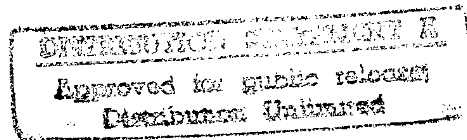
Experimental Evaluation of the Push-Relabel Method for the Minimum-Cost Flow Problem (Extended Abstract)

*Andrew V. Goldberg**
Michael Kharitonov

Department of Computer Science
Stanford University
Stanford, CA 94305

August 1991

19961114 047



*Research partially supported by NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T and DEC, ONR Young Investigator Award N00014-91-J-1855, and a grant from Mitsubishi Corp.

DTIC QUALITY INSPECTED 1

1 Introduction

The generalized cost-scaling method, introduced in [4, 7], is theoretically superior to other methods for solving the minimum-cost flow problem, in the sense that it leads to the best running time bounds currently known under the assumption that the absolute values of costs are not too big. (See also [1].) There is some evidence that the method should perform well in practice as well: an earlier cost-scaling algorithm of Blend and Jensen [2] was shown to be competitive with a well-established network simplex code. The method is very flexible and has many variants. Which version of the method works best in practice? But how good is the method in practice? These are the questions addressed in our study.

Generalized cost scaling framework can accommodate several very different algorithms, such as the push-relabel method [5, 7], the blocking flow method [3, 7, 10], multiple scaling algorithms [1], and cycle-canceling algorithms [6]. We restrict our study only to push-relabel variant of the cost-scaling approach. This variant is the most promising since, in the context of the closely related maximum flow problem, several researchers reported its superior performance compared to other popular techniques. (The most comprehensive study we are aware of was done by Michael Grigoriadis [personal communication].) For this reason we believe that the experimental study of the push-relabel variants of the method should be done first, and in this study restrict ourself to these variants.

The main advantage of the push-relabel method is its flexibility. The method allows arbitrary ordering of its basic operations. In addition, several heuristics can be used in attempt to reduce the number of basic operations. In the context of the maximum flow problem, a good choice of operation ordering and heuristics proved to be essential for obtaining an efficient implementation. The more general minimum-cost flow problem allows an even richer choice heuristics. The use of scaling also raises the question of the choice of the scaling factor. We study several operation orderings and heuristics proposed in [7], as well as new heuristics.

Although our study is not yet complete, we learned a lot. The practical performance of the method is much better then the worst-case bounds suggest. In particular, one scaling iteration seems to be not that much slower than a shortest path computation on the residual graph. Since the number of scaling iterations is small, the algorithm runs very fast. We expect the final implementation to be memory bound, so that any problem that fits into the main memory of a RISC workstation can be solved in a matter of minutes.

Since the practical behavior of the algorithm differs substantially from the worst-case behavior, some operations and heuristics with negligible worst-case cost become too expensive in practice. In some cases, the new bottleneck can be eliminated by handling such operations in a more sophisticated way. In one case, we overcame such a problem by introducing a data structure that speeds up the bottleneck operations substantially. Some heuristics improve the running times. Others do not. In some cases, a heuristic needs to be replaced by one that accomplishes a similar goal in a more "quick and dirty" way.

To summarize, our experimental results so far are very encouraging. We learned several interesting facts about the practical performance of the method and related heuristics. Our observations lead to practical improvements of some implementations and even to a revival of an earlier theoretical conjecture.

2 Preliminaries

We assume that the reader is familiar with [7] and use the definitions and notation from that paper.

3 Important Experimental Observations

During the course of our experiments we made several observations that help to explain the empirical behavior of the method and to improve selection strategies. We summarize the most relevant observations below.

3.1 Low Activity Level

The first observation is that, on the average, relatively few nodes are active during an execution of refine. This was observed earlier by Norbert Schienker and Bob Tarjan [personal communication] and Joel Wein [personal communication]. Even for moderate-size problems of several hundred nodes, about 1% of the nodes are active; the fraction of active nodes goes down as the problem size increases. Of course, the exact number of active nodes depends on the selection strategy and on the example structure. However, we found the number to be low for all the combinations we tried.

The low activity level has important implications. For example, the first-in, first-out (FIFO) selection strategy works better then the worst-case bound suggests. Also, because of the low activity level, non-trivial data structure is required for the first-active (FA) implementation, which otherwise spends most of its time scanning the node list in search of the next active node. These implications are discussed in more detail in the next section.

3.2 Outdated Prices

We say that the prices are *current* if every node with an excess from which a node with a deficit can be reached in the residual graph has an admissible arc going out of it. One way to compute current prices is to add ϵ to reduced costs, compute shortest path distances to the nodes with deficits, and add these distances to the corresponding prices. We call this computation *price update.* Note that if the prices are current, there is an admissible path from every node with excess to a node with deficit. In this case pushes move flow excesses towards flow deficits (with respect to the topological order of the admissible graph).

Prices are *outdated* if they differ significantly from any set of current prices. If the prices are outdated, flow excesses do not necessarily move toward the deficits. We found that the prices tend to stay outdated. This helps to explain the behavior of some of the algorithms.

In the context of the maximum flow problem, price updates can be performed by a breadth-first search computation, and therefore are very efficient. In fact, their use is crucial to a good implementation of push-relabel maximum flow algorithms. Our experience with price updates in the context of the minimum-cost flow problem is discussed in the next section.

3.3 Relabelings Dominate Pushes

In our experiments, the number of pushes was about twice the number of relabelings. However, a relabeling requires more work than a push, so the relabeling work dominates that of pushing. This makes it unlikely that the use of dynamic trees, as described in [7], will improve the practical performance of the method, especially since the constant factors associated with the dynamic trees are quite high.

4 Operation Orderings

4.1 Discharge Operation

The lower-level ordering is given by the discharge operation. This operation processes an active node by pushing flow out of it or, if no admissible arc is available for pushing, by relabeling the node. One variation of this operation stops when the node becomes inactive; another variation stops either when the node becomes inactive or is relabeled. We found that the former version performs better in most cases.

4.2 First-Active Implementation

The first-active (FA) implementation and the closely related *wave* implementation are good from the theoretical point of view. These implementations process active nodes in the topological order induced by the admissible graph. The FA implementation starts a new pass as soon as a node is relabeled and the wave implementation does not start a new pass until all nodes have been processed. Our experiments indicate that these selection strategies are very competitive with each other; one or another may be a little better depending on an instance. The FA implementation is easier to describe and code, so we concentrate on this implementation.

Recall that the FA implementation selects an active node with no active predecessors. The implementation described in [7] maintains the nodes in a linked list according to the topological ordering of the admissible graph. Because of the low activity level, however, this implementation spends most of its time scanning the list in search of the next active node; on large (several thousand nodes) examples, about 0.1% of the nodes are active, which means that this implementation makes thousands of useless list accesses before it succeeds in finding an active node.

In theoretical work, this problem came up in the context of an implementation that uses dynamic trees to reduce the number of push operations [7]. The proposed solution was to use finger search trees. However, this data structure is quite complicated and as a result the constant factor associated with it is probably quite big. The theoretical motivation for using this data structure is to avoid introducing a logarithmic factor into the running time bound. In the context of the implementation discussed here (which does not use dynamic trees), this theoretical objective does not apply, and of course in practice one uses the data structure that works best in any case.

One alternative is to use the balanced (or self-adjusting) search trees to maintain the nodes, with paths from the tree root to active nodes marked. The topologically first active node can be found by following the leftmost marked path from the root. However, the logarithmic and constant factors

introduced by this data structure are relatively large. We obtained a better implementation using a priority queue. We refer to this implementation as the PFA implementation throughout the rest of the paper.

The PFA implementation maintains topological numbers on nodes. These numbers are updated during relabelings. The priority queue maintains active nodes ordered by their topological numbers. Note that the PFA implementation is very much like the FIFO implementation, except that the topological numbers are maintained and a priority queue is used instead of a fifo queue. We use a k-ary heaps to implement the priority queue data structure.

There are two reasons why the priority queue implementation is better then the tree-based implementation. First, the constant factors for heaps are better then those for trees. Second, the tree contains all nodes, and the priority queue contains only active nodes; because of the low activity, the number of the active nodes is much smaller then the total number of nodes.

The corresponding implementation of the wave algorithm is slightly more complicated and needs two priority queues.

4.3 First-in First-out Implementation

The FIFO implementation is the simplest – and the fastest one so far. This implementation does not require any modifications to benefit from the low activity level.

The best known bound on the FIFO implementation of refine is $O(n^4)$ [4, 7]. A better bound was conjectured in [4]. As far as we know, this conjecture may still be true, since no $\omega(n^3)$ lower bound is known. Our experiments tend to support the conjecture.

4.4 Maximum Excess Implementation

The maximum excess selection rule was proposed in [5]. The maximum excess algorithm implementation is similar to the PFA implementation but uses node excesses as keys and does not maintain the topological numbers. The major difference is that if more flow is pushed into an already active node, its key must be increased. Therefore, unlike the PFA and FIFO implementations, the maximum excess implementation requires operations on nodes already in the active queue.

4.5 Comparison and Discussion

The PFA implementation runs much faster the the FA implemented that uses a linked list. This is due to the low activity level. The linked list implementation spends most of its time scanning the list in search of an active node; the priority queue implementation is designed to make the search for the next active node more efficient, and it succeeds.

The FIFO implementation is currently the fastest one we have. Although the theoretical bound for the PFA implementation is better than that of the FIFO implementation, the practical performance of the PFA implementation is worse. The number of push and relabel operations is about the same for the two implementations (see Section 7). The reason for the running time difference is clear: the priority queue operations are much more expensive than the fifo queue operations. Maintaining the

topological numbering of nodes also adds to the running time of the PFA implementation, but this cost is small.

Preliminary experiments suggest that the number of push and relabel operations is slightly lower for the maximum excess implementation as compared to the FIFO implementation. However, the running time is higher. This due to the fact that the maximum excess implementation requires the increase-key operation on the priority queue, since a node excess can increase if more flow is pushed into this node. Thus a push may require a relatively expensive priority queue operation, which is not the case for the FIFO and PFA implementations. The need for the increase-key operation also slows down other priority queue operations. We will report on experiments with the maximum excess implementation at the workshop.

An interesting observation is that, on the same problem instance, the number of push and relabel operations does not change too much among these implementations, although the selection rules used are quite different. This is probably due to outdated prices, which make it hard to make a "better" selection of the next active node to process. Thus the simplest and most efficient strategy – FIFO – wins.

Although our FIFO implementation is currently the fastest, the PFA implementation performance is also quite reasonable. It is possible that a heuristic improves the PFA implementation significantly more than the FIFO implementation. Both implementations deserve further study.

5 Heuristics

Pricing-out Pricing out is a simple technique that, to our knowledge, was never used in combinatorial minimum-cost flow algorithms (although similar, but more ed-hoc, heuristic is used in some simplex codes). The heuristic involves "removing" some arcs from the graph. We found this technique very effective – it improves the algorithm performance by a factor of 1.5–2, while takes just a few lines to implement. The theoretical justification of this technique is as follows: if the current circulation is ϵ-optimal and the absolute value of an arc cost exceeds $2n\epsilon$, the push-relabel method will not change the flow on this arc. Pricing out is done after every execution of refine. With a careful implementation, its cost is negligible compared to the cost of refine.

Price refinement The following heuristic was suggested in [7]. This heuristic is based on the fact that refine may produce a solution which is not only ϵ optimal, but also $\epsilon/$*scaling-factor* optimal. This can be tested by adding $\epsilon/$*scaling-factor* to reduced costs and running the Bellman-Ford shortest path algorithm on the resulting network. If no negative cycles are detected, f is $\epsilon/$*scaling-factor* optimal with respect to the resulting price function; in this case we say that the heuristic succeeds. We call this heuristic *price refinement.*

If a Bellman-Ford computation is much more efficient than a refine computation and the success rate is high, this heuristic may result in significant improvement in performance. This is the case for the FA implementation that uses a linked list. For the more efficient implementations, this heuristic does not seem to make much difference; sometimes it speeds the code up a little, sometimes it slows the code down. See Section 7 for more details.

We intend to reevaluate the above conclusion after refine and Bellman-Ford implementations are

tightened. Since the usefulness of this heuristic depends on the relative speeds of the two routines, the conclusion may change.

Price refinement can be also done using a minimum-mean cycle computation [8, 9], which may turn out to be superior to our current way of doing the refinement.

One reason why price refinement is attractive in practice is as follows. It seems that a Bellman-Ford computation is always faster then a refine computation, so even if the success ratio is small, the running time cannot increase too much because of the price refinement.

Finally, note that the running time of refine is higher for bigger scaling factors, but the running time of a Bellman-Ford computation does not depend on the scaling factor. Therefore price refinement is cheaper for bigger scaling factors. The success ratio, however, may decrease as the scaling factor grows.

Price updates Price updates are performed during refine when enough work has been done to amortize the cost of a price update. Thus price updates are guaranteed not to increase execution time by more than a constant factor. A price update is performed by adding ϵ to reduced costs, computing shortest path distances to the nodes with deficit in the residual graph, and adding the distances to the prices. After a price update, the prices are current, but they tend to become outdated quickly.

Price updates are supposed to reduce the number of push and relabel operations, but this happens only if the updates are frequent enough. (See Section 7.) In our current implementation, price updates are done using the Bellman-Ford algorithm, which makes the updates prohibitively expensive. We plan to experiment with using Dijkstra's algorithm to do the updates.

Maintaining Current Prices One can maintain current prices by modifying a relabel operation as described in [5] in the context of the maximum flow problem. We plan to implement this version of the algorithm. Although this version is unlikely to be practical, it may give some insight into the behavior of the method. In particular, it is likely that maintaining current prices reduces the number of push operations significantly.

Scaling Factor Selection The performance of the cost scaling method depends on the choice of the scaling factor. As the scaling factor grows, the number of scaling iterations decreases but the refinement time increases. It is important to choose a scaling factor that minimizes the running time. Although there seem to be no clear winner here, scaling factors of 4, 5, 6, 7, and 8 give best results. The scaling factors of 2 and 10 or greater are bad. See Section 7 for experimental results.

6 Example Generator

Our example generator produces relatively hard instances of the minimum-cost circulation problem. Currently our program does not work with more general transportation problems. We plan to modify the program to accept these problems and experiment with Netgen and other examples before the DIMACS workshop.

grid example	n	m	scaling factor	no heuristics time	pricing out time	... and price refinement time (success ratio)
30×30	900	9,000	2	21	13	13 (.50)
			3	17	11	13 (.29)
			4	16	11	12 (.30)
			5	17	11	11 (.30)
			6	17	10	10 (.25)
			7	19	11	10 (.38)
			8	19	12	11 (.38)
			9	23	14	13 (.38)
180×5	900	9,000	2	29	19	19 (.45)
			3	25	15	15 (.36)
			4	23	14	14 (.30)
			5	25	15	15 (.30)
			6	23	14	14 (.25)
			7	25	16	15 (.38)
			8	26	14	14 (.38)
			9	32	17	15 (.38)
100×100	10,000	200,000	6	914	508	527 (.30)
1000×10	10,000	200,000	6	1,661	944	970 (.30)

Figure 1: Pricing out and price refinement heuristics applied to the FIFO algorithm.

Our generator is based on the maximum flow generator of [4]. Imagine a torus with a mesh drawn on it. The distance from a node of the mesh to the nearest neighbor in both "horizontal" and "vertical" directions is one, and the torus dimensions are X and Y. We construct a graph on the nodes of the mesh. In this graph, every node has out-degree $\delta_X + \delta_Y$. To construct the graph, we connect each node v by a directed arc to all nodes within δ_X along the x-dimension of the mesh, and within δ_Y along the y-dimension. The capacity of an arc (v, w) depends on the distance d between v and w in the mesh. The capacity is selected from a uniform distribution on the interval $[0, 1.2^d]$. The cost of (v, w) is selected uniformly at random from the interval [−MAXCOST...MAXCOST].

We experimented with two kinds of examples, "square" and long and narrow, which tend to give somewhat different kinds of problems.

7 Experimental Results

In this section we present some experimental results obtained on SUN SPARC-1 workstation and interpret these results. The results are preliminary. For example, our current code is written to simplify modification and debugging. The optimized version we intend to develop for the DIMACS workshop will be 2-4 times faster due to the low level optimization along. We also expect that further study of heuristic and data structures will allow us to get additional high level improvements. When our code is stable, we intend to do more complete experiments following the DIMACS guidelines.

In this section, all running times are in CPU seconds, excluding the input and output times. Note

grid example	n	m	scaling factor	FIFO time	FIFO pushes	FIFO relabelings	PFA time	PFA pushes	PFA relabelings
30 × 30	900	9,000	2	13	406,208	241,918	17	418,090	241,406
			3	11	353,729	196,931	15	386,328	211,663
			4	11	312,794	188,881	13	312,050	185,290
			5	11	350,159	198,283	14	377,182	212,173
			6	10	339,634	206,916	15	357,370	216,121
			7	11	390,543	226,340	15	394,145	230,353
			8	12	375,009	230,700	15	410,026	250,143
			9	14	477,358	275,097	20	596,981	345,701
180 × 5	900	9,000	2	19	610,633	357,731	22	570,129	327,828
			3	15	534,576	302,571	19	473,778	261,325
			4	14	469,107	281,760	18	439,539	259,546
			5	15	521,069	295,607	19	500,290	285,739
			6	14	469,683	283,537	18	466,610	281,333
			7	16	527,481	307,552	19	524,144	304,754
			8	14	543,783	327,661	21	569,948	343,028
			9	17	672,136	389,310	21	537,600	316,813
100 × 100	10,000	200,000	6	508	8,279,438	5,095,941	619	9,031,503	5,550,991
1000 × 10	10,000	200,000	6	944	16,103,893	9,784,584	1,000	14,314,893	8,728,827

Figure 2: FIFO vs. PFA implementations. Both implementations use the pricing out heuristic.

also that our internal representation of the graph is symmetric and thus contains twice the number of arcs m indicated in the tables.

The table in Figure 1 gives running times for several variations of the FIFO algorithm and several values of the scaling factor. First we give times for the algorithm with no heuristics added. Next we add the pricing out heuristic. Running times are reduced, in some cases by nearly a factor of two.

Next we add the price refinement heuristic while keeping the pricing out one. (Note that pricing out speeds up both refine and Bellman-Ford computations.) The running times do not seem to be affected much.

As far as the scaling factors go, there are no clear winners, but there are clear losers. In particular, 2 is a looser. Large scaling factors also seem to be bad. Scaling factors from 4 to 8 seem to do best.

The table in Figure 2 compares the FIFO and the PFA implementations. Only the pricing out heuristic is used in both implementations. The table gives the running times as well as the count of push and relabel operations. This count is very closely correlated with the running time. Recall, however, that the PFA implementation is more expensive because of the priority queue operations involved. (The number of discharge operations is usually almost as large as the number of push operations, and each discharge involves a queue operation; pushes may cause queue operations as well.) Also note the correlation between the number of pushes and relabelings; the latter is usually slightly less then half of the former.

The number of push and relabel operations for the two algorithms is close. The FIFO algorithm uses a slightly smaller number of operations on square meshes (which tend to be easier); the PFA

grid example	n	m	BFS factor	FIFO time	FIFO pushes	FIFO relabelings	PFA time	PFA pushes	PFA relabelings
30 × 30	900	9,000	0.05	12	373,482	203,302	16	356,075	193,878
			0.1	11	292,328	155,028	17	392,047	209,683
			0.2	13	304,323	157,012	18	339,017	180,021
			0.5	14	256,100	126,810	23	372,583	191,612
			1.0	15	197,816	93,444	24	307,707	156,981
			1.5	15	169,468	78,267	33	320,053	162,814
			2.0	16	146,317	65,190	31	247,729	124,741
			2.5	18	143,618	63,310	33	228,948	114,947
180 × 5	900	9,000	0.05	16	470,397	252,237	26	641,541	353,096
			0.1	19	564,042	294,675	27	675,190	365,025
			0.2	19	445,891	228,765	32	705,363	372,539
			0.5	18	347,113	171,145	29	497,594	257,677
			1.0	19	275,710	131,063	34	427,756	216,223
			1.5	22	242,878	112,343	37	382,439	191,918
			2.0	21	195,393	88,120	48	401,268	200,569
			2.5	25	207,857	92,819	49	346,570	172,003
100 × 100	10,000	200,000	0.05	532	8,232,855	4,426,040	697	10,411,190	5,715,802
			0.1	616	8,537,182	4,479,448	936	13,067,271	6,999,114
			0.5	640	7,071,649	3,523,917	1,016	10,407,340	5,409,219
1000 × 10	10,000	200,000	0.05	1,110	24,098,347	12,480,779	1,824	23,454,282	12,888,609
			0.1	1,084	20,394,470	10,389,212	2,002	23,296,675	12,461,325
			0.5	1,437	14,440,673	7,244,118	1,947	19,394,078	10,054,807

Figure 3: Price updates in FIFO and PFA algorithms. Both algorithms use the pricing out heuristic and scaling factor 6.

algorithm has a slightly smaller number of operations on long and thin meshes (which are harder). In all cases, the number of operations is quite close, and the PFA algorithm looses 5% to 50%, mostly because of the extra overhead involved.

Figure 3 illustrates the effect of the price update heuristic on the FIFO and PFA algorithms. The update frequency is adjusted by the *update factor* parameter; the update frequency is proportional to the parameter value. Although the number of push and relabel operations tends to decrease as the update factor increases, this decrease is not monotone and seems unpredictable. On the positive side, frequent updates decrease the number of operations, especially the number of relabelings. On the negative side, for our current implementation, the cost of performing the updates exceeds the savings due to the reduced number of pushes and relabelings.

Comparing figures 2 and 3, we get an unexpected result: the number of operations may *increase* due to price updates, if the price updates are infrequent. In the context of the maximum flow problem, this does not seem happen. We feel that understanding this phenomena is important and may lead to a more effective way to update prices.

8 Conclusion

We made significant progress in the study of experimental performance of the push-relabel method, and have identified several promising directions to continue the research.

Although not optimized, our current FIFO implementation is fast. It solves hard examples with a thousand nodes and ten thousand arcs in under a minute. Hard examples with ten thousand nodes and two hundred thousand arcs are solved in under twenty minutes. This size is close to the maximum that can be executed on our machine (with 24M of memory) and not cause paging. (We are about to take delivery of a SPARC-2 with 64M of memory, however.) We would like to emphasize that our examples seem to be hard. For the maximum flow problem, our grid examples require time that is by almost an order of magnitude greater then the Netgen examples of comparable size; the same is likely to happen for the transportation problem.

As we have mentioned, we expect to get significantly faster code using both low and high level improvements. We expect that our final implementation will solve any problem that fits in 64M of memory in minutes, and smaller or easier problems in seconds.

From Jinn.Stanford.EDU!ango Tue Aug 20 00:28:48 PDT 1991 Received: by alice; Tue Aug 20 03:25:29 EDT 1991 Received: by inet.att.com; Tue Aug 20 03:25 EDT 1991 Received: by Jinn.Stanford.EDU (4.0/25-eef) id AA18161; Tue, 20 Aug 91 00:28:48 PDT Date: Tue, 20 Aug 91 00:28:48 PDT From: Andrew Goldberg ¡ango@jinn.stanford.edu¿ Message-Id: ¡9108200728.AA18161@Jinn.Stanford.EDU¿ To: dsj@research.att.com Subject: abstract.bbl Status: RO

References

[1] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding Minimum-Cost Flows by Double Scaling. Technical Report STAN-CS-88-1227, Department of Computer Science, Stanford University, 1988.

[2] R. G. Bland and D. L. Jensen. On the Computational Behavior of a Polynomial-Time Network Flow Algorithm. Technical Report 661, School of Operations Research and Industrial Engineering, Cornell University, 1985.

[3] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[4] 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).

[5] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988. A preliminary version appeared in *Proc. 18th ACM Symp. on Theory of Comp.*, 136-146, 1986.

[6] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Canceling Negative Cycles. *J. Assoc. Comput. Mach.*, 36, 1989. A preliminary version appeared in *Proc. 20th ACM Symp. on Theory of Comp.*, 388-397, 1988.

[7] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. of Oper. Res.*, 15:430–466, 1990. A preliminary version appeared in *Proc. 19th ACM Symp. on Theory of Comp.*, 7-18, 1987.

[8] R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Math.*, 23:309–311, 1978.

[9] R. M. Karp and J. B. Orlin. Parametric Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Math.*, 3:37–45, 1981.

[10] A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dok.*, 15:434–437, 1974.