Experimental Evaluation of a Parametric Flow Algorithm

Maxim A. Babenko [1] Andrew V. Goldberg[2]

June 2006

Technical Report
MSR-TR-2006-77

We study a practical implementation of the parametric flow algorithm of Gallo, Grigoriadis, and Tarjan. We describe an efficient implementation of the algorithm and compare it with a simpler algorithm.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]Moscow State University, Moscow, Russia. Part of this work was done while the author was visiting Microsoft Research. Email: `mab@shade.msu.ru`.

[2]Microsoft Research, 1065 La Avenida, Mountain View, CA 94062. Email: `goldberg@microsoft.com`; URL: `http://www.research.microsoft.com/~goldberg/`.

1 Introduction

The parametric flow problem is an important combinatorial optimization problem with many applications [1, 2, 4, 5, 7, 8, 11, 12, 13, 15, 17, 18, 16, 19, 21].

The best bound for the problem is achieved by an algorithm of Gallo, Grigoriadis, and Tarjan [6] (*GGT*). This algorithm uses a clever recursion to amortize a parametric flow computation over a push-relabel maximum flow computation and matches the $O(nm\log(n^2/m))$ bound for the latter [9]. Here n and m are the number of vertices and arcs in the input network, respectively. As the algorithm is fairly sophisticated, some implementation issues are not explicitly discussed in [6]. In this paper we discuss all implementation issues and present experimental results which highlight trade-offs and bottlenecks of the algorithm. We also compare the algorithm with a simple algorithm that does not use sophisticated amortization.

A very different algorithm has been proposed in [22, 23]. Although its running time bound is worse than that of GGT, the authors claim that the algorithm has a good practical performance.

2 Definitions and Notation

In this paper we consider directed graphs. Given a graph $G = (V, E)$, let $n = |V|$ and $m = |E|$. A capacity function is a function u from arcs to positive reals.

A *cut* is a partitioning of vertices $S, \overline{S} = V - S$. A cut is nontrivial if both S and $\overline{S}$ are nonempty. The capacity of the cut is defined by

$$u(S, \overline{S}) = \sum_{(v,w)\in E\cap(S\times\overline{S})} u(v, w).$$

An s-t cut is a partitioning of vertices $(S, \overline{S})$ such that $s \in S$ and $t \in \overline{S}$.

Given two vertices s and t, an *s-t flow* is a real-valued function f on arcs that satisfies capacity constraints: for all arcs a, $0 \le f(a) \le u(a)$ and conservation constraints for all vertices other that s and t: the sum of flows over incoming arcs is equal to the sum over outgoing arcs.

In a parametric flow problem we consider, capacities of arcs adjacent to s and t are functions of a parameter λ: Arcs (s, v_i) have capacities $a_i + b_i\lambda$ and arcs (w_j, t) have capacities $a_j - b_j\lambda$ for nonnegative real-valued a's and b's. Note that the former are nondecreasing, and the latter non-increasing functions of λ. Arcs not adjacent to s and t have constant capacities. When talking about a flow maximum flow of a minimum cut in a parametric network, we mean the flow or the cut of a specific value of λ.

It is well-known that the maximum flow value in a parametric network is a continuous piecewise linear function of λ. Each linear segment of the function between two breakpoints, λ' and λ'', corresponds to a cut that remains a minimum cut for $\lambda' \le \lambda\lambda''$. The parametric flow problem is to find the breakpoints and the corresponding cuts. Two important cases of the problem is to find the minimum and the maximum breakpoint.

3 Push-Relabel Algorithm

The GGT algorithm is based on the push-relabel algorithm [10] for the maximum flow problem. The push-relabel algorithm uses two basic operations, *push* and *relabel*, and maintains a flow and integral *distance labels* on vertices. The important properties of the algorithm are that the distance labels are monotonically increasing, the value of each distance label changes by $O(n)$, and the work of the algorithm is charged to the distance label increase. Using the dynamic tree data structure [20], the algorithm can be implemented to run in $O(nm \log(n^2/m))$ time.

We assume that the reader is familiar with the push-relabel algorithm as discussed in [10] or [6].

4 Largest Breakpoint

In this section we describe two algorithms for finding the largest breakpoint: a simple algorithm and its GGT variant.

First consider the simple algorithm. We maintain two values, λ_1 and λ_3, such that $\lambda_1 \leq \lambda_3$ and the desired breakpoint is between these values. See [6] for the initial values of λ_1, λ_3. We repeatedly increase the value of λ_1 until this value reaches the largest breakpoint.

Note that the trivial cut $(V - \{t\}, \{t\})$ is a minimum cut for λ_3. Denote the capacity of this cut, as a function of λ, as $a_3 + \lambda b_3$.

We do the following step (i): Compute a minimum cut for λ_1 and let the capacity of this cut be $a_1 + \lambda b_1$. If $a_1 = a_3$ and $b_1 = b_3$ stop and output λ_3. Otherwise replace λ_1 by the solution of $a_1 + \lambda b_1 = a_3 + \lambda b_3$ (i.e., $(a_3 - a_1)/(b_3 - b_1)$) and repeat.

One can show that the algorithm terminates in $O(n)$ iterations; see [6].

The GGT algorithm is a variant of the simple algorithm that uses the push-relabel algorithm [10] and amortizes the work of multiple flow computations. This is possible because the value of λ_1 monotonically increases, and because of this the algorithm can be restarted from one flow computation to another in $O(m)$ time while keeping the previous label values. Thus the only work not amortized over distance label increases is the time spend restarting the computations, which is bounded by $O(nm)$.

5 Computing All Breakpoints

A simple algorithm for computing all breakpoints works recursively. At each call, the algorithm gets an interval (λ_1, λ_3) and cuts corresponding to λ_1 and λ_3, and outputs all breakpoints in the interval. Initial values of λ_1, λ_3 which are less then and greater than all breakpoints, respectively, are easy to find (see [6]).

Let $a_1 + \lambda b_1$ and $a_3 + \lambda b_3$ be parametric capacities of the two input cuts. Set $\lambda_2 = (a_3 - a_1)/(b_3 - b_1)$ and compute the minimum cut corresponding to λ_2. If the parametric capacity of the cut is not equal to $a_1 + \lambda b_1$ or $a_3 + \lambda b_3$, then λ_2 is not a breakpoint, and we recursively find all breakpoints on (λ_1, λ_2) and on (λ_2, λ_3). Otherwise, it is a breakpoint, and we output it. Then, if the capacity is equal to $a_1 + \lambda b_1$, we recurse on the interval (λ_2, λ_3). In the other case, we recurse on (λ_1, λ_2). When making a recursive call for the interval (λ_1, λ_2), we contract the vertices on the sink side of the minimum cut corresponding to λ_2. Similarly, when making the other recursive call, we contract vertices on the source side. Each call of the algorithm is dominated by a minimum cut computation, and one can show that the number of calls is $O(n)$.

Next we describe the GGT algorithm. The algorithm uses amortization. One way to use amortization in the context of the simple algorithm is to note that when recursing on (λ_2, λ_3), one can use the distance labels (on the sink side of the computed cut) from the current flow computation and amortize the cost of such recursive calls over one maximum flow computation. Note that the distance labels on the source side of the cut are "infinite" so the other recursive call cannot be amortized. To obtain the desired bound, the GGT algorithm makes sure that the cost of the flow computation on the bigger graph is amortized.

To achieve this, the algorithm runs two flow computations in parallel; forward from the source and backward from the sink. Assume that the forward computation finishes first; the other case is symmetric. Then if the sink side of the resulting cut has at least as many vertices as the source side, we disregard the result of the backward computation. Otherwise, we finish the backward computation and keep the labels on the source side of the cut, which is at least as big as the sink side. This way the GGT algorithm amortizes the cost of the bigger recursive call at each level, leading to the desired time bound. See [6] for details.

6 Implementation Issues

Our code was written in `C++` and compiled using the `cygwin g++` compiler with `-O4` optimization option. The machine used in the experiments was HP Evo D530 with a 3.2 GHz Pentium 4 CPU and 1 GB RAM, running Windows XP Service Pack 2.

AS the initial point of our parametric flow codes, we used an implementation of the push-relabel algorithm described in [3]. In particular, we used the gap and the global update heuristics; see [3].

We implemented two versions of the Gallo-Grigoriadis-Tarjan algorithm: the complete version (GGT) that uses amortization and bidirectional flow computations and the simple version (SIMP), that starts each maximum flow from scratch and uses the forward computation only. We also implemented a variant GGT-M of the GGT algorithm that computed the maximum breakpoint using amortization (but not bidirectional flow computations, which are unnecessary) and SIMP-M that does not use amortization. The general algorithms use graph contraction and the "M" variants do not.

Dealing with precision The above discussion assumes unlimited precision arithmetic. Because of the multiplicative factors in parametric capacities, one may need high precision to distinguish between adjacent breakpoints. However, using high-precision arithmetic is expensive, and in some applications one may not need to distinguish between breakpoint values which are close together. Our approach is to use 64-bit integer arithmetic and distinguish only between breakpoints which are far enough apart. Our implementation can miss some breakpoints, but for each missed breakpoint we find a value that is close. Note that using (even double precision) floating point arithmetic does not avoid numerical issues and may lead to correctness and termination problems.

Our implementation starts by selecting an integer multiplier M and multiplying all capacities by M. The value of M is selected so that for the highest value of λ the total capacity of arcs from the source is less than 2^{62}, and for the lowest value of λ the same holds for the arcs into the sink. This choice of M guarantees that flow excesses do not exceed 2^{62}, overflow errors will be detected, and our correctness checker, which needs an extra bit of precision, can be implemented.

During the algorithm initialization, when calculating the initial range, we round λ_1 down and λ_3 up to the nearest integer. During the algorithm execution, we round the value of λ_2 down.

Note that because of the rounding, a value x we output may not be a breakpoint. However, the following properties hold.

1. If we output a value x, then there is a breakpoint within in the interval $[x - 1/M, x + 1/M]$.
2. For every breakpoint y, we output a value in $[y - 1/M, y + 1/M]$.
3. For every two distinct x_1 and x_2 we output, there are corresponding minimum cuts $(X_1, \overline{X_1})$, $(X_2, \overline{X_2})$ such that parametric capacities of the two cuts are different.

Note that if we restrict the precision of values we output, then this is the best we can do.

In addition to outputting the approximate breakpoint parameter values, we build a data structure containing the corresponding cuts. Since the cuts are nested, the data structure is an ordered list of vertices, with a pointer to the last vertex of the source-side set for each cut. Note that if all distinct breakpoints are at least $2/M$ apart, the cuts correspond to the true breakpoint values, and can be used to compute the exact breakpoint values.

7 Experimental Results

7.1 Problem Families

We used problem generators and problem families from the First DIMACS Network Flow Challenge [14]. The problem families we use are AC-DENSE, RMF-LONG, RMF-WIDE, WASH-LINE, WASH-RLG-LONG, WASH-RLG-WIDE. We make the problems parametric as follows.

For all problems except AC-DENSE (complete acyclic graphs), we randomly partition interior vertices into two groups and add arcs from s to the first group and from the second group to

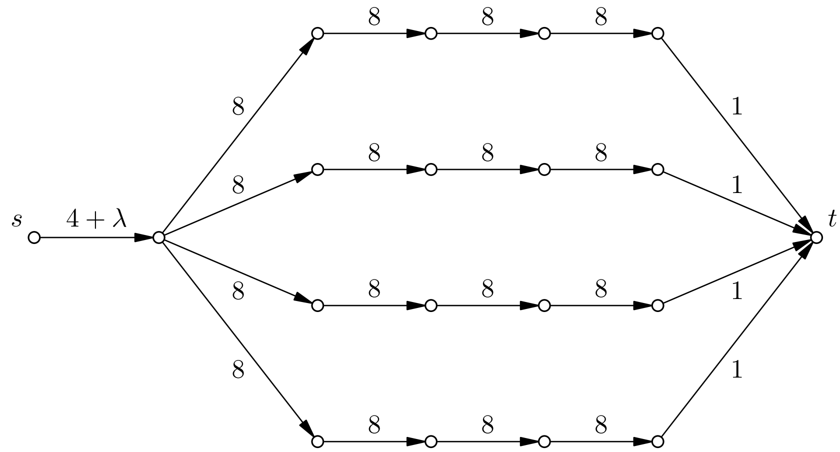

Figure 1: Asymmetric graph with $x = 4$. For non-parametric version, take $\lambda = x$

t. Then, for all problems, we parameterize the source and the sink arcs by choosing a and b coefficients independently at random from the range $[0, 10, 000]$.

Since the number of breakpoints is bounded by the number of parameterized arcs, adding such arcs introduces potential for many breakpoints. Note that if the number of breakpoints is very small, theoretical advantages of the GGT algorithm are small as well.

Finally, we use the following ASYM problem generator that produces asymmetric problems which are more difficult for the forward push-relabel algorithm implementation than for the reverse one. The non-parametric version of the ASYM problem with parameter x, illustrated on Figure 1, has a source s with the only arc (s, v) of capacity $2x$. The vertex v is the origin of x disjoint paths, each of length x, with arcs of capacity $2x$. The end vertex of each path is connected to t by an arc of capacity 1.

Intuitively, the forward push-relabel algorithm moves a large flow excess along one of the paths, saturates the arc to the sink, returns the rest of the excess to v, and takes the next path. The reverse algorithm moves a unit of flow along each path all way to the source. Without heuristics, the forward algorithm runs in $\Theta(x^3)$ time and the reverse algorithm in $O(x^2)$ (linear) time. Things are a little more complicated because of the heuristics used by the implementation, but the forward algorithm is significantly slower than the reverse algorithm.

The ASYM generator produces a parametric version of the problem by making the (s, v) arc capacity equal to $x + \lambda$.

7.2 Results

Tables 1 – 7 contain experimental data. For each problem family, we give running times (top) and the number of relabel operations (bottom) for algorithms GGT and SIMP that find all breakpoints, algorithms GGT-M and SIMP-M that find the maximum breakpoint, and the algorithm MF that

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
15488	205298	7334	0.153	4.951	2.478	0.339	0.330
			29143	345424	396200	46441	70742
30589	409056	9129	0.267	9.215	4.654	0.812	0.815
			58245	1371254	1339468	183420	266585
65536	884196	6788	0.512	23.289	11.642	3.376	3.557
			127601	7869499	6625234	1335628	1858846
130682	1773774	3573	1.117	61.901	30.332	12.391	12.395
			362427	29581882	23292892	5785413	7300493
270848	3696578	1063	10.418	161.004	75.728	36.154	33.607
			7281712	90397185	65670911	17918257	20825991
527796	7231274	713	56.456	358.037	159.192	80.703	73.261
			41753770	193141563	136617222	39847861	44305014

Table 1: Results for RMF-LONG family. Running time in seconds (top), # of relabel operations (bottom).

finds a maximum flow given the maximum breakpoint capacities. We average running times and operation counts over 10 runs for each problem size; for randomized generators, the runs use different seeds.

We use operation counts as a machine-independent measure of performance. In our experiments, the operation counts are strongly correlated with the running times, showing that our implementations are efficient.

First we discuss all problem families except ASYM. Problem families AC-DENSE, RMF-WIDE, WASH-LINE have a small number of breakpoints. The WASH-RLG-LONG and WASH-RLG-WIDE families have a relatively large number of breakpoints, about 10% or more of the number of vertices. The only exception is the largest problem in the latter family, which has somewhat fewer breakpoints, about 6.5%. For the RMF-LONG family, the number of breakpoints is relatively large for smaller problem sizes and decreases to moderate (compared to the number of vertices) for larger sizes.

We make the following observations:

- Usually GGT is slower than SIMP, but by less than a factor of two. The cost per operation is higher for GGT, but not by much.
- Usually GGT-M and SIMP-M have comparable running times. While the former usually performs fewer operations, the cost per operation is somewhat higher.
- While on some problem families MF is not that much faster than GGT-M and SIMP-M, on others it is significantly faster; e.g., WASH-RLG-LONG, on which the performance gap seems to increase with the problem size.

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
8214	110454	6	0.139	1.342	0.640	0.378	0.409
			71100	385043	271531	109967	168155
16807	227724	6	0.315	3.276	1.760	1.134	1.182
			129233	834678	726784	337661	484712
32768	446436	8	0.757	7.254	4.273	2.587	3.004
			297633	2026043	1691530	789100	1149462
65025	889752	9	0.1829	19.001	11.237	7.146	7.381
			665122	5763333	4487802	2107138	2660582
123210	1691390	9	4.670	38.229	26.445	12.497	17.090
			1490775	12103598	10420263	3541627	6307795
295788	4076634	7	22.653	133.271	89.614	36.998	61.690
			6096589	38929374	26750363	9165453	19658993

Table 2: Results for RMF-WIDE family. Running time in seconds (top), # of relabel operations (bottom).

- Usually GGT (SIMP) is not too much slower than GGT-M (SIMP-M). The biggest difference is about an order of magnitude, for the smallest RMF-LONG problems, where the number of breakpoints is large. Usually there is less of a difference. For the WASH-RLG-WIDE family, where the number of breakpoints grows linearly with n, the ratio between GGT and GGT-M stays at around a factor of 6.

- Comparing GGT to SIMP, we see that operation counts show that the additional overhead of GGT amortization is not too big – the ratio between running times and operation counts for GGT and SIMP are relatively close.

The ASYM generator is somewhat artificial, designed to show that GGT can be much faster than SIMP. Note that GGT is faster than MF. This is due to the fact that MF solves the input problem in the “hard” direction, where as GGT uses the bidirectional approach. Also note that GGT-M is a unidirectional algorithm, and it looses to GGT.

The ASYM family brings up an interesting point: the push-relabel algorithm is not symmetric in a sense that its running time may be very different from that for running the algorithm on the reversed graph. It would be interesting to get an efficient natural algorithm for which the two running times are similar. (Here we exclude running the forward and the reverse algorithms in parallel.)

8 Conclusions

We described an efficient implementation of the GGT algorithm. Although the parametric maximum flow problem is more general, and sometimes requires significantly more time to solve than

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
16386	163456	1668	0.082	3.639	2.074	0.528	0.451
			30564	1043884	1106323	252708	257422
32770	327296	3179	0.157	8.282	4.956	1.418	1.139
			61462	2881401	3048085	762869	742890
65538	654976	6162	0.340	19.976	12.899	3.636	2.988
			123011	7988775	8797262	2051302	2059408
131074	1310336	12164	0.675	51.340	31.951	9.137	7.403
			245986	23481775	23271570	5181809	5225486
262146	2621056	23666	1.376	131.171	80.803	23.390	19.001
			491132	67041926	61556732	13594976	13542167

Table 3: Results for WASH-RLG-LONG family. Running time in seconds (top), # of relabel operations (bottom).

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
65538	649216	8333	0.484	18.698	13.267	4.392	4.238
			120678	2856875	3343559	1217765	1431843
131074	1298432	16697	1.089	44.320	31.079	9.462	10.226
			240888	6850661	8013778	2381789	3420479
262146	2596864	31577	2.301	95.948	68.007	24.151	23.574
			481708	14874371	16831065	6528741	7847399
524290	5193728	34054	4.660	204.623	138.809	52.890	55.922
			964572	32471877	36889676	14044047	19571322

Table 4: Results for WASH-RLG-WIDE family. Running time in seconds (top), # of relabel operations (bottom).

the maximum flow problem, the GGT algorithm is able to solve problems with millions of arcs and thousands of breakpoints in minutes.

We constructed the ASYM problem family on which GGT is much faster than SIMP. It would be interesting to construct a family with the same property which, in addition, is more robust and natural. Also, it would be nice to construct a problem family where GGT-M will be much faster than SIMP-M, something that the ASYM family does not achieve.

Our experiments suggest that on many problem types, the additional amortization of GGT is not necessary, as the SIMP implementation is competitive or faster. On the other hand, the amortization hurts GGT only by a factor of two in memory usage and usually by less than a factor of two in running time. It is unclear, and probably application-specific, if the extra robustness of the theoretically superior algorithm is sufficient to recommend GGT over SIMP for real-life applications.

The result of [6] can be interpreted as saying that in the worst case, the complexity of the

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
4098	146417	8	0.101	0.756	0.424	0.202	0.215
			19401	57031	49915	17380	24161
8194	407448	7	0.351	1.889	1.170	0.600	0.619
			51219	83080	98445	39433	52940
16386	1109979	6	1.048	4.735	3.012	2.046	2.126
			99308	127886	154890	89669	121346
32770	3072052	6	3.472	14.936	10.585	6.867	7.864
			213810	167297	362233	207524	304405
65538	8634321	6	14.803	51.934	33.814	22.970	26.909
			630820	613118	803636	489045	740314

Table 5: Results for WASH-LINE family. Running time in seconds (top), # of relabel operations (bottom).

parametric flow problem is not much worse than that of the maximum flow problem. Our results can be interpreted as saying that in practice, one will see a noticeable difference in performance if the number of breakpoints is large, but the performance ratio will be much smaller than the number of breakpoints.

Acknowledgments

We would like to thank Bob Tarjan for helpful discussions.

References

[1] J. R. Brown. The Sharing Probkem. *Oper. Res.*, 27:324–340, 1979.

[2] P. Chaillou, P. Hansen, and Y. Manieu. Best Netwotk Flow Bounds for Quadratic Knapack Problem. In *Proc. NETFLO 83, Piza, Italy*, 1983.

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

[4] W. H. Cunningham. Optimal Attack and Reinforcement of a Network. *J. Assoc. Comput. Mach.*, 32:549–561, 1985.

[5] M. J. Eisner and D. C. Severance. Mathematical Techniques for Efficient Record Segmentation in Large Shared Database. *J. Assoc. Comput. Mach.*, 23:619–635, 1976.

[6] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A Fast Parametric Maximum Flow Algorithm and Applications. *SIAM J. Comput.*, 18:30–55, 1989.

[7] G. Gallo, P. Hammer, and B. Simeone. Quadratic Knapsack Problem. *Math. Prog.*, 12:132–149, 1980.

[8] A. V. Goldberg. Finding a Maximum Density Subgraph. Technical Report UCB/CSD/84/171, Computer Science Division, U.C. Berkeley, 1984.

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

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
512	261630	6	0.228	1.739	0.904	0.490	0.547
			3419	8167	8018	3178	4891
722	520560	7	0.464	3.863	2.049	1.034	1.136
			75328	12679	14258	5272	7642
1024	1047550	6	1.139	7.957	4.325	2.304	2.676
			8956	18568	18846	7317	11619
1444	2083690	7	2.742	20.020	10.893	5.338	6.012
			14866	32382	34858	11993	18504
2048	4192254	7	7.157	47.740	26.782	12.932	14.504
			23306	44208	49730	17673	25934
2888	8337654	8	17.943	115.609	65.968	32.660	37.231
			34151	67481	73396	27889	39821

Table 6: Results for AC-DENSE family. Running time in seconds (top), # of relabel operations (bottom).

[10] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Canceling Negative Cycles. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 388–397, 1988.

[11] D. Gusfield. On scheduling transmissions in a network. Technical Report YALEU DCS TR 481, Department of Computer Science, Yale University, 1986.

[12] T. Ichimori, H. Ishii, and T. Nishida. Optimal Sharing. *Math. Prog.*, 23:341–348, 1982.

[13] A. Itai and M. Rodeh. Sceduling transmissions in a network. *J. Algorithms*, 6:409–429, 1985.

[14] D. S. Johnson and C. C. McGeoch. *Network Flows and Matching: First DIMACS Implementation Challenge*. AMS, 1993. Proceedings of the 1-st DIMACS Implementation Challenge.

[15] N. Megiddo. Optimal Flows in Networks with Multiple Sources and Sinks. *Math. Prog.*, 7:97–107, 1974.

[16] J. C. Picard and H. D. Ratliff. Minimum Cuts and Related Problems. *Networks*, 5:357–370, 1975.

[17] J.C. Pickard and M. Queyranne. A Network Flow Solution to Some Nonlinear 0-1 Programming Porblems, with Applications to Graph Theory. *Networks*, 12:141–159, 1982.

[18] J.C. Pickard and M. Queyranne. Selected Applications of Minimum Cuts in Networks. *INFOR*, 20:394–422, 1982.

[19] J.B. Sidney. Decomposition Algorithm for Single-Machine Sequencing with Presedence Relations and Defferal Costs. *Oper. Res.*, 23:283–298, 1975.

[20] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[21] H.S. Stone. Critical load factors in two-processor distributed systems. *IEEE Trans. Soft Eng.*, 4:254–258, 1978.

[22] B. Zhang, J. Ward, and Q. Feng. A Simultaneous Parametric Maximum-Flow Algorithm for Finding the Complete Chain of Solutions. Technical Report HPL-2004-1101, HP Labs, Palo Alto, CA, 2004.

[23] B. Zhang, J. Ward, and Q. Feng. Simultaneous Parametric Maximum-Flow Algorithm for with Vertex Balancing. Technical Report HPL-2005-121, HP Labs, Palo Alto, CA, 2005.

n	m	# bp	MF	GGT	SIMP	GGT-M	SIMP-M
19884	119290	1	0.476	0.293	0.620	0.806	1.081
			407901	29846	422836	631808	885508
40003	240004	1	1.356	0.590	1.656	2.265	3.314
			1131037	60033	1161071	1791513	2692093
79527	477148	1	3.526	1.140	4.109	5.751	8.378
			3078210	119330	3137899	4808728	7135863
160003	960004	1	11.694	2.584	13.093	18.896	28.171
			9323077	240061	9443142	14604877	22047238
319228	1915354	1	29.562	4.860	31.976	47.459	66.375
			25227626	478924	25467134	40314573	55640790
640003	3840004	1	109.042	12.284	115.454	176.031	113.656
			72658042	960121	73138169	116345221	73429121

Table 7: Results for ASYM family. Running time in seconds (top), # of relabel operations (bottom).