Cut Tree Algorithms

Andrew V. Goldberg*

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
avg@research.nj.nec.com

Kostas Tsioutsiouliklis

Computer Science Department
Princeton University
Princeton, NJ 08540
kt@cs.princeton.edu

Abstract

This is an experimental study of algorithms for the cut tree problem. We study the Gomory-Hu and Gusfield's algorithms as well as heuristics aimed to make the former algorithm faster. We develop an efficient implementation of the Gomory-Hu algorithm. We also develop problem families for testing cut tree algorithms. In our tests, the Gomory-Hu algorithm with a right combination of heuristics was significantly more robust than Gusfield's algorithm.

1 Introduction

Cut trees, introduced by Gomory and Hu [10] and also known as *Gomory-Hu trees*, represent the structure of all s-t cuts of undirected graphs in a compact way. The cut trees have many applications.

All known algorithms for building cut trees use a minimum s-t cut subroutine. The most efficient currently known way to find a minimum s-t cut is using a maximum flow algorithm. See [8] for the currently known maximum flow bounds. Gomory and Hu [10] showed how to solve the tree problem using $n-1$[1] minimum cut computations and graph contractions. Gusfield [12] proposed an algorithm that does not use graph contraction; all $n-1$ minimum s-t cut computations are performed on the input graph. Gusfield's algorithm is very simple and can be implemented by adding a few lines to a maximum flow code.

Computational performance of algorithms for closely related problems, the maximum flow problem and the (global, e.g. over all s,t pairs) minimum cut problem has been studied extensively; see e.g. [1, 4, 5, 7, 18] for computational studies of the former problem and [3, 15, 16, 17, 19] for the latter. Both problems can be solved well in practice: most problems that fit in RAM of a modern computer can be solved in a few minutes. The cut tree problem appears more difficult, for one needs to solve $n-1$ minimum s-t cut problems.

Therefore, computational performance of cut tree algorithms is of great interest. Implementations of cut tree algorithms exist – for example, as subroutines of TSP codes [2, 11]. However, we are not aware of any published computational studies of cut tree algorithms. In this paper we undertake such a study.

We describe how to implement Gomory-Hu and Gusfield's algorithms efficiently (which is nontrivial for the former). We also introduce and study heuristics aimed at improved computational performance of these algorithms. Our computational experiments lead to a good understanding of practical performance of the cut tree algorithms.

2 Definitions and Notation

The input to the cut tree problem is an undirected graph $G = (V, E)$ and a capacity function $c : E \Rightarrow \mathbf{R}^+$. We denote $|V| = n$ and $|E| = m$. A *cut* (X, Y) in G is a partitioning of V into two nonempty sets. We say that an edge *crosses* the cut if its two endpoints are on different sides of the cut. *Capacity of a cut* is the sum of capacities of edges crossing the cut.

We distinguish between *vertices* and *nodes*. We refer to the elements of V as vertices. Nodes correspond to subsets of vertices. (A node can be a single-element subset.) We need the distinction because we use contraction operations.

For $s, t \in V$, an s-t cut is a cut such that s and t are on different sides of it. A *minimum s-t cut* is an s-t cut of minimum capacity. A *(global) minimum cut* is a minimum s-t cut over all s, t pairs.

A *cut tree* is a weighted tree T on V with the following property. For every pair of distinct vertices s and t, let e be a minimum weight edge on the unique path from s to t in T. Deleting e from T separates T

*Current address: InterTrust STAR Laboratory, 460 Oakmead Parkway, Sunnyvale, CA 94086.

[1] We denote the number of vertices and edges in the input graph by n and m, respectively.

into two connected components, X and Y. Then (X, Y) is a minimum s-t cut. Note that T is not a subgraph of G, i.e. edges of T do not need to be in E.

3 Gomory-Hu Algorithm

In this section we outline the Gomory-Hu algorithm and its efficient implementation. We also discuss heuristics that may improve algorithm's performance in practice. We provide only the details of the algorithm needed to describe the implementation and the heuristics. For a complete description, see e.g. [6, 10, 14].

The Gomory-Hu algorithm is recursive. It distinguishes between two kinds of nodes: original and contracted. A vertex of the input graph is an *original node*. If there are more than one original nodes, the algorithm picks two, s and t, finds a minimum s-t cut (S, T), and forms two graphs, G_s by contracting S into a contracted node, and G_t by contracting T. Then it recursively builds cut trees in G_s and G_t and puts these trees together. One can see that the algorithm maintains the following invariant: a trivial cut around a contracted node is a minimum cut between this node and some other node in the graph. If there is only one original node, the algorithm has enough information to construct a cut tree; in this case the recursion bottoms out.

Because of the contraction operations, efficient implementation of the Gomory-Hu algorithm is nontrivial. (This was the main motivation behind Gusfield's algorithm.) A naive implementation of contractions allocates new memory for a contracted node and its edges. This may result in $\Omega(n^2)$ memory allocation even for a sparse graph. We describe an implementation that uses $O(\log n)$ extra node records and $O(n)$ extra edge records. These records are allocated as a block at the beginning of the computation, avoiding expensive allocation of small pieces of memory throughout the computation and improving locality of reference.

We maintain the following information at every step of the computation. Recall that the computation is recursive. For each recursive call currently in progress, we maintain information about the cut computed at this level and the graphs obtained by contracting one of the cut sides. When the first recursive call returns, we mark node and edge records of the corresponding subgraph as free. We also maintain information about the contracted nodes (on which side of the cut they are each time), since this determines the structure of the final cut tree. Finally, we maintain a data structure that builds the cut tree according to the cuts found so far.

Our implementation first recurses on the subgraph with a smaller number of nodes and uses fewer additional nodes and edges because of this. We analyze these numbers next.

For the second recursive call, we can reuse the nodes and the edges of the already processed subgraph and use no additional storage. The number of extra nodes we need is determined by the longest sequence of left branches in a root-to-leaf path in the recursion tree (corresponding to the first recursive calls), which is $\lceil \log_2 n \rceil$ because we recurse on the smaller subgraph first. Similarly, the number of extra edges needed is determined by the maximum, over all root-to-leaf paths, of the sum of sizes of subproblems corresponding to left branches. The maximum is bounded by n.

Note that our implementation destroys the input graph. If this is not desirable, one can make a copy of the graph before running the algorithm. All our codes are superlinear, and the time to make the copy would be negligible except for small graphs.

When implemented as described above, direct overhead of contraction operations is small; contraction usually costs less than the corresponding minimum cut computation. However, there is also indirect cost: locality of the input graph representation suffers because of the contractions, reducing the number of cache hits somewhat.

At high level, two major factors determine the computational performance of the algorithm. The first one is the balance (e.g., the ratio of the number of nodes) of the cuts found by the algorithm. In the worst case, one side of every such cut contains one node. In the best case, the cuts are balanced. In the latter case, assuming that minimum cut computations are superlinear, the first one dominates the total running time. The second factor is the hardness of the minimum cut subproblems. Heuristics that lead to more balanced cuts or simpler subproblems improve the algorithm performance.

The *balance* heuristic aims at keeping the cuts balanced. Assume we have at least four original nodes. First we compute all minimum cuts between two such nodes, a and b, and take the most balanced cut. If the cut is sufficiently balanced (e.g. the ratio of the number of nodes of the larger and the smaller parts does not exceed a threshold), we proceed. Otherwise, we pick two nodes, c and d, on the bigger side of the cut. We compute all minimum cuts between c and d, take the most balanced one, compare it to the most balanced minimum cut between a and b, and choose the best. We may have to compute twice as many cuts, so the worst-case loss is about a factor of two. The best-case gain is much larger.

We can also use both cuts, since according to Gomory-Hu [10], if the cuts are crossing, we can always find noncrossing cuts. We use this technique, although

it usually does not lead to a big speedup: most often one of the cuts is quite unbalanced, and the computation to find noncrossing cuts is relatively expensive.

The *mincut heuristic* makes use of the Hao-Orlin algorithm [13] for finding global mincuts. This algorithm uses the push-relabel method to find a minimum cut between the source and the sink. Then it contracts the source and the sink, and selects a new sink. Hao and Orlin show that with a careful implementation of many push-relabel algorithms, the asymptotic worst-case time bound for these $n-1$ minimum s-t cut computations is the same as that for one minimum s-t cut computation of the underlying algorithm.

Note that the first cut found by the Hao-Orlin algorithm is a minimum s-t cut in the input graph. Also, the algorithm finds a minimum cut, which is a minimum s-t cut for any s, t on the opposite sides of it. We prove a lemma that allows to use several cuts found by the algorithm in the cut tree construction.

The Hao-Orlin algorithm has the following property. Let s be the initial source and let S be the set of vertices contracted into the source at some point of an execution of the algorithm. Let λ be the capacity of the smallest cut found up to this point (initially $\lambda = \infty$). Then for any $x \in S$, the capacity of a minimum s-x cut is at least λ.

LEMMA 3.1. *Suppose that t is the next sink and the minimum S-t cut has value $\lambda' \leq \lambda$. Then this cut is also a minimum cut between s and t in G.*

Proof. Suppose that there is a smaller cut between s and t. This cut cannot separate s from a vertex $x \in S$ because s and x are λ-connected. Thus the cut separates S and t. This contradicts the definition of λ'.

The mincut heuristic uses the above lemma and finds several minimum s-t cuts with one Hao-Orlin computation. This number is usually small, so we use this heuristic together with the balance heuristic to obtain one or two cuts — the most balanced ones.

The *source selection heuristic* is aimed at both making minimum cut computations simpler and making balanced cuts more likely. This heuristic uses the fact that any original node can be chosen as the source for the next minimum cut computation. After choosing a sink for the computation, we choose an original node that is furthest away from the sink as the source. (All distances are with respect to a unit length function.) Note that we use an implementation of the push-relabel method [9] based on that of [4]. This implementation computes distances to the sink during the initialization, so the source selection heuristic adds essentially no overhead.

As part of the source selection heuristic, we choose the source/sink to be the heaviest nodes of the graph (e.g. nodes with the highest total capacity of adjacent edges), since this sometimes leads to more balanced cuts.

3.1 Our Implementations After studying different ways of incorporating heuristics into the Gomory-Hu algorithm, we report on two codes. The GH code uses no heuristics and picks the next source/sink pair at random. The GHs code uses the source selection heuristic. The GHG code uses the mincut heuristic in the following way: Initially, it picks the two heaviest nodes as the source and the sink of the Hao-Orlin algorithm.[2] As soon as it finds a cut in the decreasing sequence which is more balanced than the first cut found, it splits the graph according to both this cut and the first one. Our experience shows that using the mincut heuristic is the best way to find balanced cuts at low expense.

4 Gusfield's Algorithm

Like the Gomory-Hu algorithm, Gusfield's algorithm [12] consists of $n-1$ iterations of a minimum cut subroutine and bookkeeping that puts the resulting cuts together. Gusfield's algorithm, however, does not contract vertices and works with the original graph, making it easy to implement. At each of the $n-1$ iterations of Gusfield's algorithm, a different vertex is chosen as the source. This choice determines the sink.

Low-level operations of this algorithm are efficient because of its simplicity and the fact that the algorithm takes advantage of locality of the input graph representation. However, all minimum cut subproblems are as big as the original graph. Furthermore, the algorithm has less flexibility for adding heuristics. The only flexibility is the choice of the next source. We choose the next source at random. We refer to the resulting implementation as GUS.

5 Experimental Setup

For our computations, we used a SUN Sparc Ultra-2 workstation with 256MB memory running SunOS 5.5.1. All the code is written in C and compiled with 'gcc' and optimization option -O4. Our implementations are written in the same style and are derived from the Hao-Orlin algorithm implementation of [3]. We attempted to make all implementations as efficient as possible.

For our tests we use problem families from the previous minimum cut studies [3, 16, 17, 19], but instead of finding a minimum cut of a graph, we build a cut

[2] A random choice is much less robust.

tree. We omit the description of the problem families. Detailed descriptions appear in [16]. We do not report on PR2–PR4 problem families because the results are very close to those for the PR1 family, and on REG3–REG4 families because the results are very close to those for the REG1 and REG2 families. We also use two new problem families produced by two generators, PATHGEN and TREEGEN, described below. A summary of the problem families we use appears in Table 2.

The PATHGEN generator works as follows. Given a parameter k, it builds a path of $k-1$ "heavy" edges and connects the remaining $n-k$ vertices to the path vertices by heavy edges, at random. Then it adds "light" edges at random to achieve the desired number of arcs and to make the minimum cut problems more difficult. This generator takes the following parameters:

- n, the number of vertices;
- d, the density of the graph as a percentage;
- k, the path length;
- P, the path arc capacity parameter;
- S, the seed.

Heavy edge capacities are chosen uniformly at random from the interval $[1, \ldots, 100 \cdot P]$ and light edge capacities from $[1, \ldots, 100]$.

The value of k determines the path shape. For example, if $k = n$ then we get one heavy path through all the nodes; if $k = 1$, then the graph is a star. We use PATHGEN to produce the PATH problem family. We use $n = 2,000$, $d = 10$, $P = 1,000$, and k changing from 1 to $2,000$.

The TREEGEN generator works as follows. Given a parameter k, it builds a tree by connecting vertex i, $2 \leq i \leq n$, to a randomly chosen vertex in $[1, \min(i-1, k)]$. The tree edges are heavy. Then it adds "light" edges at random to achieve the desired number of arcs and to make the minimum cut problems more difficult. This generator takes the following parameters:

- n, the number of vertices;
- d, the density of the graph as a percentage;
- k, the shape parameter mentioned above;
- P, the path arc capacity parameter;
- S, the seed.

The generator chooses heavy edge capacities uniformly at random from the interval $[1, \ldots, 100 \cdot P]$ and light edge capacities from $[1, \ldots, 100]$.

	GUS	GH	GHS	GHG
BIKEWHE	◯	◯	◯	◯+
CYC1	◯+	◯	◯	◯
DBLCYC	⊗	◯	◯	◯+
IRREG1	◯+	◯	◯	◯+
NOI1	◯+	◯	◯	◯
NOI2	⊙	◯+	◯	◯
NOI3	◯+	◯	◯	◯
NOI4	⊙	◯	◯	◯
NOI5	•	⊙	◯+	◯
NOI6	◯	◯	◯	◯
PATH	⊗	•	◯	◯
PR1	◯+	◯	◯	◯
PR5	◯	◯	◯+	◯
PR6	⊙	◯	◯	◯
PR7	⊙	◯+	◯	◯
PR8	⊙	◯+	◯	◯
REG1	◯+	◯	◯	◯
REG2	◯	◯	◯	◯
TREE	⊗	•	◯	⊙
TSP	⊗	◯	◯	◯
WHE	◯	◯	◯	◯+

Table 1: Summary of algorithm performance. ◯ means good, ⊙ means fair, ⊗ means poor, and • means bad. + marks the fastest code(s).

The value of k determines the shape of the tree. For example, if $k = 1$ then the tree is a star. If $k = n - 1$, then a tree is obtained by connecting each vertex except the first one to a randomly chosen preceding vertex.

We use TREEGEN to produce the TREE problem family.

6 Experimental Results

In this section we describe our experimental results. Table 1 summarizes these results. At the end of this paper we append data only for the TSP, PATH, and TREE problem families; the technical report version contains more data. We chose these problem families because we think the data for them is more interesting, and not because it is representative.

We use the following scoring system in the table. We normalize the times by that of the fastest code and use a factor of two as the threshold between adjacent scores. For example, if the fastest code runs in x seconds, a code running in $1.5x$ seconds is rated good, in $3x$ seconds – fair, in $7x$ seconds – poor, in $12x$ – bad. Our choice of the threshold makes it less likely that a code not rated good in our experiment would be the fastest under a different compiler and machine architecture combination. The scoring is done

using instances with the biggest performance difference (usually, the largest instances) for a problem family. If a code is consistently faster than the other codes, we mark that code with a +. Several codes can be marked so if their performance is very close, and no codes can be marked if there is no consistent winner. Note that no code will get a good score on a problem family if every code performs relatively poorly for some parameter values.

This scoring system gives a general idea of relative performance of the codes and is robust with respect to many low-level implementation details and machine architecture variations. Note that in some cases larger problem sizes may amplify performance differences and thus change the scores.

Data tables of the technical report (like the ones in the Appendix), give much more information than the above scores and can be used to explain performance differences. All our implementations are based on the push-relabel maximum flow method; we give counts of the push and relabel operations which give a machine-independent measure of performance. We also give the average size (the number of nodes and the number of edges) of the s-t cut problems solved. The average problem size is correlated with the algorithm performance. In addition to the total running time, we give the time spend computing minimum s-t cuts (CutTime) and the time spend on auxilary operations (ManipTime) such as building the cut tree, contracting nodes, etc. The total time is equal to the sum of the CutTime, ManipTime, initialization time, and postprocessing (e.g. output) time.

The data shows that GUS is not robust. Although it is the fastest code on many problem families, in some cases it performs much worse than the Gomory-Hu algorithms.

Operation counts show that Gusfield's algorithm wins mostly due to its simplicity and better spatial locality resulting from the lack of contraction operations. The algorithm works on the original graph, so the average subproblem size is the original graph size. In contrast, Gomory-Hu algorithm wins when it gets balanced cuts which reduce the average size of the subproblems and reduces the number of push and relabel operations (which dominate the computation).

Note that if one assumes that contraction operations do not increase the number of push and relabel operations needed to solve a minimum s-t cut problem, the only reason Gusfield's algorithm may be faster than the Gomory-Hu algorithm is because of better locality and the lack of contraction operations. Since the work of the latter can be amortized, Gusfield's algorithm cannot win by more than a moderate constant factor. The Gomory-Hu algorithm can, and in some cases does, win by a wide margin.

GHS is the most robust code in our study. This code received good marks on all problem families. It is close to the fastest code on all problem families, although it is seldom the fastest.

Receiving only one fair mark, GHG is almost as robust as GHS. On some input classes (BIKEWHE, DBLCYC, WHE), it outperforms the other codes. This is due to the fact that on these problem classes, GHG finds more balanced cuts and on the average works with smaller problems. In fact, the average problem size for GHG tends to be somewhat smaller than for GHS. However, the size is never much smaller, and often does not pay for the additional overhead. The TREE family is the only problem family where GHG gets a fair score. On this family the average problem size for GHG is smaller than for GHS, but the problems are easy for GHS, and GHG, finding several cuts for each problem, spent significantly more time on each of those problems.

The GH code performs similarly to GHS on most problem families except for a small number of families (in particular PATH and TREE), where the former code is noticeably slower. This seems to suggest that the source-sink heuristic is more robust than random selection.

7 Concluding Remarks

In this section we summarize our work and discuss heuristics that work as well as those that do not work.

Currently, the cut tree problems are substantially harder than the related maximum flow and minimum cut problems, both in theory and in practice. This is a good motivation for improving theoretical bounds for the problem and developing faster codes for it. Our study is a step towards the faster codes.

We get a good understanding of implementation issues for the existing cut tree algorithms, as well as a good understanding of computational performance of these algorithms. The latter suggests heuristics for improved performance. We provide experimental evidence that the Gomory-Hu algorithm is more robust than Gusfield's algorithm. This is because all subproblems solved by Gusfield's algorithm have the same size as the input problem. For the Gomory-Hu algorithm, however, the average problem size can be much smaller than the original problem size. The Gomory-Hu algorithm performance is less predictable, because the average problem size depends on the heuristics used. Good heuristics reduce the size and can substantially improve performance on some problems.

The source selection strategy of selecting the heaviest node as the source is the most robust in our tests.

Random selection does not work as well, especially in combination with the mincut heuristic.

We experimented with the simple balance heuristic of selecting the best of two cuts at every recursive call of the algorithm. The resulting implementation was usually slower than GH, although not by much, and never significantly faster. This is because the best of the two cuts is usually not much more balanced than the first cut. The Hao-Orlin algorithm provides more opportunities for finding a more balanced cut, and implementations similar to GHG deserve further investigation.

Padberg-Rinaldi heuristics [19] proved very useful for certain classes of global minimum cut problems. One can use these heuristics (in a somewhat restricted form) to speed up s-t cut computations in the Gomory-Hu algorithm. However, on the problems these heuristics are effective, their use tends to lead to less balanced cuts and worse running times. This is because the heuristics tend to contract together large subsets of nodes. Although we invested substantial effort, we could not use the heuristics to consistently speed up our codes.

Further study of heuristics for the Gomory-Hu algorithm may provide significant improvements.

References

[1] 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.

[2] D. L. Applegate and W. J. Cook. Personal communication. Rice University, 1997.

[3] C. S. Chekuri, A. V. Goldberg D. R. Karger, M. S. Levine, and C. Stein. Experimental Study of Minimum Cut Algorithms. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333, 1997.

[4] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.

[5] 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.

[6] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[7] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[8] A. V. Goldberg and S. Rao. Beyond the Flow Decomposition Barrier. In *Proc. 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 2–11, 1997.

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

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

[11] M. Groetschel. Personal communication. ZIB Berlin, 1997.

[12] D. Gusfield. Very Simple Methods for All Pairs Network Flow Analysis. *SIAM Journal on Computing*, 19:143–155, 1990.

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

[14] T. C. Hu. *Combinatorial Algorithms*. Addison-Wesley, Reading, MA, 1982.

[15] M. Juenger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. In *Proc. 1st Workshop on Algorithm Engineering, Venice, Italy*, 1997. Available via URL http://www.dsi.unive.it/ wae97/proceedings/.

[16] Matthew S. Levine. Experimental Study of Minimum Cut Algorithms. Technical Report MIT-LCS-TR-719, MIT Lab for Computer Science, 1997.

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

[18] 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.

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

Problem family	Generator	# nodes	# edges	Other parameters
BIKEWHE	bikewheelgen	32,64,...,1024	$2n-3$	
CYC1	cyclegen	64,...,4096	n	
DBLCYC	dblcyclegen	64,...,1024	$2n$	
IRREG	irregulargen	1000	4000-5000	$k = 8, 9, W \in [0 \ldots 1000]$
NOI1	noigen	100-800	density: 50%	$P = n, k = 1$
NOI2	noigen	100-800	density: 50%	$P = n, k = 2$
NOI3	noigen	500	6000-124000	$P = 1000, k = 1$
NOI4	noigen	500	6000-124000	$P = 1000, k = 2$
NOI5	noigen	500	62000	$P = 1000$ $k = 1, 3, ..., 100, 500$
NOI6	noigen	500	62000	$P = 5000, 2000, ..., 10, 1$ $k = 2$
PATH	pathgen	2000	20000	$P = 1,000$ $k \in [1 \ldots 2000]$
PR1	prgen	200,400,...,1000	density: 2%	$k = 1$
PR5	prgen	200,400,...,1000	density: 2%	$k = 2$
PR6	prgen	200,400,...,1000	density: 10%	$k = 2$
PR7	prgen	200,400,...,600	density: 50%	$k = 2$
PR8	prgen	200,400,...,600	density: 100%	$k = 2$
REG1	regulargen	301	301,...,90300	
REG2	regulargen	50,100,...,800	$50n$	
TREE	treegen	800	density: 50%	$k \in [1 \ldots 800]$
TSP	tsp-instances	500-13000	$\approx n$	
WHE	wheelgen	64,128,...,1024	$2n-2$	

Table 2: Problem families reported on in this paper. We experimented with more families, but do not report on some where the results were similar to the ones we include.

Appendix: Data Tables

In the following tables we present data for the TSP, TREE and PATH problem families.

Notes:

- The data reported has been averaged over multiple runs (5 different graph inputs were created each time at random) for every problem case.
- N and M denote the number of vertices and edges respectively.
- *Aver.N* and *Aver.M* are equal to the average size (vertices and edges, respectively) over all subgraphs during the min-cut computations. For GUS these values are equal to N and M. For GH, GHS and GHG they are often significantly smaller.
- $CutTime$ corresponds to the total running time required to find all the cuts; this time is dominated by the max-flow computations.
- $ManipTime$ is the time required to manipulate the cuts. This includes the time to build the tree and for GH, GHS and GHG also includes the time for the contractions done. Moreover, for GHG it includes the time to perform double-splitting, according to the two most balanced cuts found so far.
- *Relabels* and *Pushes* account for the total number of relabels and pushes during the min-cut computation.
- $TotalTime$ is the total CPU-time for each algorithm. $TotalTime$ should be slightly bigger than $CutTime + ManipTime$, ($TotalTime$ also includes initialization and final output).
- The TSP table also contains the names of the TSP-instances considered, and the PATH and TREE tables also show the value of parameter k.

TSP										
	Name	N	M	Aver.N	Aver.M	CutTime	ManTime	Relabels	Pushes	TotTime
gus	tsp.pr76.x.2	76	90	76.0	90.0	0.046	0.018	11266	14127	0.068
gh	tsp.pr76.x.2	76	90	26.3	47.1	0.014	0.018	3349	4188	0.034
ghs	tsp.pr76.x.2	76	90	30.8	53.5	0.020	0.018	4535	5709	0.042
ghg	tsp.pr76.x.2	76	90	20.8	35.3	0.014	0.010	3003	4422	0.026
gus	tsp.att532.x.1	532	787	532.0	787.0	4.028	0.930	1063351	1562818	4.990
gh	tsp.att532.x.1	532	787	44.0	90.3	0.400	0.544	113526	163446	0.972
ghs	tsp.att532.x.1	532	787	52.6	104.6	0.512	0.560	153862	209975	1.110
ghg	tsp.att532.x.1	532	787	19.4	37.5	0.242	0.508	41882	72700	0.778
gus	tsp.vm1084.x.1	1084	1252	1084.0	1252.0	16.850	5.778	4620731	6759490	22.722
gh	tsp.vm1084.x.1	1084	1252	151.0	251.2	1.784	3.912	692992	791288	5.760
ghs	tsp.vm1084.x.1	1084	1252	148.4	244.4	2.174	3.898	729703	837499	6.126
ghg	tsp.vm1084.x.1	1084	1252	69.0	113.0	2.642	3.758	585930	1019210	6.462
gus	tsp.d1291.x.1	1291	1942	1291.0	1942.0	34.922	10.210	8799141	13794299	45.244
gh	tsp.d1291.x.1	1291	1942	49.6	108.0	1.692	6.858	468269	722254	8.596
ghs	tsp.d1291.x.1	1291	1942	65.6	133.6	2.528	6.812	698900	1090299	9.396
ghg	tsp.d1291.x.1	1291	1942	32.1	65.5	1.768	5.398	354765	633638	7.236
gus	tsp.rl1323.x.1	1323	2169	1323.0	2169.0	48.742	11.360	11847414	18843579	60.212
gh	tsp.rl1323.x.1	1323	2169	145.0	347.9	7.476	7.862	1982525	3171867	15.410
ghs	tsp.rl1323.x.1	1323	2169	179.6	400.5	6.494	8.012	1674102	2568434	14.568
ghg	tsp.rl1323.x.1	1323	2169	84.6	187.3	5.492	6.324	1184215	2043948	11.884
gus	tsp.rl1323.x.2	1323	2195	1323.0	2195.0	54.468	11.472	13154067	21379535	66.032
gh	tsp.rl1323.x.2	1323	2195	136.018	331.1	6.970	7.898	1858710	2970391	14.936
ghs	tsp.rl1323.x.2	1323	2195	161.6	367.1	10.320	7.904	2587602	4283288	18.274
ghg	tsp.rl1323.x.2	1323	2195	87.0	198.2	8.554	6.276	1809131	3182704	14.888
gus	tsp.fl1400.x.1	1400	2231	1400.0	2231.0	22.182	13.050	5611206	7733388	35.336
gh	tsp.fl1400.x.1	1400	2231	104.3	221.5	2.770	9.140	882686	1215741	11.982
ghs	tsp.fl1400.x.1	1400	2231	153.7	296.6	3.902	9.102	1178439	1543050	13.072
ghg	tsp.fl1400.x.1	1400	2231	54.1	107.2	2.134	7.360	440222	722726	9.562
gus	tsp.vm1748.x.1	1748	2336	1748.0	2336.0	57.994	22.826	14378206	21863109	80.952
gh	tsp.vm1748.x.1	1748	2336	84.0	179.3	5.760	16.740	1690678	2613124	22.568
ghs	tsp.vm1748.x.1	1748	2336	146.6	276.7	7.700	17.180	2141324	3188744	24.974
ghg	tsp.vm1748.x.1	1748	2336	83.9	156.1	6.270	13.430	1292352	2156314	19.782
gus	tsp.r15934.x.1	5934	7287	5934.0	7287.0	872.524	259.342	215146904	339530461	1132.362
gh	tsp.r15934.x.1	5934	7287	94.7	179.3	21.992	198.364	6548048	9811932	220.642
ghs	tsp.r15934.x.1	5934	7287	166.7	290.1	34.826	199.054	10326695	15343457	234.178
ghg	tsp.r15934.x.1	5934	7287	74.8	132.4	19.593	167.161	3690344	6859845	186.579
gus	tsp.r15934.x.2	5934	7627	5934.0	7627.0	1361.879	264.833	323352112	539711410	1627.174
gh	tsp.r15934.x.2	5934	7627	124.6	248.8	38.140	199.346	11642651	16912327	237.774
ghs	tsp.r15934.x.2	5934	7627	160.5	294.5	72.360	199.152	21090031	33203500	271.778
ghg	tsp.r15934.x.2	5934	7627	75.9	136.7	36.728	167.710	7753845	14153522	204.682
gus	usa13509.xo.15631	13509	15631	13509.0	15631.0	2530.176	1502.263	531347283	678321694	4033.327
gh	usa13509.xo.15631	13509	15631	214.0	390.3	104.680	1052.889	30634681	39561542	1158.174
ghs	usa13509.xo.15631	13509	15631	381.4	662.4	147.000	1063.709	45549472	53759023	1211.342
ghg	usa13509.xo.15631	13509	15631	109.5	187.9	99.529	892.320	20050422	34115253	992.468
gus	usa13509.xo.17494	13509	17494	13509.0	17494.0	2936.825	1636.194	549530523	709759847	4574.013
gh	usa13509.xo.17494	13509	17494	509.4	1011.7	295.418	1148.741	69743866	96070354	1444.972
ghs	usa13509.xo.17494	13509	17494	1029.4	1907.0	340.760	1189.828	81149905	103254869	1531.428
ghg	usa13509.xo.17494	13509	17494	316.1	571.8	440.779	1023.730	80970759	138601854	1465.180
gus	d15112.xo.19057	15112	19057	15112.0	19057.0	3268.710	1935.979	646836745	816539734	5205.931
gh	d15112.xo.19057	15112	19057	765.8	1491.1	405.040	1415.519	96301486	128562817	1821.206
ghs	d15112.xo.19057	15112	19057	1470.4	2678.7	564.729	1465.350	130209542	170710980	2030.944
ghg	d15112.xo.19057	15112	19057	512.1	915.0	991.442	1254.776	191025627	326933722	2247.079

Table 3: Data for TSP family

TREE										
	N	M	K	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	800	160600	1	800.000	160600.000	14.668	131.290	127.200	251171.000	146.026
gh	800	160600	1	800.000	126637.999	139.682	225.440	654496.800	3382659.200	365.164
ghs	800	160600	1	800.000	126635.997	51.730	215.438	127.200	251171.800	267.196
ghg	800	160600	1	400.501	63397.247	195.330	164.424	353357.400	31922655.000	359.770
gus	800	160600	3	800.000	160600.000	46.530	131.504	170829.800	794471.000	178.098
gh	800	160600	3	793.240	125034.007	167.616	224.196	765747.200	4583496.400	391.840
ghs	800	160600	3	403.766	38863.526	18.934	73.984	7531.600	284183.800	92.936
ghg	800	160600	3	202.330	19504.642	50.922	55.570	130895.800	8917101.400	106.522
gus	800	160600	5	800.000	160600.000	59.126	131.236	237671.400	1124891.400	190.412
gh	800	160600	5	792.881	124943.023	185.174	219.818	849349.400	5763163.000	405.030
ghs	800	160600	5	296.024	22990.661	11.962	44.134	11237.400	313122.400	56.118
ghg	800	160600	5	148.798	11610.357	28.988	32.786	91232.200	5175053.600	61.796
gus	800	160600	10	800.000	160600.000	80.516	131.440	349191.000	1602387.400	212.018
gh	800	160600	10	789.170	124159.165	219.236	220.068	968735.200	6947990.200	439.336
ghs	800	160600	10	194.260	11909.418	7.200	23.224	16703.400	355732.000	30.476
ghg	800	160600	10	98.268	6132.021	15.710	17.238	67740.800	2736431.000	32.962
gus	800	160600	20	800.000	160600.000	98.936	131.174	451982.800	2125038.400	230.176
gh	800	160600	20	776.144	121513.256	245.868	216.928	1055504.200	7735155.000	462.838
ghs	800	160600	20	131.732	7341.176	5.842	14.622	22917.200	408319.600	20.496
ghg	800	160600	20	68.127	3990.482	12.174	11.434	61921.600	1857187.800	23.626
gus	800	160600	50	800.000	160600.000	115.680	131.382	545664.400	2696860.800	247.110
gh	800	160600	50	727.921	112241.486	268.230	202.242	1109984.000	8832373.800	470.506
ghs	800	160600	50	99.229	6610.697	7.598	13.654	37106.600	507136.800	21.286
ghg	800	160600	50	54.427	3978.673	17.628	11.798	87508.600	1859914.000	29.450
gus	800	160600	100	800.000	160600.000	126.788	131.646	598343.800	3171652.000	258.488
gh	800	160600	100	664.094	101533.409	258.378	186.488	1054070.200	8568625.000	444.904
ghs	800	160600	100	102.320	8889.526	11.888	18.540	55722.200	616169.800	30.464
ghg	800	160600	100	60.021	5679.074	35.894	17.040	162158.400	3017374.200	52.946
gus	800	160600	200	800.000	160600.000	138.074	131.416	644957.200	3737396.000	269.544
gh	800	160600	200	594.630	91044.894	259.294	172.300	1005457.800	8267321.000	431.610
ghs	800	160600	200	121.064	13033.333	20.328	27.702	86009.000	798572.000	48.064
ghg	800	160600	200	75.154	8543.694	77.432	26.386	303894.200	5516712.800	103.842
gus	800	160600	400	800.000	160600.000	154.596	131.048	704840.200	4323066.800	285.720
gh	800	160600	400	508.545	78112.801	242.806	152.198	930439.000	7687778.200	395.044
ghs	800	160600	400	156.707	19788.667	32.374	42.148	123498.200	1073789.400	74.558
ghg	800	160600	400	98.211	12798.786	144.732	39.998	502969.000	9447691.200	184.744
gus	800	160600	800	800.000	160600.000	185.920	131.012	809714.600	5163665.600	316.990
gh	800	160600	800	449.869	69666.439	238.776	137.512	895883.200	7362390.400	376.320
ghs	800	160600	800	192.218	26378.757	46.742	55.636	166621.600	1424371.000	102.430
ghg	800	160600	800	115.630	16127.198	212.824	49.914	696121.400	12522017.200	262.764

Table 4: Data for TREE family

PATH										
	N	M	K	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	2000	21990	1	2000.000	21990.000	2.738	71.926	11.000	39757.600	74.844
gh	2000	21990	1	2000.000	22864.000	53.860	111.120	3462058.200	5089280.400	165.076
ghs	2000	21990	1	2000.000	22862.001	19.528	112.792	11.000	39757.600	132.434
ghg	2000	21990	1	1000.500	11436.718	66.922	79.284	1674373.600	10123340.200	146.248
gus	2000	21990	4	2000.000	21990.000	14.940	71.836	757042.200	2022180.800	86.910
gh	2000	21990	4	1989.436	22709.442	80.414	110.346	5323845.600	9611058.800	190.858
ghs	2000	21990	4	504.145	2776.646	2.264	31.808	7482.600	73443.800	34.180
ghg	2000	21990	4	253.030	1398.473	5.224	16.604	254095.200	664068.200	21.886
gus	2000	21990	15	2000.000	21990.000	46.264	73.574	2950639.800	6537896.600	119.994
gh	2000	21990	15	1778.399	20386.260	130.034	100.612	7040216.000	16579151.400	230.760
ghs	2000	21990	15	143.347	628.790	1.196	24.778	30778.800	173520.200	26.070
ghg	2000	21990	15	75.218	354.884	2.288	11.702	118590.800	385170.600	14.042
gus	2000	21990	50	2000.000	21990.000	83.076	72.760	5057902.600	10189574.200	155.996
gh	2000	21990	50	1260.434	15504.246	136.430	81.156	6536603.200	18526364.400	217.706
ghs	2000	21990	50	60.545	381.101	2.194	23.872	90910.400	403747.600	26.186
ghg	2000	21990	50	38.562	294.205	3.900	11.676	177543.400	722060.400	15.636
gus	2000	21990	200	2000.000	21990.000	110.852	71.096	6804711.800	12232899.800	182.078
gh	2000	21990	200	266.618	3667.483	42.074	35.896	2037790.200	6330626.400	78.070
ghs	2000	21990	200	43.030	458.313	4.270	24.228	196448.200	636261.600	28.586
ghg	2000	21990	200	34.977	402.492	8.172	12.656	347545.600	1285304.800	20.882
gus	2000	21990	800	2000.000	21990.000	242.030	71.264	13343243.800	24378523.800	313.436
gh	2000	21990	800	124.977	1846.756	36.848	29.216	1658893.600	4227637.400	66.180
ghs	2000	21990	800	64.565	956.401	11.258	25.950	499685.400	1265666.400	37.322
ghg	2000	21990	800	50.957	731.543	32.882	15.852	1243339.800	3886804.200	48.794
gus	2000	21990	2000	2000.000	21990.000	554.889	70.765	28851668.800	50298108.400	625.804
gh	2000	21990	2000	126.658	2030.558	62.116	29.720	2671666.200	5836707.400	91.926
ghs	2000	21990	2000	103.921	1701.291	34.484	28.802	1501881.400	3255455.400	63.360
ghg	2000	21990	2000	67.000	1053.022	89.398	19.110	3371604.600	8772053.800	108.568

Table 5: Data for PATH family