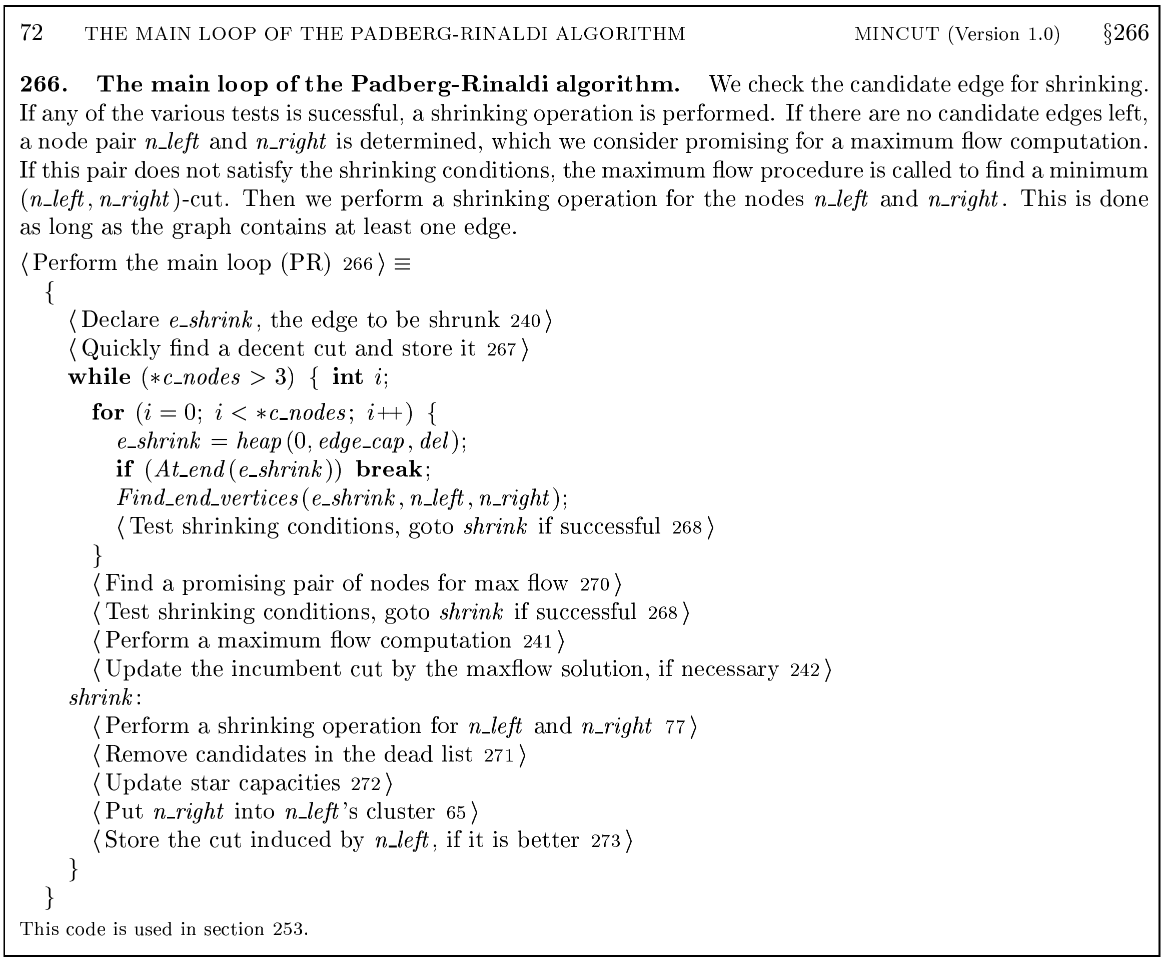
NOIBH:	the Nagamochi-Ono-Ibaraki algorithm with a priority queue based on a binary heap
NOIFH:	the Nagamochi-Ono-Ibaraki algorithm with a priority queue based on a Fibonacci heap
NOIHY:	the Nagamochi-Ono-Ibaraki-Hybrid algorithm
SWBH:	the Stoer-Wagner algorithm with a priority queue based on a binary heap
SWFH:	the Stoer-Wagner algorithm with a priority queue based on a Fibonacci heap
HO:	the Hao-Orlin algorithm
KS:	the Karger-Stein algorithm

The purpose of including two versions of NOI and SW is to test the "folklore" assumption that Fibonacci heaps are better in theory (and their use is required in the asymptotic analysis of both algorithms), but binary heaps are better in practice.

All implementations except GH and PR also come with the suffix "-PREP". This means that, in a initial preprocessing step, the Padberg-Rinaldi algorithm is run to the point where it would perform the first max-flow-min-cut computation. In some cases, drastic problem size reductions result.

4. The experiments

We performed the experiments on a Siemens Nixdorf SCENIC Pro M6 personal computer under the LINUX 2.0.27 operating system. The machine is equipped with a Pentium Pro Processor (200 MHz) with 256 kilobytes of cache memory. The `C`-code obtained from



72 THE MAIN LOOP OF THE PADBERG-RINALDI ALGORITHM MINCUT (Version 1.0) §266

266. The main loop of the Padberg-Rinaldi algorithm. We check the candidate edge for shrinking. If any of the various tests is sucessful, a shrinking operation is performed. If there are no candidate edges left, a node pair *n_left* and *n_right* is determined, which we consider promising for a maximum flow computation. If this pair does not satisfy the shrinking conditions, the maximum flow procedure is called to find a minimum (*n_left*, *n_right*)-cut. Then we perform a shrinking operation for the nodes *n_left* and *n_right*. This is done as long as the graph contains at least one edge.

```
⟨ Perform the main loop (PR) 266 ⟩ ≡
  {
    ⟨ Declare e_shrink, the edge to be shrunk 240 ⟩
    ⟨ Quickly find a decent cut and store it 267 ⟩
    while (*c_nodes > 3) { int i;
      for (i = 0; i < *c_nodes; i++) {
        e_shrink = heap(0, edge_cap, del);
        if (At_end(e_shrink)) break;
        Find_end_vertices(e_shrink, n_left, n_right);
        ⟨ Test shrinking conditions, goto shrink if successful 268 ⟩
      }
      ⟨ Find a promising pair of nodes for max flow 270 ⟩
      ⟨ Test shrinking conditions, goto shrink if successful 268 ⟩
      ⟨ Perform a maximum flow computation 241 ⟩
      ⟨ Update the incumbent cut by the maxflow solution, if necessary 242 ⟩
    shrink:
      ⟨ Perform a shrinking operation for n_left and n_right 77 ⟩
      ⟨ Remove candidates in the dead list 271 ⟩
      ⟨ Update star capacities 272 ⟩
      ⟨ Put n_right into n_left's cluster 65 ⟩
      ⟨ Store the cut induced by n_left, if it is better 273 ⟩
    }
  }
```

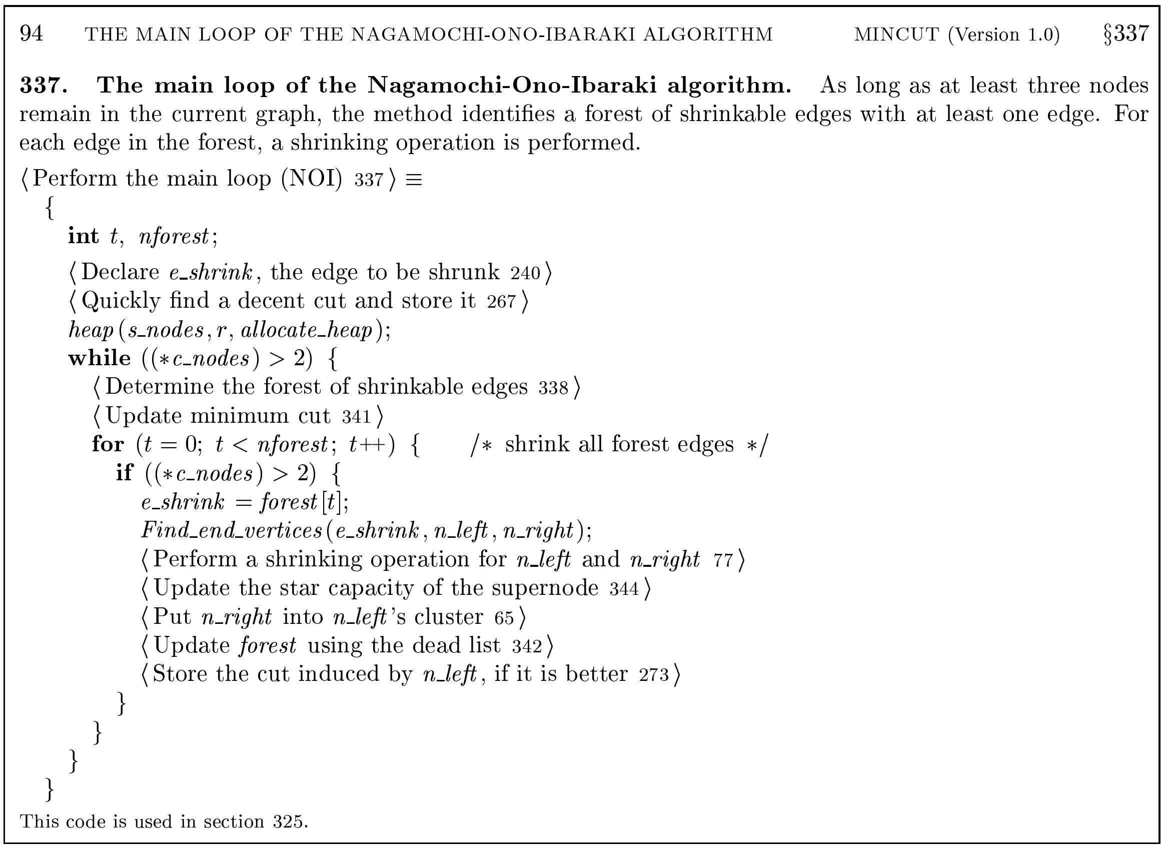
This code is used in section 253.

Figure 4. The main loop of the Padberg-Rinaldi algorithm

running `ctangle` on our CWEB files was compiled using the GNU `C`-compiler version 2.7.2 with optimization option `-O3`.

We are aware of the fact that the computation times that we are going to report depend heavily on the hardware and the software environment. For example, we repeated the experiments that we report on in the following on a SUN SPARCstation Ultra-I with 512 kilobytes of cache memory under the SOLARIS 2.6 operating system, also with GNU `C`-compiler version 2.7.2 with optimization option `-O3`, on a smaller set of instances. We found that the quotient of the running times varies between 1 and 2 in favor of the personal computer. As we are using a nowadays rather typical environment, even the raw computation times may give useful information. Since we publish the implementations of the algorithms and the problem instances [JRT98a,b], it is easy to perform the same experiments in a different environment, plus own experiments, possibly with own (hybrid) algorithms assembled from the pieces available in the package.

A matter of dispute in making experimental computational comparisons is the choice of data sets. The operations research community is well off having public libraries of instances for, e.g., linear programming [Gay85], (mixed) integer programming [BBI92],



94 THE MAIN LOOP OF THE NAGAMOCHI-ONO-IBARAKI ALGORITHM MINCUT (Version 1.0) §337

337. The main loop of the Nagamochi-Ono-Ibaraki algorithm. As long as at least three nodes remain in the current graph, the method identifies a forest of shrinkable edges with at least one edge. For each edge in the forest, a shrinking operation is performed.

```
⟨Perform the main loop (NOI) 337⟩ ≡
  {
    int t, nforest;
    ⟨Declare e_shrink, the edge to be shrunk 240⟩
    ⟨Quickly find a decent cut and store it 267⟩
    heap(s_nodes, r, allocate_heap);
    while ((*c_nodes) > 2) {
      ⟨Determine the forest of shrinkable edges 338⟩
      ⟨Update minimum cut 341⟩
      for (t = 0; t < nforest; t++) {      /* shrink all forest edges */
        if ((*c_nodes) > 2) {
          e_shrink = forest[t];
          Find_end_vertices(e_shrink, n_left, n_right);
          ⟨Perform a shrinking operation for n_left and n_right 77⟩
          ⟨Update the star capacity of the supernode 344⟩
          ⟨Put n_right into n_left's cluster 65⟩
          ⟨Update forest using the dead list 342⟩
          ⟨Store the cut induced by n_left, if it is better 273⟩
        }
      }
    }
  }
```

This code is used in section 325.

Figure 5. The main loop of the Nagamochi-Ono-Ibaraki algorithm

traveling salesman [Rei91], or quadratic assignment [BKR94]. Unfortunately, there is not yet such a test set for the minimum capacity cut problem.

Moreover, a potential user of a minimum capacity cut algorithm might be interested in problem instances which have not been investigated so far. What algorithm should she/he choose for her/his application? This decision turns out to be very difficult if her/his instances differ completely in structure and size from the ones examined so far.

Therefore, we decided not to define artificial problem instances that are likely to be of no interest to the community. Rather, we conduct experiments on the family of problem types used in [NOI94]. We believe that these instance classes give a good overview of the performance of the algorithms on graphs with interesting characteristics. However, we are mostly interested in the performance of the algorithms on problem instances that arise in the separation algorithms mentioned in the introduction. Unfortunately, all instances in the computational studies performed in [Sto92], [Mut95], and [AJR98] are too small (up to around 100 nodes) to be interesting. We did some testing on the instances arising in [Mut95], but even the slowest algorithms solved them in around 0.01 seconds. However, instances arising in the separation of subtour elimination inequalities in the TSP are big enough to be interesting. Therefore, we chose a series of instances of this type as our second test set. We believe that this instance type is the most interesting challenge for minimum capacity cut algorithms from a practical point of view.

We start by repeating the experiments published in [NOI94]. These random instances are specified by the following parameters:

n	$= \|V\|$ the number of nodes,
d	density of edges in %: $m = \|E\| = \frac{n(n-1)}{2} \times \frac{d}{100}$,
k	decomposition number ($1 \leq k \leq n$),
p	scaling factor ($0 < p \leq 1$) for $k \geq 2$.

The edge weights $c(\{u, v\})$ are generated as pseudo random numbers by the program `gb_unif_rand()` of the Stanford GraphBase independently and uniformly in the interval [0,100). Using the same pseudo random generator, a node is assigned to cluster C ($1 \leq C \leq k$) with probability $\frac{1}{k}$. Then, for all edges $\{u, v\}$ such that u and v are in different clusters, we change $c(\{u, v\})$ to $p \times c(\{u, v\})$. All capacities are given with an accuracy of six decimal digits after the point, all calculations involving capacities are carried out in the `C double` format.

[NOI94] contains six experiments that we refer to as NOIEXP1,...,NOIEXP6. Our first experiment (Figure 6) repeats NOIEXP1 for all tested algorithms without preprocessing. We found that in all [NOI94] experiments except NOIEXP5 preprocessing does not change the graphs at all. The plotted running times are averages over 10 samples for NA, GH, SWBH, SWFH, and KS and over 100 samples for PR, NOIBH, NOIFH, NOIHY, and HO. So Figure 6 is based on a total of 3480 individual runs. Here, and in all other experiments, KS is run only once rather than $\log^2 |V|$ times. Nevertheless, it did not fail once to compute a correct solution in all [NOI94] experiments but NOIEXP2 and NOIEXP4. The ranking in Figure 6 with respect to decreasing running time is NA, GH, SW, KS, HO, PR, NOI, NOIHY.

It is clear at the outset that NA and GH cannot seriously compete with the other algorithms, and that SW is always outperformed by NOI, but we have verified this anyway for all [NOI94] experiments. Therefore, we plot in Figures 7, 8, 9, 10, 11, 11a, and 12 only the running times of KS, HO, PR, NOI, and NOIHY. These figures are based on 10 samples for each of the algorithms. Since NOI comes in two versions, we had to perform a total of 3480 individual runs. In the plots, we do not distinguish between the binary heap and Fibonacci heap versions, because there is no visible difference in all [NOI94] experiments. It is, however, true that for the big and dense instances we tested, NOIFH and SWFH are indeed (very slightly) faster than NOIBH and SWBH, respectively. In contrast to [NOI94], the number of nodes is $n = 1000$, unless otherwise specified in NOIEXP1 and NOIEXP2. The captions explain the experiments. In Figure 7, the experiment of Figure 6 is repeated with bigger instances. Table 1 gives evidence why the three best algorithms perform so well. The column headers mean:

n:	number of nodes
time:	cpu-time in seconds
#mf:	number of (s, t)-max-flow-min-cut computations
#pr1:	number of successes in testing PRCOND1
#pr2:	number of successes in testing PRCOND2
#pr3:	number of successes in testing PRCOND3
#pr4:	number of successes in testing PRCOND4

#fail:	number of failures on all PRCONDs
#mi:	number of major iterations
#prs:	number of shrinkings due to successful testing of PRCONDs

The typical behavior of PR appears to be the following. In the beginning, all tests fail on the $|V|$ edges with highest capacity. Then one (s,t)-max-flow-min-cut computation is performed on the original graph, and afterwards the PRCONDs 1–3 are sufficient to shrink the graph down to two nodes. (Of course, for each individual run, we have #mf + #pr1 + #pr2 + #pr3 + #pr4 = $|V| - 3$. Since averages are taken and the results are truncated to integers, the numbers given in the tables may add up to slightly less than $|V| - 3$.) In NOI, the concept of shrinkable forests is crucial for its practical efficiency. It reduces the number of major iterations, that would be $n - 1$ without the forest concept, e.g. in the case of SW, to 3.4–6.7. In NOIHY, the additional PR element makes this reduction even more drastic. In our experiments roughly 10%–30% of the $n - 1$ shrinkings are due to the forest conditions, the remaining 90%–70% to successful PRCOND tests. As the authors of [NOI94] remark, a minimum capacity cut can be expected to be attained at the star of a single node in the NOIEXP1 experiments.

In the NOIEXP2 experiment, whose results are displayed in Figure 8 and Table 2, a minimum capacity cut separating the node set in roughly two halves can be expected (and this is also empirically true). KS (one run!) fails for about 5% of all instances. As expected, KS is not affected much by the different decomposition number in comparison to NOIEXP1. (The running time of KS only depends on $|V|$ and $|E|$, not on the edge capacities.) Also the behavior of HO is very much the same as in NOIEXP1. PR performs slightly better than in NOIEXP1. This is obviously due to the fact that there is still only one (s,t)-max-flow-min-cut computation in the beginning, but afterwards the cheap test for PRCOND1 is more often successful, so that the work on the more expensive later tests is reduced. NOI is faster due to less major iterations and NOIHY behaves essentially like in NOIEXP1.

In the NOIEXP3 experiment, the edge density varies from 5% to 100%. The results are displayed in Figure 9 and Table 3. HO outperforms PR at small densities and NOI at high densities. Beyond 50% density, PR becomes better than NOI. A look a Table 3 explains this. Beyond 50% density, the number of (s,t)-max-flow-min-cut computations reduces to 1 on the average, and the number of failing PRCOND tests reduces drastically. The average number of major iterations of NOI grows monotonically from 2.0 for 5% density to 19.2 for 100% density. NOIHY stays at one major iteration in each run.

At this point, we should mention similarities and differences in comparison to the computational study in [NOI94]. For NOIEXP1 and NOIEXP2, our results are in agreement, except that in [NOI94] no difference between NOI and NOIHY was observed. We believe that our revised view is due to our better implementation of the PR algorithm. In [NOI94], an old FORTRAN implementation of Padberg and Rinaldi was used. The old version gave somewhat strange results in the NOIEXP3 experiments that we could not observe with our implementation. Otherwise, our results on the performance of NOI and NOIHY are in agreement with [NOI94].

The NOIEXP4 results are displayed in Figure 10 and Table 4. This is a repetition of NOIEXP3 with decomposition number 2. As expected, KS behaves essentially as in the NOIEXP3 experiment. KS fails in about 10% of the runs with density 25%, about 20% of the runs with density 37.5%, and about 5% of the runs with density 50%. The performance of HO in comparison to NOIEXP3 is much better, HO outperforms PR over the whole range. The number of (s,t)-max-flow-min-cut computations in PR drops to one from 50% density on, NOI maintains roughly 3 major iterations over the whole spectrum, and NOIHY stays at 1 major iteration over the whole spectrum. The difference to the results of [NOI94] is that they could not observe different running times for NOI and NOIHY, and PR behaved strangely. Our explanation is the same as for the NOIEXP3 experiments.

In Figure 11 and Table 5, we display the results of the NOIEXP5 experiment, in which the decomposition number varies from 1 to 200. As expected, KS is not affected by this. HO outperforms PR for decomposition numbers 2,3, and 5, but is significantly outperformed by the three best algorithms for higher decomposition numbers. PR has a peak of running time at $k = 5$, due to a peak in (s,t)-max-flow-min-cut computations of 5, and 2417 failing tests on the average. We have not been able to come up with a satisfying explanation of this phenomenon that was also observed, although in a much more drastic way, by [NOI94] for $k = 3$ (with the old implementation of PR). We are in agreement with [NOI94] that the behavior of NOI and NOIHY is hardly distinguishable. The NOIEXP5 experiment is the only one in which PR-preprocessing helps. Figure 11a gives the results. The PR preprocessing does not produce any reductions until $k = 5$, for $k = 10$ it reduces from 1000 to 415 nodes on the average, for $k = 20$ to 71 nodes, for $k = 50$ to 64 nodes, for $k = 100$ to 35 nodes, and for $k = 200$ to 146 nodes. The PR curve is copied from Figure 11 for reference. There is no significant difference in performance for any of the fast methods from $k = 20$ on.

Figure 12 and Table 6 display the results for the NOIEXP6 experiment. Here, the scaling factor p changes. Again, KS is not affected as expected. For the smallest two scaling factors, HO outperforms PR. PR, NOI and NOIHY are also hardly affected. Except for the strange behavior of the old implementation of PR, this is in total agreement with the [NOI94] experiments.

In all six NOIEXP experiments, NOIHY consistently performed best. This is certainly due to the fact that it almost always needs only one major iteration. More sophisticated internal PR testing can easily be conceived of, however, in this test set this would only cost additional time instead of shortening the overall running time.

Now let us turn to the experiments with "practical" problem instances arising in a branch-and-cut algorithm for the traveling salesman problem (TSP). For the traveling salesman problem instances `lin318`, `gr666`, `pr1002`, `u1432` and `pr2392` of the TSPLIB [Rei91], we generated all minimum capacity cut problem instances that had to be solved during the optimization process by the algorithm of Naddef and Thienel [NT98]. The number in the name of the instance gives the number of nodes of each minimum capacity cut problem instance associated with the TSP problem instance.

In total, the code of [NT98] had to solve 58 minimum cut problem instances for `lin318`, 149 for `gr666`, 88 for `pr1002`, 232 for `u1432`, and 164 for `pr2392`. These instances are very sparse: the number of edges is roughly 1.5 times the number of nodes. Their structural properties are not covered by the types in the NOIEXP experiments.

Figure 13 and Table 7 show the results for all algorithms (without PR preprocessing) we implemented. A total of 6910 runs had to be performed for this experiment. Even though PR performs from 3 to 32 (s, t)-max-flow-min-cut computations on the average, it clearly outperforms all other algorithms. We notice that PRCOND4 is often successful here, whereas it never was in all NOIEXP experiments. NOIHY dominates NOI by far, as the additional internal PR tests reduce the number of major iterations considerably. HO behaves roughly like NOIHY. Here we see a clear difference in running time for the binary/Fibonacci heap versions of NOI and SW, with the binary heap coming out as the clear winner (as folklore predicted). The extremely poor behavior of KS is significant. Moreover, with a single run, it fails in roughly 16% of all runs, but running it more often is out of the question. Finally, we ran all algorithms with PR preprocessing. The results that are based on 5528 runs, are displayed in Figure 14. The problem sizes are reduced from 318 to 20 nodes, from 666 to 46 nodes, from 1002 to 34 nodes, from 1432 to 145 nodes, and from 2392 to 84 nodes, respectively, on the average. This explains the peaks at size 1432. With PR preprocessing, all algorithms except NA and KS are slightly better than PR.

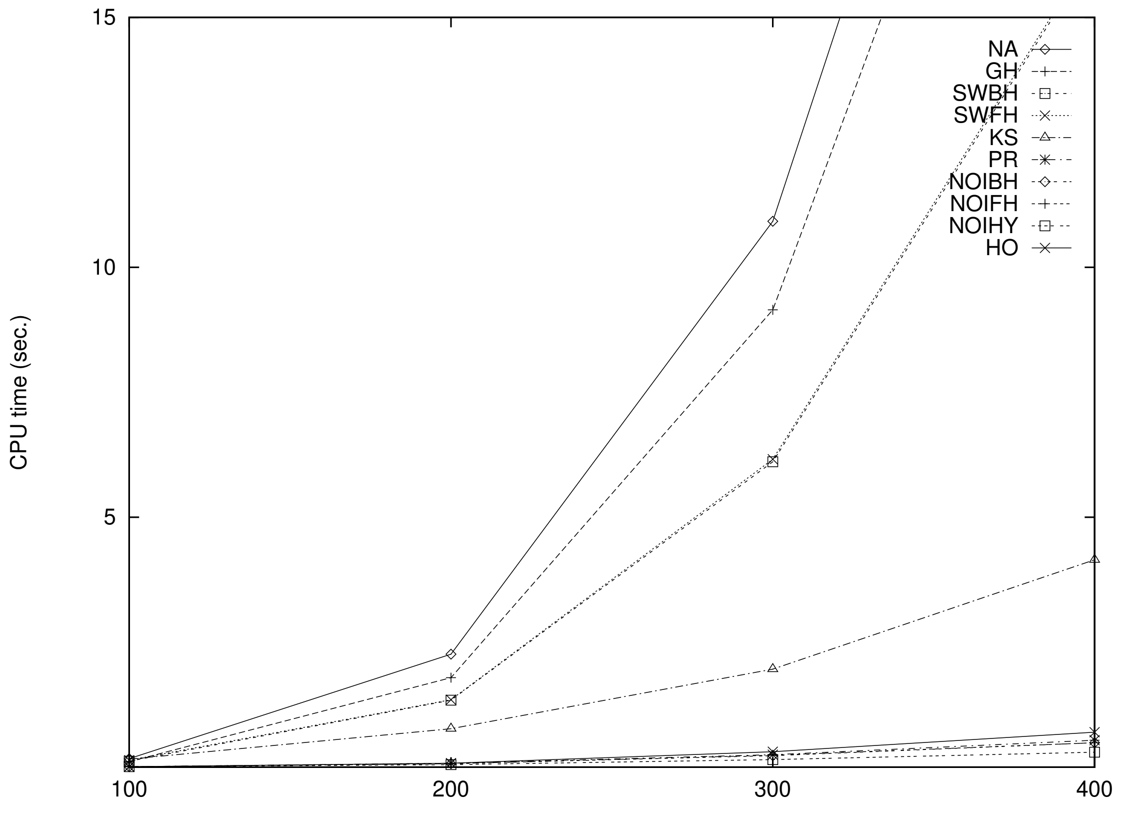


Figure 6. Instances of [NOI94] type ($100 \leq n \leq 400$, $d = 50$, $k = 1$, $p = \frac{1}{n}$)

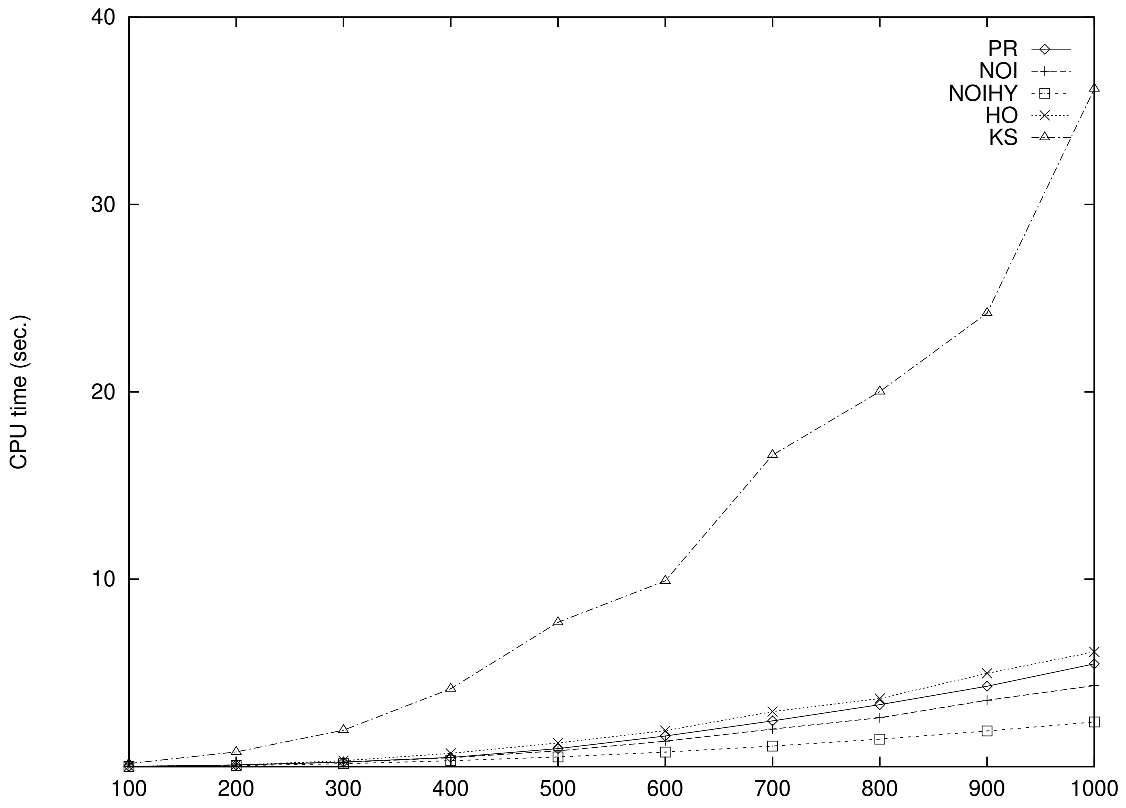


Figure 7. Instances of [NOI94] type ($100 \leq n \leq 1000$, $d = 50$, $k = 1$, $p = \frac{1}{n}$)

	PR							NOI		NOIHY		
n	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
100	0.01	0	24	25	45	0	101	0.01	3.4	0.01	1.0	71
200	0.08	1	38	62	94	0	205	0.08	4.3	0.05	1.0	157
300	0.24	1	51	103	141	0	314	0.23	4.5	0.15	1.0	243
400	0.49	1	64	139	191	0	448	0.47	4.8	0.31	1.0	330
500	0.96	1	65	189	240	0	710	0.84	5.6	0.51	1.0	430
600	1.63	1	67	237	289	0	1087	1.35	5.9	0.77	1.0	529
700	2.44	1	69	290	335	0	1332	2.00	6.6	1.09	1.0	627
800	3.30	1	85	324	385	0	1601	2.61	6.4	1.46	1.0	711
900	4.29	2	85	375	433	0	1804	3.54	6.9	1.90	1.0	810
1000	5.48	1	102	410	481	0	1913	4.32	6.7	2.38	1.0	891

Table 1. Instances of [NOI94] type ($100 \leq n \leq 1000$, $d = 50$, $k = 1$, $p = \frac{1}{n}$)

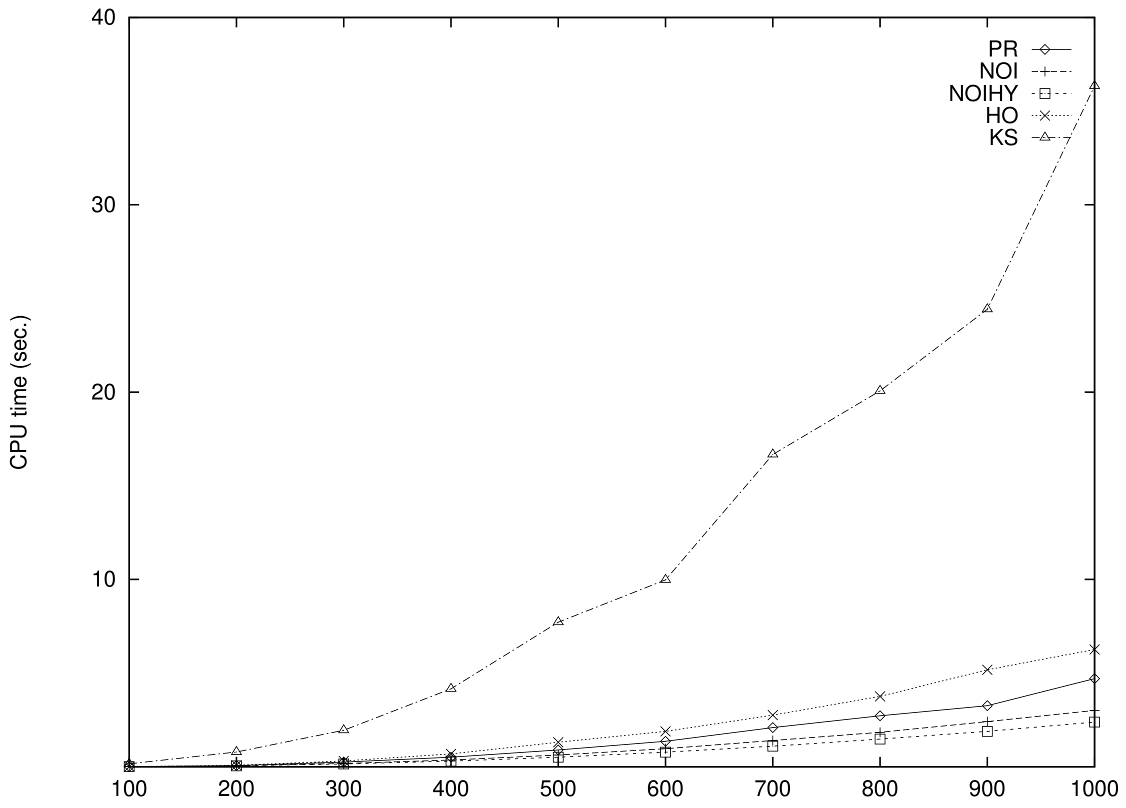


Figure 8. Instances of [NOI94] type ($100 \le n \le 1000$, $d = 50$, $k = 2$, $p = \frac{1}{n}$)

	PR							NOI		NOIHY		
n	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
100	0.01	0	55	5	35	0	4	0.01	2.8	0.01	1.2	40
200	0.07	0	96	13	86	0	141	0.06	3.0	0.05	1.0	116
300	0.25	0	145	14	136	0	301	0.18	3.1	0.15	1.0	197
400	0.51	1	169	38	187	0	401	0.36	3.0	0.30	1.0	278
500	0.90	1	224	39	231	0	503	0.63	3.0	0.51	1.0	381
600	1.36	1	255	58	282	0	625	0.97	3.1	0.78	1.0	469
700	2.09	1	297	69	329	0	918	1.40	3.0	1.10	1.0	575
800	2.72	1	344	73	377	0	954	1.84	3.0	1.48	1.0	638
900	3.26	1	327	139	427	0	1012	2.41	3.0	1.89	1.0	749
1000	4.71	1	417	103	474	0	1405	3.01	3.1	2.39	1.0	830

Table 2. Instances of [NOI94] type ($100 \le n \le 1000$, $d = 50$, $k = 2$, $p = \frac{1}{n}$)

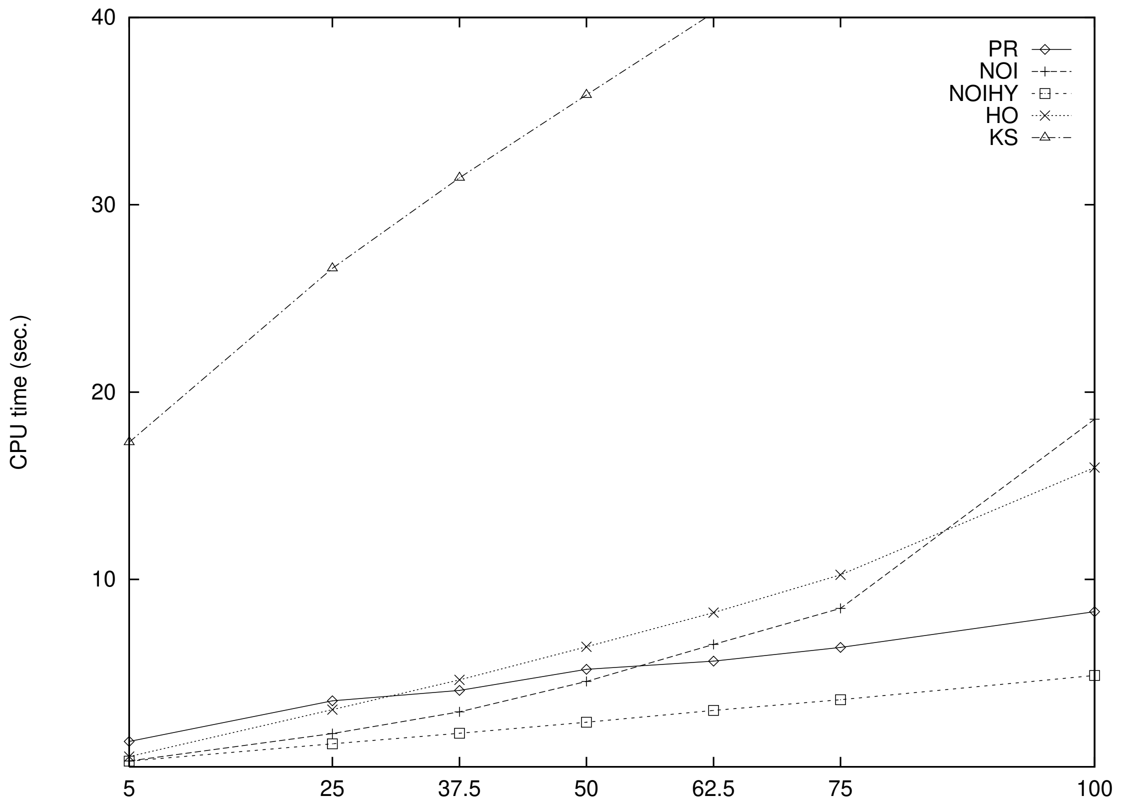


Figure 9. Instances of [NOI94] type ($n = 1000$, $5 \leq d \leq 100$, $k = 1$, $p = \frac{1}{n}$)

	PR							NOI		NOIHY		
d	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
5.0	1.35	6	504	45	439	0	7389	0.31	2.0	0.30	1.0	491
25.0	3.52	2	169	346	477	0	2939	1.77	4.8	1.23	1.0	825
37.5	4.08	2	132	382	480	0	2033	2.93	5.5	1.79	1.0	865
50.0	5.20	2	91	422	481	0	2003	4.56	7.1	2.38	1.0	903
62.5	5.64	1	64	445	485	0	1484	6.53	8.8	3.01	1.0	931
75.0	6.37	1	58	446	491	0	1217	8.46	10.2	3.58	1.0	936
100.0	8.28	1	20	479	496	0	1002	18.55	19.2	4.88	1.0	975

Table 3. Instances of [NOI94] type ($n = 1000$, $5 \leq d \leq 100$, $k = 1$, $p = \frac{1}{n}$)

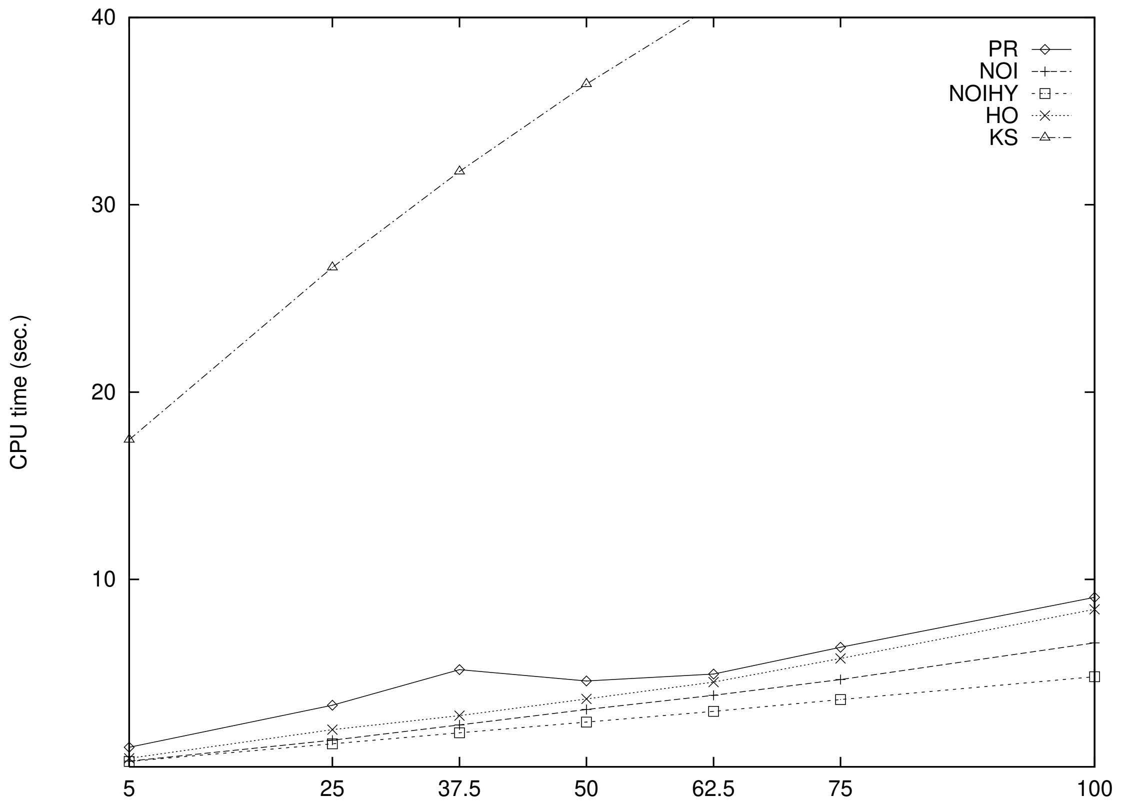


Figure 10. Instances of [NOI94] type ($n = 1000$, $5 \leq d \leq 100$, $k = 2$, $p = \frac{1}{n}$)

	PR							NOI		NOIHY		
d	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
5.0	1.04	5	755	2	233	0	3923	0.30	2.6	0.29	1.0	241
25.0	3.29	3	479	54	459	0	2109	1.42	3.0	1.23	1.0	700
37.5	5.19	2	427	99	467	0	2002	2.24	3.0	1.82	1.0	802
50.0	4.58	1	415	106	473	0	1412	3.06	3.0	2.39	1.0	848
62.5	4.95	1	354	161	480	0	1007	3.82	3.4	2.96	1.0	849
75.0	6.38	1	384	126	484	0	1001	4.66	3.3	3.59	1.0	868
100.0	9.05	1	352	152	490	0	1001	6.62	3.4	4.81	1.0	939

Table 4. Instances of [NOI94] type ($n = 1000$, $5 \leq d \leq 100$, $k = 2$, $p = \frac{1}{n}$)

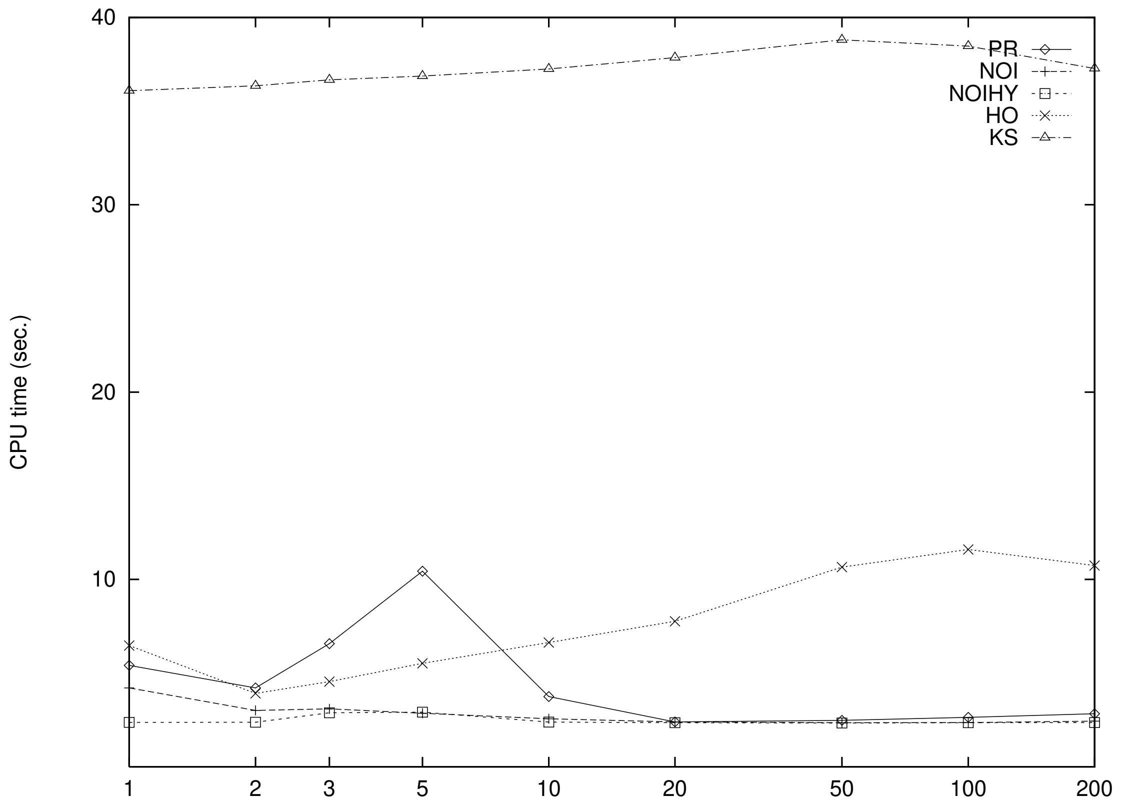


Figure 11. Instances of [NOI94] type ($n = 1000$, $d = 50$, $1 \leq k \leq 200$, $p = \frac{1}{n}$)

	PR							NOI		NOIHY		
k	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
1	5.41	2	105	408	480	0	2007	4.21	6.5	2.37	1.0	889
2	4.21	1	385	135	474	0	1220	3.01	3.0	2.38	1.0	819
3	6.57	3	304	210	478	0	1592	3.09	4.9	2.88	2.0	548
5	10.44	5	318	201	472	0	2417	2.86	6.7	2.92	2.9	235
10	3.75	3	491	58	443	0	514	2.56	4.8	2.39	1.1	456
20	2.40	0	740	5	250	0	24	2.40	3.3	2.35	1.0	256
50	2.48	0	960	2	33	0	0	2.35	2.9	2.34	1.0	69
100	2.64	0	981	3	11	0	0	2.36	2.2	2.35	1.0	91
200	2.83	0	923	17	56	0	0	2.44	2.0	2.36	1.0	212

Table 5. Instances of [NOI94] type ($n = 1000$, $d = 50$, $1 \leq k \leq 200$, $p = \frac{1}{n}$)

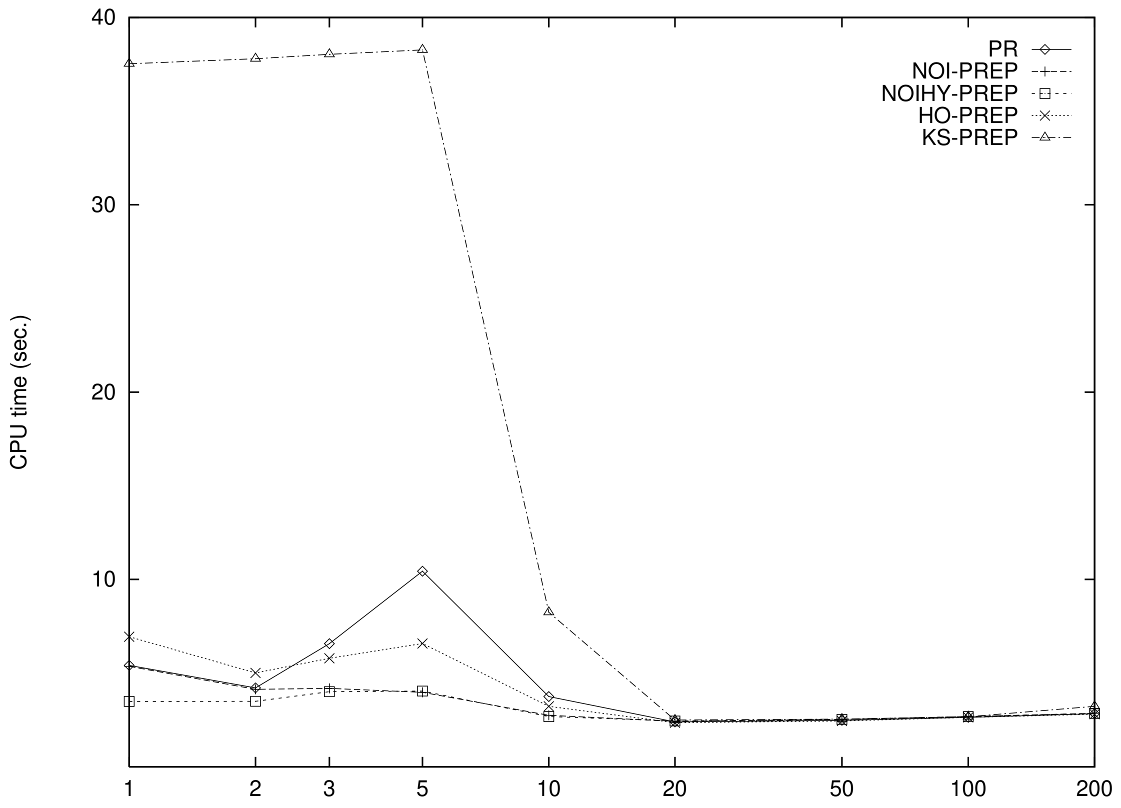


Figure 11a. Instances of [NOI94] type ($n = 1000$, $d = 50$, $1 \leq k \leq 200$, $p = \frac{1}{n}$)

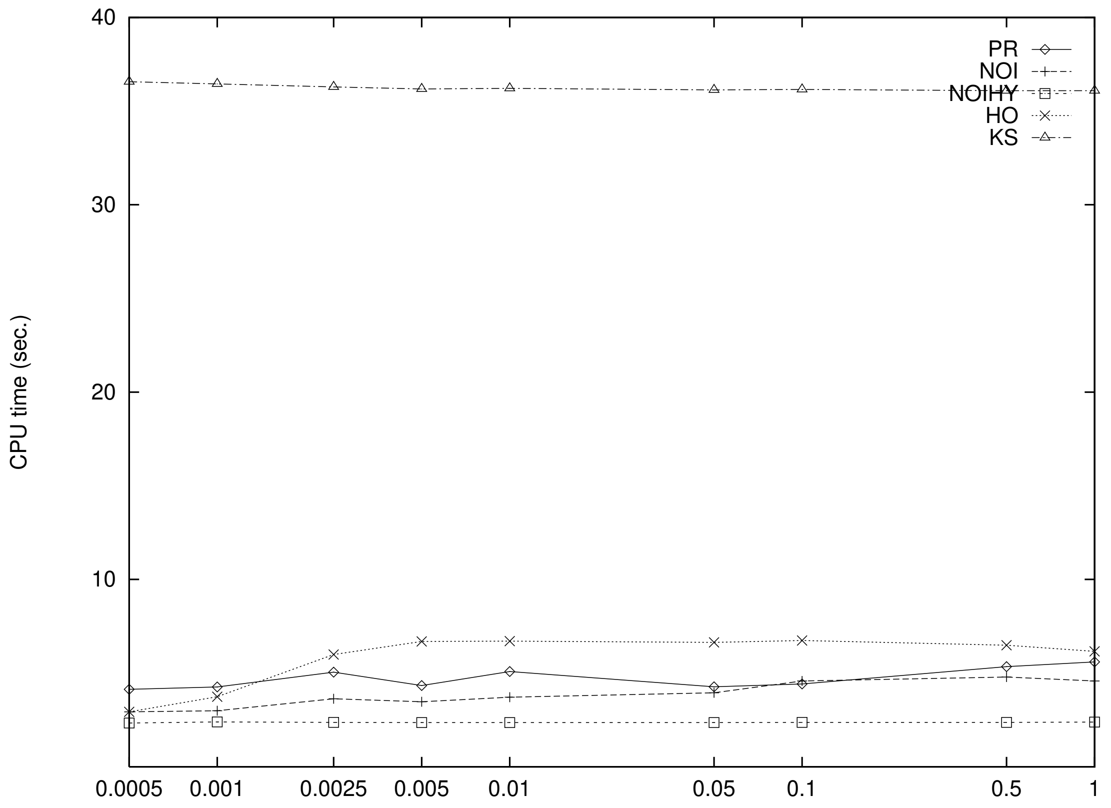


Figure 12. Instances of [NOI94] type ($n = 1000$, $d = 50$, $k = 2$, $0.0005 \leq p \leq 1$)

	PR							NOI		NOIHY		
p	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
0.0005	4.14	1	543	128	323	0	1110	2.93	3.0	2.34	1.0	829
0.0010	4.26	1	394	125	474	0	1307	2.99	3.0	2.40	1.0	813
0.0025	5.05	2	158	357	479	0	1526	3.63	5.9	2.37	1.0	839
0.0050	4.34	1	177	336	482	0	1312	3.47	5.4	2.36	1.0	820
0.0100	5.09	1	145	369	480	0	1620	3.72	5.8	2.36	1.0	849
0.0500	4.27	1	125	388	482	0	1411	3.95	5.8	2.36	1.0	876
0.1000	4.42	1	70	445	479	0	1449	4.58	7.0	2.37	1.0	917
0.5000	5.35	2	84	431	478	0	2004	4.79	7.8	2.37	1.0	904
1.0000	5.60	2	86	427	481	0	2005	4.58	7.1	2.39	1.0	910

Table 6. Instances of [NOI94] type ($n = 1000$, $d = 50$, $k = 2$, $0.0005 \leq p \leq 1$)

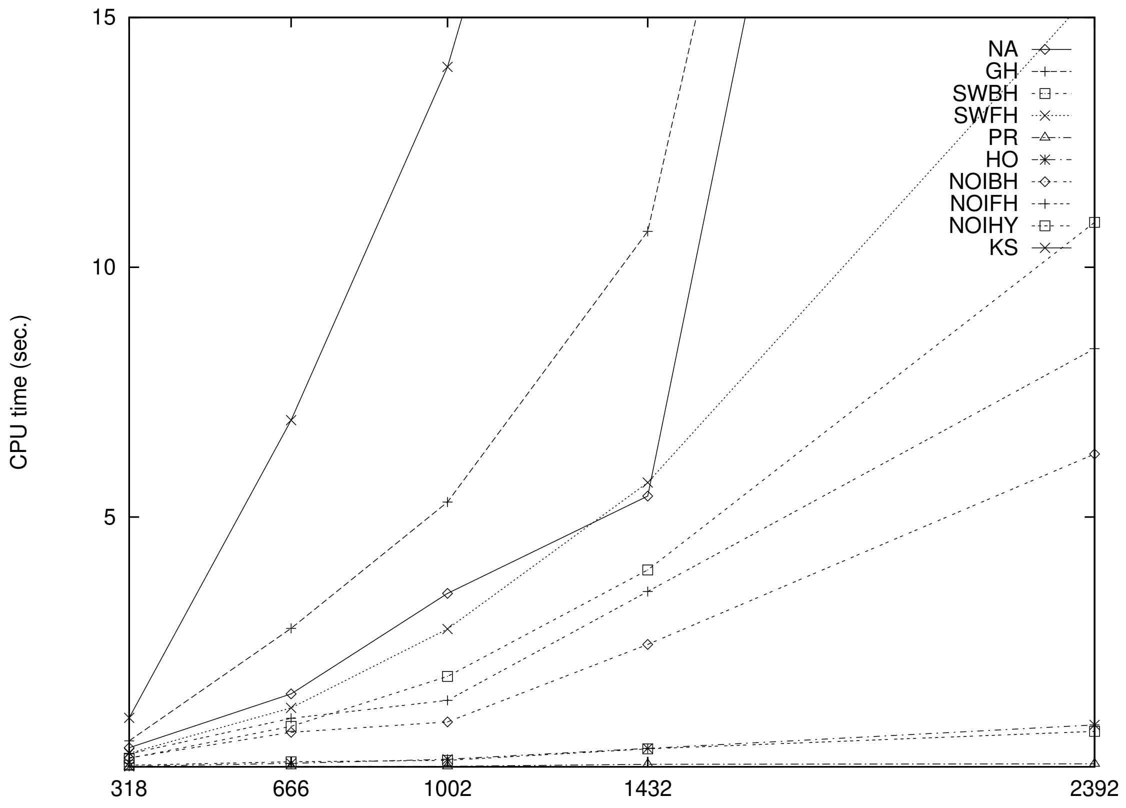


Figure 13. TSP instances without Padberg-Rinaldi preprocessing

	PR							NOI		NOIHY		
n	time	#mf	#pr1	#pr2	#pr3	#pr4	#fail	time	#mi	time	#mi	#prs
318	0.00	3	4	276	0	29	43	0.18	288	0.03	43	254
666	0.01	9	41	556	3	51	108	0.69	521	0.10	64	526
1002	0.01	6	186	737	11	56	86	0.90	500	0.13	57	603
1432	0.05	32	83	1192	14	106	362	2.45	941	0.36	100	1042
2392	0.06	17	531	1704	19	114	209	6.26	1295	0.71	113	1769

Table 7. TSP instances without Padberg-Rinaldi preprocessing

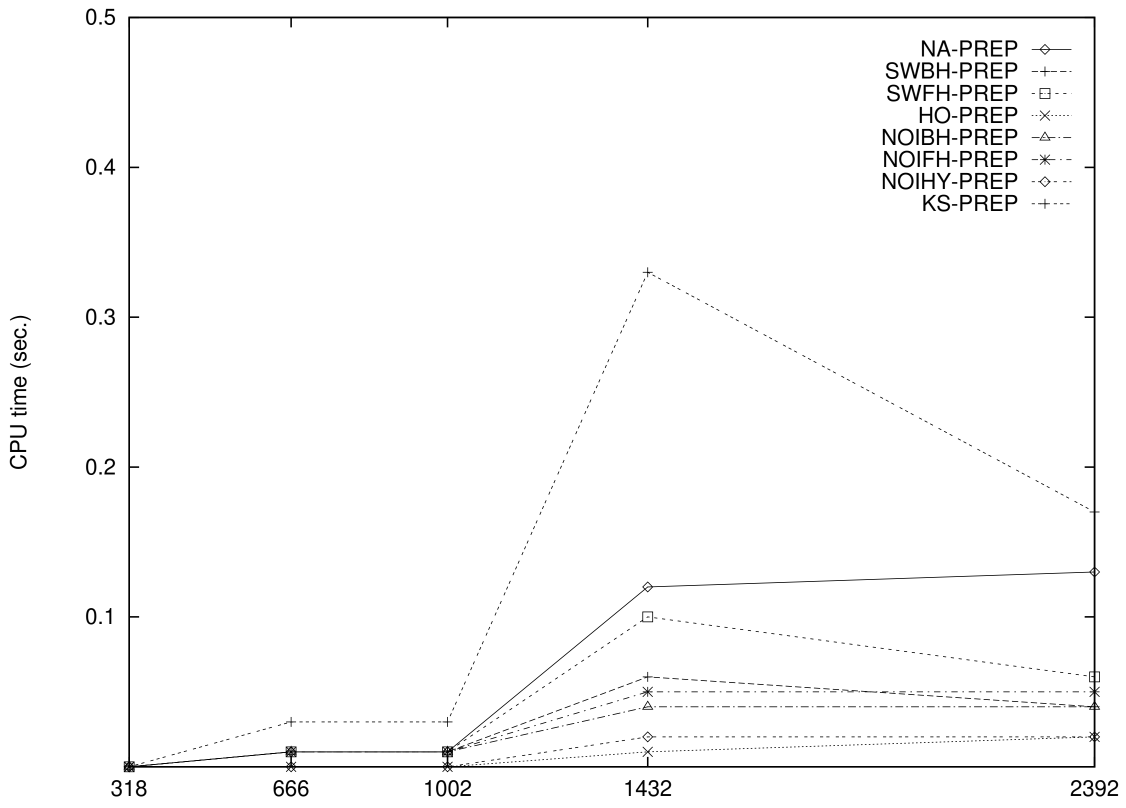


Figure 14. TSP instances with Padberg-Rinaldi preprocessing

5. Conclusions

Our experiments show that the theoretical worst case performance for algorithms solving the minimum cut problem should not be the only guideline for the selection of an algorithm.

There are numerous ways to combine the experimentally evaluated algorithms into hybrid versions, one of which is reported in [NOI94] and many others in [CGKLS97]. Except for NOIHY, we decided to refrain from trying more such experiments, because our only purpose here is to provide a (contemporary) basis for evaluating the practical performance of published minimum capacity cut algorithms. Our experimental experience makes us believe that the "right" combination is highly dependent on structural properties of the considered class of instances. For the TSP-type instances, [CGKLS97] is an excellent source of information, and we cannot identify other interesting classes that make the considerable experimental efforts that would have to be invested, appear worthwhile at the time of writing. Rather, we hope that interested readers with their particular instances will do the appropriate "algorithm engineering" based on our "generic" implementations of minimum capacity cut algorithms.

Acknowledgements

In an earlier version of this article, we had used a version of KS that required $\Theta(|V|^2)$ space as in the conference version of [KS96]. We are grateful to Max Böhm, who pointed out to us that shrinking and unshrinking in reverse order can be done efficiently in $O(|E|)$ space. This led to our current version of KS.

Thanks are due to two anonymous referees, who read an earlier version very carefully and made many valuable suggestions for improvement of this article, almost all of which we gratefully followed.

References

[AJR98] N. Ascheuer, M. Jünger, and G. Reinelt (1998), "A branch and cut algorithm for the asymmetric Hamiltonian path problem with precedence constraints", Technical Report 98.323, Universität zu Köln.

[BB93] T. Badics and E. Boros (1993), "Implementing a maximum flow algorithm: Experiments with dynamic trees", in: D. S. Johnson and C. C. McGeoch (eds.), *Network flows and matching, First DIMACS implementation challenge*, American Mathematical Society, 43–63.

[BBI92] R.E. Bixby, E.A. Boyd, and R.R. Indovina (1992), "MIPLIB: A test set of mixed integer programming problems", *Siam News* , March 1992, 16.

[BKR94] R. Burkhard, S. Karisch, and F. Rendl (1994), "QAPLIB—A quadratic assignment problem library", Technical Report 287, Institut für Mathematik, Technische Universität Graz.

[CG95] B.V. Cherkassky and A.V. Goldberg (1995), "On implementing push-relabel method for the maximum flow problem", in: E. Balas and J. Clausen (eds.), *Proceedings of the 4th international IPCO conference, Lecture Notes in Computer Science 920, Springer, Berlin*, 157–171.

[CGKLS97] C.S. Chekuri, A.V. Goldberg, D.R. Karger, M.S. Levine, and C. Stein (1997), "Experimental study of minimum cut algorithms", in: *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97), New Orleans*, 324–333.

[DM89] U. Derigs and W. Meier (1988), "Implementing Goldberg's max-flow algorithm—a computational investigation", *ZOR—Methods and Models of Operations Research* **33**, 383–403.

[Fra94] A. Frank (1994), "On the edge connectivity algorithm of Nagamochi and Ibaraki", Technical Report, Labaratoire Artemis, IMAG, Université J. Fourier, Grenoble.

[Fuj94] S. Fujishige (1994), "Another simple proof of the validity of Nagamochi and Ibaraki's min-cut algorithm and Queyrannes extension to symmetric submodular function minimization", Technical Report, Forschungsinstitut für Diskrete Mathematik, Universität Bonn.

[Gay85] D.M. Gay (1985), "Electronic mail distribution of linear programming test problems", *COAL Newsletter* **13**, 10–12.

[GH61] R.E. Gomory and T.C. Hu (1961), "Multi-terminal network flows", *SIAM Journal* **9**, 551–570.

[GT88] A.V. Goldberg and R.E. Tarjan (1988), "A new approach to the maximum flow problem", *Journal of the ACM* **35**, 921–940.

[Gus90] D. Gusfield (1990), "Very simple methods for all pairs network flow analysis", *SIAM Journal on Computing* **19**, 143–155.

[HO92] J. Hao and J.B. Orlin (1992), "A faster algorithm for finding the minimum cut in a graph", in: *Proceedings of the third annual ACM-SIAM symposium on discrete algorithms*, 165–174.

[HO94] J. Hao and J.B. Orlin (1994), "A faster algorithm for finding the minimum cut in a directed graph", *Journal of Algorithms* **17**, 424-464.

[JRR95] M. Jünger, G. Reinelt, and G. Rinaldi (1995), "The traveling salesman problem", in: M. Ball, T. Magnanti, C.L. Monma, and G.L. Nemhauser (eds.), *Handbook on Operations Research and Management Sciences, Vol. 7*, North Holland, Amsterdam, 225–330.

[JRT98a] M. Jünger, G. Rinaldi, and S. Thienel (1998), "C-code extracted from [JRT98b]", on the WWW under `http://www.informatik.uni-koeln.de/ls_juenger/projects/mincut.html`.

[JRT98b] M. Jünger, G. Rinaldi, and S. Thienel (1998), "MINCUT—a MINimum CUT algorithm library", software system, to appear.

[Kar96] D.R. Karger (1996), "Minimum cuts in near-linear time", in: G. Miller (ed.), *Proceedings of the 28th ACM Symposium on Theory of Computing, ACM Press*, 56–63.

[KL93] D.E. Knuth and S. Levy (1993), "The CWEB system of structured documentation", Technical Report and Software Package, `ftp://labrea.stanford.edu:/pub/cweb`.

[Knu86] D.E. Knuth (1986), "TEX: The program", Addison-Wesley, Reading, Massachusetts.

[Knu93] D.E. Knuth (1993), "The Stanford GraphBase: A platform for combinatorial computing", Addison-Wesley, Reading, Massachusetts.

[KRT94] V. King, S. Rao, and R. Tarjan (1994), "A faster deterministic maximum flow algorithm", *Journal of Algorithms* **17**, 447–474.

[KS96] D.R. Karger and C. Stein (1996), "A new approach to the minimum cut problem", *Journal of the ACM* **43**, 601–640, a preliminary version appeared in *Proceedings of the 25th ACM Symposium on the Theory of Computing, San Diego, CA, 757–765.*

[Mut95] P. Mutzel (1995), "A polyhedral approach to planar augmentation and related problems", in: P. Spirakis (ed.), *Algorithms—ESA'95,* Lecture Notes in Computer Science 979, Springer, Heidelberg, 494–507.

[NI92] H. Nagamochi and T. Ibaraki (1992), "Computing edge-connectivity in multi-graphs and capacitated graphs", *SIAM Journal on Discrete Mathematics* **5**, 54–66.

[NOI94] H. Nagamochi, T. Ono, and T. Ibaraki (1994), "Implementing an efficient minimum capacity cut algorithm", *Mathematical Programming* **67**, 325–341.

[NT98] D. Naddef and S. Thienel (1998), “Efficient separation routines for the symmetric traveling salesman problem”, Technical Report, Universität zu Köln, to appear.

[PR90] M.W. Padberg and G. Rinaldi (1990), “An efficient algorithm for the minimum capacity cut problem”, *Mathematical Programming* **47**, 19–36.

[Rei91] G. Reinelt (1991), “TSPLIB—A traveling salesman problem library”, *ORSA Journal on Computing* **3**, 376–384.

[Sto92] M. Stoer (1992), “Design of survivable networks”, Lecture Notes in Mathematics 1531, Springer, Heidelberg.

[SW94] M. Stoer and F. Wagner (1994), “A simple min cut algorithm”, in: J. van Leeuwen (ed.), *Algorithms—ESA'94,* Lecture Notes in Computer Science 855, Springer, Heidelberg, 141–147.