Experimental Study of Minimum Cut Algorithms

Chandra S. Chekuri[*] Andrew V. Goldberg[†] David R. Karger[‡]
Matthew S. Levine[§] Cliff Stein[¶]

Abstract

Recently, several new algorithms have been developed for the minimum cut problem. These algorithms are very different from the earlier ones and from each other and substantially improve the worst-case time bounds for the problem. In this paper, we conduct experimental evaluation the relative performance of these algorithms. In the process, we develop heuristics and data structures that substantially improve the practical performance of the algorithms. We also develop problem families for testing minimum cut algorithms. Our work leads to a better understanding of the practical performance of minimum cut algorithms and produces very efficient codes for the problem.

[*]Computer Science Department, Stanford University, Stanford, CA 94305, `chekuri@theory.stanford.edu`.
Supported by NSF Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

[†]NEC Research Institute, Princeton, NJ 08540, `avg@research.nj.nec.com`

[‡]Laboratory for Computer Science, MIT, Cambridge, MA 02139, `karger@theory.lcs.mit.edu`.
Research partly supported by ARPA contract N00014-95-1-1246.

[§]Laboratory for Computer Science, MIT, Cambridge, MA 02139, `mslevine@theory.lcs.mit.edu`.
Research partly supported by a grant from the World Wide Web Consortium. Some of this work was done while visiting the second author at NEC.

[¶]Department of Computer Science, Dartmouth College, Hanover, NH, 03755, `cliff@cs.dartmouth.edu`.
Research partly supported by NSF Award CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award. Some of this work was done while visiting the second author at NEC, and while visiting Stanford University.

1 Introduction

The minimum cut problem is the problem of partitioning the vertices of an n-vertex, m-edge weighted undirected graph into two sets so that the total weight of the set of edges with endpoints in different sets is minimized. This problem has many applications, including network reliability theory [18, 27], information retrieval [4], compilers for parallel languages [5], and as a subroutine in cutting-plane algorithms for the Traveling Salesman problem (TSP) [3].

The problem of finding a minimum capacity cut between two specified vertices, s and t, is called the minimum s-t cut problem, and is closely related to the minimum cut problem. The classical Gomory-Hu algorithm [14] solves the minimum cut problem using $n-1$ minimum s-t cut computations. The fastest current algorithms for the s-t cut problem [1, 6, 7, 13, 21] use flow techniques, in particular the push-relabel method [13], and run in $\omega(nm)$ time. For the minimum cut problem, Hao and Orlin [15, 16] have given an algorithm (`ho`), based on the push-relabel method, that shows how to perform all $n-1$ minimum s-t cuts in time asymptotically equal to that needed to perform one s-t minimum cut computation. This algorithm runs in $O(nm\log(n^2/m))$ time.

Several new algorithms discovered recently are theoretically more efficient – time bounds for these algorithms are competitive with or better than the best time bounds for the minimum s-t cut problem. The algorithm of Nagamochi and Ibaraki [22] (`ni`) runs in $O(n(m+n\log n))$ time. The algorithm of Karger and Stein [20] (`ks`) runs in $O(n^2\log^3 n)$ expected time. Two closely related algorithms of Karger [19] (`k`) run in $O(m\log^3 n)$ and $O(n^2\log n)$

expected time. These algorithms are based on new techniques which do not use flows. These algorithms do not extend to directed graphs, while the flow based algorithms, include **ho**, do.

In this paper we study the practical performance of these recent minimum cut algorithms. The main part of our study is the design of data structures and heuristics for efficient implementations of these algorithms. Our goal is to obtain efficient implementations of the new algorithms and to compare them against each other and against a good previous code. In order to perform meaningful comparisons, we develop problem generators and test families for evaluating and comparing performance of the minimum cut codes. We also run and analyze our algorithms on data that arises during the TSP algorithm of Applegate and Cook [3]. Our problem families are carefully selected and proved very useful for comparing and tuning minimum cut codes.

The first efficient implementation of a minimum cut algorithm that we are aware of is due to Padberg and Rinaldi [25]. In order to reduce the number of maximum flow computations needed by the Gomory-Hu algorithm, they developed a set of heuristics which contract certain edges during the computation. These contractions, which we call the *PR heuristics*, apply in the context of other algorithms and often lead to a big improvement in performance.

Nagamochi et al. [23] describe an efficient implementation of **ni**that uses the PR heuristics. The data of [23] suggests that the **hybrid** code of Nagamochi et al. is more efficient than the Padberg-Rinaldi code. The former code seems to be the fastest minimum cut code described in a paper published prior to our work.

A major contribution of this paper are new heuristics for improving practical performance of the algorithms. We propose a common preprocessing, based on PR heuristic, which takes $O(m \log n)$ time and usually is nearly as helpful as the $\Omega(nm)$ version of Padberg and Rinaldi. We introduce algorithm-specific ways of using PR heuristics during execution of main subroutines of the algorithms. We also develop several new heuristics that significantly improve performance of our implementations.

We now briefly detail the highlights of the implementations of our four codes.

Our implementation of **ni** builds on **hybrid** [23]. However, we use a different strategy for applying the PR heuristics and develop a new technique for graph contraction. As a result, our code is always competitive with **hybrid**, and sometimes outperforms it by a wide margin.

Implementations of the push-relabel method for the maximum flow problem have been well-studied, *e.g.* [2, 8, 10, 11, 24]. A maximum flow code of Cherkassky and Goldberg [8] was the starting point of our implementation of **ho**. The implementation uses the heuristics used in the maximum flow code – global update and gap relabeling. In addition, we use the graph contraction data structures mentioned above, the PR heuristics, and several new heuristics.

The Karger-Stein and Karger's algorithms are randomized. For these algorithms, we needed to develop somewhat different strategies for random edge selection than those that appeared in the original papers [20, 19]. These codes also benefit from the PR heuristics as well as new heuristics.

We make significant progress in understanding practical performance of minimum cut algorithms and in establishing testing standards for future codes. Our study shows that no single algorithm dominates the others. Overall, **ho** is the best code. The second best code is **ni**.

The push-relabel method has been extensively studied, and **ho** takes advantage what has been learned in the maximum flow context. Our code **ni** takes advantage of performance-improving ideas of [23]. Our implementations of **ks** and **k**, however, were developed from scratch. Our study shows how important data structures and heuristics are for the minimum cut algorithm performance. New ideas may improve performance. This is more likely for **ks** and **k**, which have been studied less.

Due to the lack of space (the full paper is about 10 times longer than this version), we omit many descriptions, proofs, and most of the experimental data. The full version can be reached via the second author's home page, URL `http://www.neci.nj.nec.com/homepages/avg.html`, or the fifth author's page URL `http://www.cs.dartmouth.edu/c̃liff/papers/`

2 Background

2.1 Definitions and Notation We assume that the reader is familiar with basic definitions. Let $G = (V, E, c)$ be an undirected graph with vertex set V, edge set E and non-negative real edge capacities $c : E \rightarrow \mathbf{R}^+$. Let $n = |V|$ and $m = |E|$. We will use $\lambda(G)$ to denote the value of the minimum cut of G, and $\lambda_{v,w}(G)$ to denote the value of the minimum v-w cut. Some algorithms we discuss use flows. In this case, we transform an undirected graph into a directed graph in the standard way.

2.2 Data Structures for Graph Contraction Given a graph G and edge $\{v, w\} \in E$, we define $G/\{v, w\}$, the contraction of edge $\{v, w\}$, by deleting node w and replacing each edge of the form $\{w, x\}$ by an edge $\{v, x\}$. Our implementations use graph contraction. In this section we describe the graph data structure used to deal with edge contractions efficiently.

We represent an undirected graph as a symmetric directed graph using the adjacency list representation. An edge $\{u, v\}$ is represented by two arcs, (u, v) and (v, u). These arcs have pointers to each other. An arc (u, v) appears on the adjacency list of u and has a pointer to v

Suppose we contract an edge $\{u, v\}$. One way to implement the contraction is to do *compact contraction* by merging the adjacency lists of u and v, merging parallel arcs, and deleting self-loops. A careful implementation of compact contraction of $\{u, v\}$ takes time proportional to the sum of degrees of u and v before the contraction.

An alternative way to contract an edge $\{u, v\}$ is to use *set-union contraction.* In this implementation, each node in the current graph corresponds to a set of vertices of the input graph. The sets are represented using the disjoint-set union data structure [9]. An edge $\{u, v\}$ is contracted, in constant time, by forming a union of of the vertex sets corresponding to u and v and appending the adjacency list of v to that of u.

The advantage of the set-union contraction is in efficiency of the contraction operation. Its disadvantages come from the parallel arcs and self-loops which remain in the graph and from the increased cost of finding the head node of an arc. Our algorithms will make use of both of these data structures at different times.

2.3 Gomory-Hu Algorithm Gomory and Hu [14] showed that a minimum cut can be computed in $n - 1$ minimum s-t cut computations. The underlying idea is important for understanding other minimum cut algorithms. Consider a pair of distinct vertices s and t and a minimum s-t cut A. Then either $\lambda_{s,t}(G) = \lambda(G)$ or s and t are on the same side of every minimum cut.

2.4 Discovery Times *Discovery time* is the time from the beginning of the computation until the minimum cut is found. The discovery time may be much less then the total running time of an algorithm and are useful for several reasons. For example, they can be used to estimate real-life performance of randomized algorithms. The theoretical upper bound on the number of "trials" required to achieve the desired success probability can can be very pessimistic. If discovery time of a randomized algorithm is always a tiny fraction of the total time, one may be justified in making a smaller number of "trials."

3 Padberg-Rinaldi Heuristics

Recall that the Gomory-Hu algorithm performs a series of maximum flow computations, each of which identifies one edge which can be contracted. The improvements of Padberg and Rinaldi [25] are based on additional tests (*PR tests*) that allow contracting certain nodes without performing any maximum flow computations. Four of the tests are inexpensive enough to be of practical interest.

LEMMA 3.1. ([25]) *Let $\hat{\lambda}$ be an upper bound on $\lambda(G)$. If $v, w \in V$ satisfy either one of the following conditions:*
(PR1) $c(v, w) \geq \hat{\lambda}$,
(PR2) $c(v) \leq 2c(v, w)$ *or* $c(w) \leq 2c(v, w)$,
(PR3) $\exists u$ *such that* $c(v) \leq 2\{c(v, w) + c(v, u)\}$ *and* $c(w) \leq 2\{c(v, w) + c(w, u)\}$,
(PR4) $c(v, w) + \sum_u \min(c(v, u), c(w, u)) \geq \hat{\lambda}$
then one of the following conditions must hold:
(a)$\{v\}$ *or* $\{w\}$ *is a minimum cut*
(c) the edge $\{v, w\}$ *forms the unique minimum cut,*
(d) $\exists$ *a minimum cut* $(A, \overline{A})$ *with* $v \in A$ *and* $w \in A$.

In all our algorithms, we maintain explicitly node capacities $c(v)$, edge capacities $c(v, w)$, and $\hat{\lambda}$, an upper bound on the minimum cut, which corresponds to the smallest capacity cut we have encountered so far. Thus tests PR1 and PR2 can be performed together in $O(1)$ time per edge. For a fixed pair v, w, PR3 and PR4 can be checked in $O(n)$ time. We always apply the two tests together, and compute the sum for PR4 while doing PR3.

The Padberg-Rinaldi tests have been used previously [25, 23]. We propose new ways of using these tests with are both effective and efficient.

3.1 PR Passes We introduce the idea of *PR passes*. All our codes that use PR heuristic apply these passes as preprocessing. ni also applies the passes internally. Each pass takes linear time. *Pass-1-2* combines tests PR1 and PR2 and *pass-3-4* combines tests PR3 and PR4.

Intuitively, pass-1-2 applies the two tests to every edge and every node of the graph. This is not precise, however, because edge contractions change the graph. Our implementation has the invariant that if an edge $\{u, v\}$ was not contracted, and did not become a self-loop and deleted during the pass, then the PR tests failed on $\{u, v\}$ at some point during a pass. We also ensure that the work done by the pass is either amortized against other work done by the algorithm, or the pass decreases the number of nodes in the graph by a constant fraction.

Pass-3-4 applies PR3 and PR4 to as many nodes as possible while scanning each node at most twice. To maintain this invariant, we sometimes apply a weaker version of PR4 by taking the sum over a subset of neighbors of v. First all nodes are marked as unscanned. Then we pick an unscanned node v and apply tests 3 and 4 to its adjacent edges. In the process, we mark v and neighbors of v (including new neighbors created by contractions) as scanned. The pass terminates when all nodes are scanned.

3.2 Common Preprocessing On some instances, in particular the TSP instances, the problem size can be substantially reduced using PR tests on the input problem. On these problems, we use the following preprocessing algorithm. We start by finding the minimum single-vertex cut. Then we repeatedly apply pass-1-2 as long as a pass decreases the number of nodes by at least a factor of $1 - \gamma$. Second repeatedly apply pass-3-4 as long as a pass decreases the number of nodes by at least a factor of $1 - \gamma$. Finally, if either pass-1-2 was executed more than once or pass-3-4 was executed more than once or if pass-1-2 followed by pass-3-4 reduces the number of nodes by at least a factor of $1 - \beta$, we repeat the pass. The bail-out parameters β and γ are algorithm-specific, and we always choose $\beta \geq \gamma$. The value of β is higher for algorithms that use PR tests internally and lower for other algorithms. Note that the preprocessing runs in $O(m \log n)$ time.

4 Nagamochi-Ibaraki Algorithm

We assume that the reader is familiar with the Nagamochi-Ibaraki algorithms (see [22, 23]) and describe our implementation of it.

4.1 Data Structures We implemented ni using both compact contraction and set-union contraction. We use amortization techniques to combine the two data structures. Recall that the algorithm operates in phases. Each phase takes linear time in the current graph size, plus some priority queue operations.

We combine the two data structures as follows. At the beginning of every phase, we compact the graph by deleting self-loops, merging parallel arcs, and making every arc head point directly to the set representative. We use the compact representation during a phase. During the edge contraction sequence after the phase, we use the set-union data structure. This combination is efficient because the more efficient data structure is used in an appropriate part of the computation. Graph compaction is the additional overhead of the combination scheme, but the compaction takes linear time – usually less time than the preceding phase.

4.2 Incorporating the Padberg-Rinaldi Heuristic Nagamochi et al. [23] show that the Nagamochi-Ibaraki algorithm can benefit from the PR heuristics. At the end of every phase, they take a node created by the last contraction of the phase and apply PR tests to “large capacity” edges adjacent to this node until a large capacity edge is not contracted. This is the strategy used in their code `hybrid`. Although in some cases this strategy works well, it has several disadvantages. First, there is no preprocessing stage. Second, the tests are applied only to one node after every phase. Third, the tests may be expensive because we may end up doing $\Omega(n^2)$ work.

We incorporate PR heuristics into `ni` in a well-behaved manner. We use PR preprocessing on the input graph and, after every k-th phase, we apply pass-1-2 and pass-3-4. (We set $k = 2$ for our tests.) Preprocessing takes $O(m \log n)$ time. Since the cost of each PR pass is linear, it is amortized by the cost of the preceding phase.

5 Hao-Orlin Algorithm

Hao-Orlin algorithm [17] is based on the push-relabel method [13] for the maximum flow problem. The starting point for our implementation, described in this section, was a maximum flow code of [8]. We assume familiarity with these papers.

5.1 Implementation Details

Graph representation. We use graph data structures similar to those used in `ni`, with additional fields to support variables needs for preflow-push algorithms, such as flow values on edges and distance labels and excess at nodes. When using PR heuristics, we contract the source and the sink at the end of every s-t cut computation.

Global updates. As in the maximum flow context, *global updates*, *i.e.,* computing exact distances to the sink, are useful for many problem classes. We modify this heuristic to work in the context of `ho` and use it in our implementation.

Single node layers. The algorithm freezes layers of nodes. For the case when a layer consists of a single node, we have a more efficient method of dealing with it. See the full paper for details.

5.2 Heuristics

5.2.1 Padberg-Rinaldi Heuristics

As in all our algorithms, we apply PR preprocessing at the beginning. Our *internal* PR heuristic is based on the following fact that is easy to prove using [17]: If s is the source and the edge $\{s, w\}$ passes one of the tests `PR1`, ..., `PR4`, then we can saturate all arcs out of w, contract the edge, and continue.

We use amortization to decide when to apply `PR1` and `PR2` to the source. When the algorithm performs enough work to amortize the tests previous, we apply the tests again. We use similar strategy to apply `PR3` and `PR4`.

5.2.2 Excess contractions.

We introduce a simple new heuristic that often allows us to contract a node in the middle of a flow computation. The general results on the push-relabel method [13] imply that the excess at a node v is a lower bound on the capacity of the minimum s-v cut. Thus, if at some point during an s-t cut computation the excess at v becomes greater than or equal to the capacity of the minimum cut we have seen so far, we contract v into the source and saturate arcs going out of v. Note that v can be either regular or frozen. The correctness proof for this heuristic is straight-forward. We call this heuristic *excess detection.*

6 Karger-Stein Algorithm

The full description of the recursive contraction algorithm of Karger and Stein appears in [20]. The version we implemented differs slightly from the recursive contraction algorithm described in [20]. In that algorithm we repeatedly select and contract one edge at a time until the number of graph nodes is reduced to $n/\sqrt{2}$. In our implementation, at each contraction phase we mark each edge $\{u, v\}$ with probability $1-2^{-w(u,v)/W}$ and contract the marked edges. This new parameter W must be handled carefully. Initially, we have no upper bound on the minimum cut value. If W is extremely large relative to c then the probability of edge contraction is extremely small, which could make the recursion depth and running time arbitrarily large. However, we have a convenient upper bound on c in the form of the minimum degree of the input graph. It is easy to prove that if we use this upper bound, with high probability the size of the graph is reduced by a constant fraction at each iteration. This suffices to prove a time bound which is polynomial, though not quite as good as the original algorithm's. We conjecture that the new algorithm's performance is in fact equal to that of the old one's, but this remains to be proved.

Our new implementation requires an analysis somewhat different from that of [20]. See the full paper for details.

6.1 Implementation Details We use a simple data structure for storing the graphs: an array of edges, where each edge stores its endpoints and weight and an array of vertices, where each vertex maintains the sum of the weight of its incident edges, its name, and several auxiliary fields. Note that each vertex does not maintain a list of its incident edges, as this was not initially necessary in our implementation. An advantage of this representation is that edge lists are stored without a linked-list data structure overlay; this improves performance by making the data structure smaller and increasing memory reference locality as we traverse the edge list.

We found that it is much wiser to randomly choose and contract sets of edges, as described above, rather than choosing and contracting the edges one by one, as is done in [20]. We needed to use an exponential distribution to sample each edge with probability exponential in its weight; this was not a significant part of the running time.

It should be noted that our algorithm as stated makes no use of adjacency lists. However, we still found it useful to sort edges according to their endpoints. One reason is that contracting nodes tends to create parallel edges. By merging these parallel edges, we reduce the number of edges the algorithm must consider for contraction in later iterations. Furthermore, the `PR3` and `PR4` tests (discussed below) require a standard adjacency list representation to work properly. Sorting the edges gives us an implicit adjacency list representation.

As edges are contracted, we use a set union data structure to maintain the nodes of the graph. Besides letting us determine the structure of the minimum cut we find, this also lets us sort the edges.

6.2 PR tests As in other implementations, we do PR preprocessing. For this algorithm, these initial PR tests are particularly important, as they occur before beginning the randomized portion of the algorithm.

Next we discuss internal tests. The lack of an adjacency list representation in our code meant that the PR tests were completely different in terms of implementation. `PR1` and `PR2` tests apply to individual edges, and were therefore easy to implement on our original data structures. `PR3` and `PR4` tests apply to groups of edges leaving one node, and so we were forced to contort our original data structures to support them. Our overall experience with the latter tests was that they were an effective preprocessing tool but did not tend to help once the Contraction Algorithm actually began to run. Therefore, we imagine that a new implementation of the code might use a "PR34-friendly" adjacency list data structure for preprocessing and then convert to our original representation for the actual Contraction Algorithm.

See the full paper for details.

7 Karger's Algorithm

Karger's algorithm is described in [19]. Karger's first finds a packing of spanning trees; then it examines the trees in the packing and for each tree finds the smallest cut that one or two tree edges. This leaves open many implementation options. We discuss the major ones below.

Packing spanning trees. Karger suggests two entirely different ways to pack spanning trees. We chose to use Gabow's algorithm [12]. It would be interesting to try the alternative ([26]) too.

Checking trees for 2-respecting cuts. Karger gives two different ways to check for 2-respecting cuts: a relatively simple dynamic programming method that takes $\Theta(n^2)$ time per tree, and more elaborate method that uses dynamic trees, but takes only $O(m \log^2 n)$ per tree. We believe that the constant factor hidden in the O of the latter is large, so we only implemented the former.

Estimating the Minimum Cut In order to find the value of the minimum cut the algorithm randomly samples the edges with a probability that depends on the value of the minimum cut. To handle the apparent circularity of this process, we start by getting a crude approximation and then sample the edges, find a tree packing, and double the probability and repeat if the number of trees in the packing is smaller than expected. This implementation takes $O(m \log n)$ time.

Picking the ϵ. In order to have high probability of some tree 2-respecting, the expected number of trees has to be $150 \log n$. With $216 \log n$ trees, if none 1-respect, then at least $1/5$ of them 2-respect, and we could check fewer than $150 \log n$ trees for 2-respecting at the cost of having found more trees in the first place. Unfortunately, in either case, the running time was several orders of magnitude worse than the other algorithms. We discovered, however, that on our test examples, finding only $6 \log n$ trees and checking only 10 of them for 2-respecting gave the right answer on all the inputs. While there is reason to believe that the theoretical analysis is not tight, this implementation must be considered heuristic. Note that $6 \log n$ trees causes all cuts to be within $\epsilon = 1$ of their expected values with probability $1 - O(1/n)$ according to the theoretical analysis.

Adding the PR Heuristics We use the same PR preprocessing as `ho` and `ni`. Since the algorithm doesn't do any other contractions, the only way the PR heuristics could help internally is if we discover a new upper bound on the minimum cut before we get a good tree packing. We implicitly get a new upper bound when we are doubling probabilities, but it is not safe to run PR tests based on an expected upper bound. Nevertheless, as mentioned above, we check the cut defined by the minimum cut in the sample to update our upper bound if possible, so when an update occurs that reduces the lower bound by more than 5% we run the PR tests again.

8 Problem Families

After experimenting with many problem generators and families, we restricted our attention to the most interesting ones. In our final experiment, we used five different problem generators, and for each generator generated several different problem families, for a total of twelve different families. We also did experiments on minimum cut problems that arise in the solution of large TSP problems via cutting plane algorithms [3]. One of our generators, NOIGEN, is an implementation of the generator of [23]. The remaining generators were designed and implemented from scratch. One of the contributions of our work is to introduce this useful collection of generators than can be used by others to test minimum cut algorithms.

Due to the lack of space, we give data only for three problem families, and only the data that shows relative algorithm performance. The first family is NOI1, produced by NOIGEN generator and is similar to the first problem family of [23]. Two other families are generated by the REGGEN generator, which produces unions of random cycles. Each REG2 problem contains 50 cycles; each REG3 problem contains 2 cycles.

The full paper contains much more data, including data for other families and data that shows effectiveness of heuristics. Relative performance of the algorithms is very different for these three families.

9 Experimental Results

We give data for our four implementations `ni` (Nagamochi-Ibaraki algorithm), `ho` (Hao-Orlin algorithm) `ks` (Karger-Stein algorithm), and `k` (Karger's algorithm). For the last code, we use the version that picks $6 \log n$ trees, which is well below the theoretical bound. (See Section 7.) The theoretically justified version, however, was very slow, and `k` always produced correct answers in our tests. We also give data for `hybrid`, the Nagamochi *et al.* [23] implementation of the Nagamochi-Ibaraki algorithm with a PR heuristic.

The most robust code in our study is `ho`, but it does not dominate all other algorithms. The second best code is `ni`. See the full paper for discussion of performance of individual and effectiveness of heuristics. Also, the full paper gives the details of our experimental setup.

Table 1 gives data for NOI1 family. On this family, `ni` is the fastest code, with `ho` and `hybrid` slower by about a factor of two. The other two codes, `ks` and `k`, are much slower. The latter code is somewhat faster than the former.

Table 2 gives data for REG2 family. On REG2 problems, `ho` is clearly the fastest, with other algorithms asymptotically slower.

Table 3 gives data for REG3 family. Recall that REG3 problems are unions of two random cycles. On these problems, `k` works like Gabow's algorithm and its running time is almost linear. For `ks` the constant factors are large, but the asymptotic performance is the second best. Even on the largest problem, however, `ks` is significantly slower than `ho`. Other codes, `ho`, `ni`, and `hybrid`, exhibit growth that is at least quadratic in the problems size. However, `ho` has very small constants and is the fastest on all problems except for the largest, where it is slower than `k`. The worst codes are `ni` and `hybrid`.

10 Concluding Remarks

Our study produced several efficient codes which improve the previous state of the art. Some of the algorithms in our study have not been implemented before. We also introduce new or improved heuristics that substantially improve performance.

Our results confirm practical importance of the Padberg-Rinaldi heuristics. We introduce PR passes and show that they are an effective way to apply PR tests in the preprocessing stage of all the algorithms. The passes are also an important component of our internal PR heuristic for `ni`.

Improved data structures and PR heuristic lead to an efficient implementation of the Nagamochi-Ibaraki algorithm. This implementation, `ni`, is more robust than the previous implementation `hybrid`. The implementation `ho` of the Hao-Orlin algorithm is the most robust in our tests. This is due in part to previous work on implementations of push-relabel algorithms for the maximum flow problem. Performance of `k` shows that this algorithm needs to be investigated further. Our implementation of the Karger-Stein algorithm does not perform as well as the best codes, but its performance is not bad when compared to the previous codes. This implementation may be useful in some contexts, for example if one would like to find all minimum cuts.

`ni` outperforms `ho` if the number of `ni` phases is very small, *i.e.*, when a phase contracts many nodes. This suggests a hybrid algorithm that, after the preprocessing, applies `ni` phases (with internal PR tests) while the number of nodes decreases by a constant factor after each phase, and then switches to `ho`. We tested this hybrid algorithm, and it proved to be more robust than `ni` and `ho`. In fact, this is the most robust implementation we currently have. This and other hybrid algorithms deserve further study.

References

[1] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved Time Bounds for the Maximum Flow Problem. *SIAM J. Comput.*, 18:939–954, 1989.

[2] R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.

[3] D. L. Applegate and W. J. Cook. Personal communication. 1996.

[4] R. A. Botafogo. Cluster Analysis for Hypertext Systems. In *Proc. of the 16-th Annual ACM SIGIR Conference of Res. and Dev. in Info. Retrieval*, pages 116–125, 1993.

[5] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array Distribution in Data-Parallel Programs. In *Languages and Compilers for Parallel Computing*, pages 76–91. Lecture Notes in Computer Science series, vol. 896, Springer-Verlag, 1996.

[6] J. Cheriyan and T. Hagerup. A randomized maximum flow algorithm. In *Proc. FOCS 30*, pages 118–123, 1989.

[7] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a Maximum Flow be Computed in $o(nm)$ Time? In *Proc. ICALP*, 1990.

[8] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. Technical Report STAN-CS-94-1523, Department of Computer Science, Stanford University, 1994.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[10] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

[11] U. Derigs and W. Meier. An Evaluation of Algorithmic Refinements and Proper Data-Structures for the Preflow-Push Approach for Maximum Flow. In *ASI Series on Computer and System Sciences*, volume 8, pages 209–223. NATO, 1992.

[12] H. N. Gabow. A Matroid Approach to Finding Edge Connectivity and Packing Arborescences. *J. Comp. and Syst. Sci.*, 50:259–273, 1995.

[13] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	300	22425	0.31	0.07	—	—	—	—	—	—	3	0.51	—	—
ho	300	22425	0.32	0.05	0.04	0.33	0.05	0.25	0	0.00	165	0.11	384908	0.08
ni	300	22425	0.18	0.02	0.03	0.57	0.04	0.26	0	0.00	240	0.04	202990	0.03
k	300	22425	3.03	0.04	0.03	0.38	0.05	0.17	0	0.00	0	0.00	1049549	0.07
ks	300	22425	5.04	0.08	0.02	0.50	0.23	0.07	0	0.00	3375	0.02	1350153	0.07
hybrid	400	39900	0.60	0.02	—	—	—	—	—	—	3	0.76	—	—
ho	400	39900	0.65	0.03	0.07	0.16	0.10	0.08	0	0.00	250	0.03	676746	0.01
ni	400	39900	0.39	0.03	0.05	0.42	0.10	0.08	0	0.00	326	0.04	362615	0.04
k	400	39900	5.78	0.03	0.06	0.29	0.09	0.11	0	0.00	0	0.00	1887817	0.07
ks	400	39900	10.82	0.06	0.02	0.20	0.51	0.03	0	0.00	4448	0.02	2514772	0.04
hybrid	500	62375	1.02	0.05	—	—	—	—	—	—	4	0.36	—	—
ho	500	62375	1.05	0.01	0.05	0.62	0.17	0.05	0	0.00	325	0.06	1028294	0.01
ni	500	62375	0.66	0.04	0.06	0.65	0.15	0.05	0	0.00	407	0.05	553674	0.02
k	500	62375	9.66	0.03	0.06	0.64	0.14	0.08	0	0.00	0	0.00	3017636	0.09
ks	500	62375	18.86	0.11	0.05	0.11	0.95	0.03	0	0.00	5769	0.03	4118577	0.08
hybrid	600	89850	1.61	0.05	—	—	—	—	—	—	5	0.28	—	—
ho	600	89850	1.60	0.01	0.15	0.45	0.25	0.05	0	0.00	406	0.03	1525886	0.02
ni	600	89850	0.97	0.01	0.14	0.42	0.23	0.02	0	0.00	507	0.03	802661	0.02
k	600	89850	14.51	0.10	0.14	0.46	0.21	0.04	0	0.00	0	0.00	4561222	0.23
ks	600	89850	33.15	0.04	0.09	0.11	1.50	0.02	0	0.00	7139	0.03	6529056	0.02
hybrid	700	122325	2.54	0.06	—	—	—	—	—	—	5	0.25	—	—
ho	700	122325	2.27	0.01	0.23	0.30	0.35	0.02	0	0.00	506	0.02	2101482	0.01
ni	700	122325	1.41	0.02	0.24	0.28	0.31	0.03	0	0.00	620	0.02	1132793	0.03
k	700	122325	19.78	0.02	0.23	0.24	0.29	0.03	0	0.00	0	0.00	5726697	0.07
ks	700	122325	51.63	0.03	0.13	0.04	2.13	0.01	0	0.00	8768	0.03	9368782	0.03
hybrid	800	159800	3.26	0.07	—	—	—	—	—	—	7	0.31	—	—
ho	800	159800	3.02	0.01	0.22	0.73	0.46	0.02	0	0.00	570	0.05	2729953	0.01
ni	800	159800	1.83	0.04	0.21	0.72	0.42	0.01	0	0.00	686	0.04	1448069	0.03
k	800	159800	26.47	0.03	0.21	0.72	0.39	0.02	0	0.00	0	0.00	8006861	0.09
ks	800	159800	67.22	0.06	0.17	0.05	2.81	0.01	0	0.00	9752	0.04	11886606	0.04
hybrid	900	202275	4.54	0.05	—	—	—	—	—	—	7	0.09	—	—
ho	900	202275	3.91	0.01	0.28	0.71	0.58	0.01	0	0.00	664	0.02	3483609	0.01
ni	900	202275	2.36	0.02	0.28	0.65	0.53	0.01	0	0.00	802	0.02	1847438	0.01
k	900	202275	34.42	0.11	0.28	0.66	0.50	0.02	0	0.00	0	0.00	10396242	0.25
ks	900	202275	92.15	0.02	0.21	0.05	3.63	0.01	0	0.00	11290	0.03	15490878	0.02
hybrid	1000	249750	5.94	0.03	—	—	—	—	—	—	5	0.37	—	—
ho	1000	249750	4.93	0.01	0.30	0.60	0.73	0.01	0	0.00	759	0.01	4284685	0.01
ni	1000	249750	3.02	0.01	0.29	0.61	0.66	0.01	0	0.00	902	0.01	2355983	0.01
k	1000	249750	42.30	0.02	0.30	0.57	0.62	0.01	0	0.00	0	0.00	12173350	0.06
ks	1000	249750	122.12	0.01	0.26	0.04	4.55	0.00	0	0.00	13301	0.01	20070021	0.01

Table 1: NOI1 family

[14] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *J. SIAM*, 9:551–570, 1961.

[15] J. Hao. A Faster Algorithm for Finding the Minimum Cut of a Graph. Unpublished manuscript, 1991.

[16] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut of a Graph. In *Proc. SODA 3*, pages 165–174, 1992.

[17] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut in a Directed Graph. *J. Algorithms*, 17:424–446, 1994.

[18] D. R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. In *Proc. STOC 27*, pages 11–17, 1995.

[19] D. R. Karger. Minimum Cuts in Near-Linear Time. In *Proc. STOC 28*, pages 56–63, 1996.

[20] D. R. Karger and C. Stein. An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts. In *Proc. STOC 25*, pages 757–765, 1993.

[21] V. King, S. Rao, and R. Tarjan. A Faster Deterministic Maximum Flow Algorithm. *J. Algorithms*, 17:447–474, 1994.

[22] H. Nagamochi and T. Ibaraki. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. Disc. Meth.*, 5:54–66, 1992.

[23] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an Efficient Minimum Capacity Cut Algorithm. *Math. Prog.*, 67:297–324, 1994.

[24] Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.

[25] M. Padberg and G. Rinaldi. An Efficient Algorithm for the Minimum Capacity Cut Problem. *Math. Prog.*, 47:19–36, 1990.

[26] S. A. Plotkin, D. Shmoys, and É. Tardos. Fast Approximation Algorithms for Fractional Packing and Covering. In *Proc. FOCS 32*, 1991.

[27] A. Ramanathan and C. Colbourn. Counting Almost Minimum Cutsets with Reliability Applications. *Math. Prog.*, 39:253–261, 1987.

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	50	2500	0.04	0.19	—	—	—	—	—	—	10	0.07	—	—
ho	50	2500	0.02	0.00	0.00	0.00	0.00	2.00	0	0.00	42	0.02	30955	0.05
ni	50	2500	0.02	0.00	0.00	0.00	0.00	2.00	0	0.00	46	0.00	20664	0.01
k	50	2500	0.13	1.05	0.00	0.00	0.00	2.29	0	0.00	0	0.00	38170	0.29
ks	50	2500	0.28	0.05	0.00	2.29	0.02	0.37	0	0.00	571	0.04	81052	0.05
hybrid	100	5000	0.25	0.03	—	—	—	—	—	—	14	0.07	—	—
ho	100	5000	0.05	0.36	0.00	0.00	0.01	1.22	0	0.00	72	0.21	112675	0.21
ni	100	5000	0.05	0.17	0.00	0.00	0.00	2.00	0	0.00	95	0.00	89037	0.03
k	100	5000	0.52	0.81	0.00	0.00	0.00	2.71	0	0.00	0	0.00	106738	0.44
ks	100	5000	0.96	0.04	0.00	2.29	0.03	0.17	0	0.00	1350	0.03	285035	0.04
hybrid	200	10000	1.34	0.05	—	—	—	—	—	—	32	0.03	—	—
ho	200	10000	0.17	0.11	0.00	0.00	0.02	0.50	0	0.00	154	0.19	354763	0.10
ni	200	10000	0.17	0.15	0.00	0.00	0.00	2.00	0	0.00	191	0.01	376816	0.17
k	200	10000	1.23	0.35	0.00	0.00	0.01	0.62	0	0.00	0	0.00	353150	0.18
ks	200	10000	3.02	0.03	0.01	1.33	0.08	0.06	0	0.00	2995	0.02	844806	0.03
hybrid	400	20000	6.64	0.03	—	—	—	—	—	—	66	0.01	—	—
ho	400	20000	0.56	0.11	0.00	0.00	0.05	0.15	0	0.00	291	0.17	872105	0.13
ni	400	20000	9.06	0.02	0.00	0.00	0.04	0.26	0	0.00	277	0.01	13649236	0.02
k	400	20000	4.93	0.16	0.00	3.39	0.04	0.23	0	0.00	0	0.00	865319	0.18
ks	400	20000	8.75	0.02	0.01	1.04	0.21	0.06	0	0.00	6277	0.02	2183688	0.02
hybrid	800	40000	33.25	0.03	—	—	—	—	—	—	139	0.02	—	—
ho	800	40000	2.00	0.17	0.00	0.00	0.12	0.03	0	0.00	646	0.08	2556624	0.18
ni	800	40000	48.19	0.01	0.00	0.00	0.10	0.00	0	0.00	557	0.00	56706079	0.01
k	800	40000	26.04	0.07	0.00	4.90	0.11	0.09	0	0.00	0	0.00	231376	0.00
ks	800	40000	26.15	0.01	0.02	0.19	0.60	0.04	0	0.00	13599	0.01	5443391	0.01

Table 2: REG2 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	256	512	0.13	0.11	—	—	—	—	—	—	1	0.73	—	—
ho	256	512	0.02	0.50	0.00	0.00	0.00	0.00	4	0.44	139	0.08	20739	0.16
ni	256	512	0.10	0.13	0.00	0.00	0.00	0.00	4	0.44	125	0.22	142274	0.07
k	256	512	0.04	0.24	0.00	0.00	0.00	0.00	4	0.44	0	0.00	3407	0.00
ks	256	512	0.45	0.06	0.00	0.00	0.00	0.00	7	0.63	4501	0.03	133258	0.05
hybrid	512	1024	0.45	0.04	—	—	—	—	—	—	2	0.42	—	—
ho	512	1024	0.04	0.26	0.00	0.00	0.00	0.00	4	0.27	318	0.13	51902	0.11
ni	512	1024	0.39	0.07	0.00	2.00	0.00	0.00	4	0.27	231	0.12	529756	0.08
k	512	1024	0.08	0.16	0.00	0.00	0.00	2.71	4	0.27	0	0.00	6831	0.00
ks	512	1024	1.12	0.03	0.00	0.00	0.01	0.62	6	0.36	9894	0.02	318691	0.02
hybrid	1024	2048	1.80	0.03	—	—	—	—	—	—	4	0.42	—	—
ho	1024	2048	0.09	0.23	0.00	0.00	0.01	0.82	6	0.32	651	0.03	124559	0.11
ni	1024	2048	1.46	0.05	0.00	0.00	0.00	0.00	6	0.32	358	0.22	1954614	0.05
k	1024	2048	0.22	0.13	0.00	0.00	0.00	4.90	6	0.32	0	0.00	13666	0.00
ks	1024	2048	2.68	0.02	0.00	4.90	0.03	0.15	9	0.23	21465	0.01	729360	0.01
hybrid	2048	4096	8.62	0.04	—	—	—	—	—	—	2	0.50	—	—
ho	2048	4096	0.31	0.29	0.00	0.00	0.01	0.82	6	0.30	1291	0.02	401943	0.30
ni	2048	4096	5.95	0.03	0.00	0.00	0.01	1.22	6	0.30	626	0.35	7326869	0.03
k	2048	4096	0.57	0.07	0.00	0.00	0.01	0.62	6	0.30	0	0.00	27313	0.00
ks	2048	4096	5.91	0.02	0.00	1.78	0.06	0.16	8	0.41	45017	0.01	1536256	0.01
hybrid	4096	8192	39.82	0.02	—	—	—	—	—	—	2	0.42	—	—
ho	4096	8192	1.04	0.16	0.00	0.00	0.03	0.14	5	0.36	2674	0.04	1177472	0.15
ni	4096	8192	26.06	0.02	0.00	0.00	0.03	0.00	5	0.36	972	0.32	28309175	0.03
k	4096	8192	1.62	0.05	0.00	0.00	0.03	0.00	5	0.36	0	0.00	54598	0.00
ks	4096	8192	16.18	0.01	0.00	1.78	0.13	0.07	7	0.31	97944	0.00	3326858	0.00
hybrid	8192	16384	177.96	0.01	—	—	—	—	—	—	1	0.31	—	—
ho	8192	16384	4.14	0.39	0.00	0.00	0.10	0.08	5	0.22	5400	0.02	3589940	0.40
ni	8192	16384	167.89	0.03	0.00	2.00	0.08	0.00	5	0.22	2275	0.25	105627355	0.02
k	8192	16384	4.03	0.04	0.00	4.90	0.08	0.10	5	0.22	0	0.00	109226	0.00
ks	8192	16384	43.27	0.01	0.02	0.20	0.42	0.02	8	0.33	210805	0.01	7149966	0.00
hybrid	16384	32768	843.45	0.00	—	—	—	—	—	—	1	1.10	—	—
ho	16384	32768	18.53	0.37	0.00	0.00	0.24	0.04	5	0.61	10651	0.07	13903247	0.38
ni	16384	32768	780.60	0.01	0.00	2.00	0.21	0.05	5	0.61	4665	0.28	392336486	0.01
k	16384	32768	9.51	0.02	0.00	4.90	0.20	0.03	5	0.61	0	0.00	218432	0.00
ks	16384	32768	111.66	0.00	0.07	0.07	1.20	0.02	5	0.71	434485	0.00	14772484	0.00

Table 3: REG3 family