FAST NEW POLYNOMIAL ALGORITHMS FOR THE MAXIMUM FLOW PROBLEM

Cenk Çalışkan, Woodbury School of Business, Utah Valley University, 800 W. University Pkwy, Orem, UT 84058, (801) 863-6487, cenk.caliskan@uvu.edu

ABSTRACT

We propose new fast polynomial algorithms for the maximum flow problem. Our generic approach is to start with a pseudoflow and convert it to a flow, at which point the maximum flow is established. We propose two approaches to do this: (1) the (feasible) flow is obtained by sending flow on paths, (2)the flow is obtained by sending flow on arcs. The first approach differs from augmenting path algorithms in the sense that feasibility is obtained at the termination of the algorithm, but they are also similar because they incrementally push flows on paths. In this research, we present the two approaches to solving the maximum flow problem, as well as some preliminary results of computational experiments with the new algorithms.

Keywords: Graph theory, Network flows, Algorithms, Complexity, Maximum flow.

INTRODUCTION

The classical maximum flow problem was first introduced by Fulkerson and Dantzig [11]. Its first exact solution algorithm was proposed by Ford and Fulkerson [8]. Let $G(N, A)$ be a directed network where N is the set of nodes; A, the set of arcs; and $s, t \in N$, the source and the sink nodes, respectively. For each arc $(i, j) \in A$, u_{ij} represents the capacity of (i, j). We denote the largest arc capacity U; the number of nodes, n; the number of arcs, m. The problem is to send the maximum possible flow from s to t.

Ford and Fulkerson's algorithm runs in $O(nmU)$ time, which is pseudopolynomial. We can categorize the existing maximum flow algorithms as follows: (1) augmenting path (2) primal and dual network simplex (3) preflow-push (4) pseudoflow (5) MA ordering. Ahuja et al. [4] is an excellent reference text on the subject. Among augmenting path algorithms, the earliest one is by Ford and Fulkerson [8]. Edmonds and Karp [7] developed two polynomial algorithms in this category. The first one augments flows with maximum residual capacity and runs in $O(m^2 \log U)$ time, and the second one augments flows on shortest paths and runs in $O(nm^2)$ time. Independently, Dinic [6] introduced the concept of shortest path networks called *layered networks* and obtained an $O(n^2m)$ time algorithm. Gabow [12] developed the first capacity scaling algorithm that is based on bit scaling. Ahuja and Orlin [2] developed a capacity scaling algorithm similar to Gabow's and runs in $O(m^2 \log U)$ time. Ahuja and Orlin [2] also developed a shortest augmenting path algorithm that uses the concept of distance labels that is due to Goldberg [13] and Goldberg and Tarjan [14, 15]. Ahuja and Orlin [2] combined their capacity scaling and shortest augmenting path algorithms to obtain $O(nm \log U)$ time.

Karzanov [18] developed the first preflow-push algorithm that runs in $O(n^3)$ time and uses the concept of layered networks. Shiloach and Vishkin [20] developed another $O(n^3)$ time preflow-push algorithm that is a precursor of the FIFO preflow-push algorithm. Goldberg [13] first used the distance labels in Shiloach and Vishkin's algorithm and obtained an $O(n^3)$ time algorithm. Goldberg and Tarjan [14, 15] improved the running time of this algorithm to $O(nm \log(n^2/m))$ by using a data structure called the *dynamic tree* developed by Sleator and Tarjan [21]. Ahuja and Orlin [1] developed the excess scaling algorithm that runs in $O(nm + n^2 \log U)$ time. Ahuja et al. [3] further improved this running time to $O(nm \log(n\sqrt{\log U}/m + 2))$. Orlin [19] recently developed an algorithm that runs in $O(nm)$ time or better for the maximum flow problem.

Fujishige [9] and Fujishige and Isotani [10] developed a maximum flow algorithm called MA ordering algorithm. MA ordering algorithm is different from all other approaches to the maximum flow problem. In this algorithm, flows are pushed neither on augmenting paths, nor on individual arcs. The running time of the algorithm is $O(nm \log U)$.

Hochbaum [16, 17] and Chandran and Hochbaum [5] developed the first algorithm that uses the concept of pseudoflows. Our algorithms are conceptually similar, but simpler and do not require any special data structures.

ASSUMPTIONS AND DEFINITIONS

We assume that all arc capacities are integers, the source node s has no incoming arcs, the sink node t has no outgoing arcs, and all other nodes $i \in N$ have at least one incoming and one outgoing arc. These assumptions are not restrictive. Fractional arc capacities can be multiplied by a suitably large number. Flow into s or out of t will not increase the flow from s to t, so those arcs can be removed from the network. Nodes with only incoming or outgoing arcs cannot be on any path from s to t; we can remove them and their adjoint arcs.

A *pseudoflow* is a function $x : A \rightarrow R^+ \cup \{0\}$ that satisfies the arc capacity constraints, i.e. $0 \leq x_{ij} \leq u_{ij}$ $\forall (i,j) \in A$. A *flow* is a pseudoflow that also satisfies the flow balance equations, i.e. $\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = 0$ $\forall i \in N \setminus \{s,t\}$. The *imbalance* of node i with respect to a pseudoflow x is defined as $e(i) = \sum_{(j,i)\in A} x_{ji} - \sum_{(i,j)\in A} x_{ij}$ $\forall i \in N \setminus \{s,t\}$. A node i is an *excess* node if $e(i) > 0$, and $e(i)$ is the *excess* of i; a *deficit* node if $e(i) < 0$, and $-e(i)$ is the *deficit* of i; and a *balanced* node if $e(i) = 0$. Arc $(i,j) \in G$ is called *saturated* if $x_{ij} = u_{ij}$.

Given a pseudoflow x, the corresponding *residual network* $G(x)$ is defined as follows: We replace each arc $(i,j) \in G$ with two arcs: (i,j) and its reversal (j,i). The residual capacity of (i,j) is defined as $r_{ij} = u_{ij} - x_{ij}$, and the residual capacity of (j,i), as $r_{ji} = x_{ij}$. We only include in $G(x)$ arcs with positive residual capacity. We refer to $G(x,\Delta)$ the Δ-residual network with respect to x and define it as a subset of $G(x)$ that consists of arcs $(i,j) \in G(x)$ for which $r_{ij} \geq \Delta$. A path from s to t in $G(x)$ or $G(x,\Delta)$ is called an *augmenting path*. A path from an excess node to a deficit node in $G(x)$ or $G(x,\Delta)$ is called a *balancing path*. A pseudoflow x satisfies the optimality conditions if there is no path from s to t in $G(x)$;

and the pseudoflow x is called a maximum pseudoflow. A maximum pseudoflow x is called: s-blocked if there is no path from s to any deficit node in $G(x)$, t-blocked if there is no path from any excess node to t in $G(x)$, st-blocked if it is both s-blocked and t-blocked, and blocked if it is st-blocked and there is no directed path between excess and deficit nodes in $G(x)$. Figure 1 illustrates these concepts of blocked flows with a simple example network. The numbers on the arcs in the original network are the capacities.

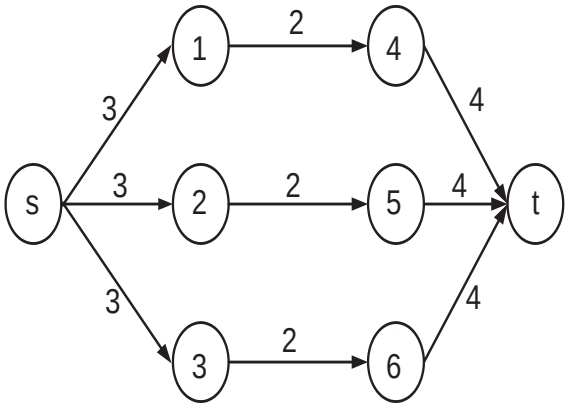
PATH-BASED ALGORITHMS

In this section, we propose algorithms that turn a pseudoflow to an optimal flow by sending flows on what we call "balancing paths." These algorithms start with a pseudoflow instead of a flow. We call the algorithms *balancing path* because the algorithms start with an st-blocked pseudoflow and convert this pseudoflow to a flow by iteratively balancing the imbalanced nodes. The algorithms maintain the optimality conditions throughout, so the resulting flow is optimal. The generic algorithm has two stages: (1) Balancing stage and (2) Flow cancelling stage. In balancing stage, the algorithm repeatedly picks an excess node and tries to send its excess to deficit nodes through balancing paths in the residual network. At the end of this stage the resulting pseudoflow is blocked. This means (i) there is no path between excess and deficits or (ii) no excess nodes are left in the network or (iii) no deficit nodes are left or (iv) the network is balanced. When all nodes are balanced the algorithm terminates. Otherwise it proceeds to the flow cancellation stage. In flow cancelling stage, the algorithm sends flows from excess nodes to node s to eliminate excesses, and flows from node t to deficit nodes to eliminate deficits. At the end of this stage all nodes are balanced and the flow is maximal.

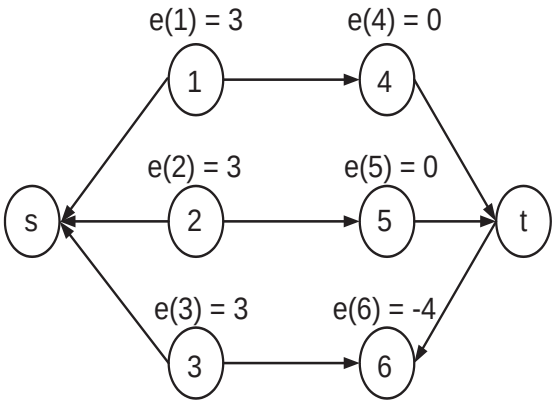
The algorithm requires an st-blocked pseudoflow to start. There are two easily constructed st-blocked pseudoflows: (i) saturate all arcs in G, (ii) Saturate only arcs incident to s and t. Either way, we will have created an st-blocked pseudoflow. By assumption, there are no paths from t to s so when we saturate all arcs, there cannot remain a path from s to t. Again, by assumption, s cannot have any path out of it, whereas t cannot have any path leading into it.

Capacity Scaling: With capacity scaling, the algorithm prioritizes arcs that have "sufficiently large" capacities and nodes that have "sufficiently large" imbalances. Both balancing and flow cancelling stages are carried out in $\log U + 1$ capacity scaling phases. Both stages start with a Δ-residual network where $\Delta = 2^{\lfloor \log U \rfloor}$. Δ is then halved at every capacity scaling phase. In a Δ-scaling phase, only $e(i) \geq \Delta$ and $e(i) \leq -\Delta$ are considered as excess and deficits. It can be shown that the algorithm has the same complexity as its augmenting path counterpart, i.e. it runs in $O(m^2 \log U)$

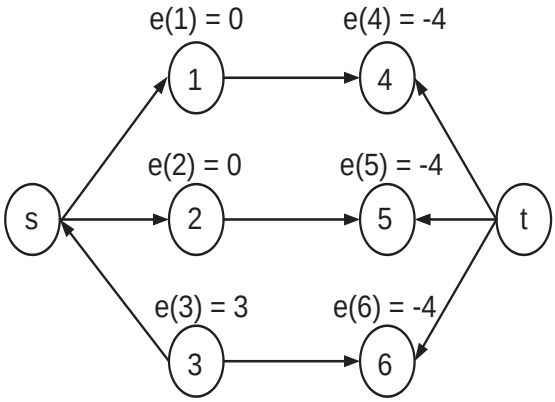
Shortest Path: A shortest balancing path is defined as the shortest path from an excess node to a deficit node in the *balancing stage*,and as the shortest path from an excess node to s in the source part, and from t to a deficit node in the sink part of the *flow cancelling stage*. We use a modified version of the distance labels introduced by Goldberg [13] and Goldberg



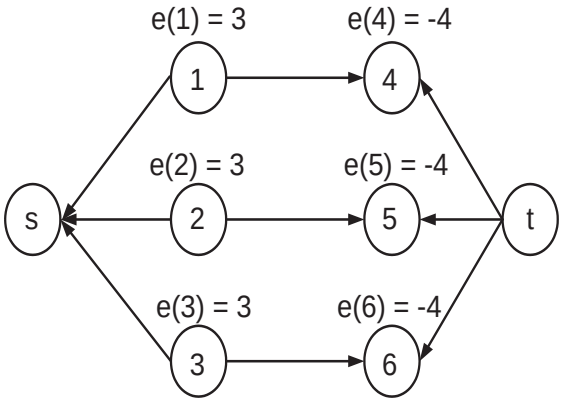
An example (original network)



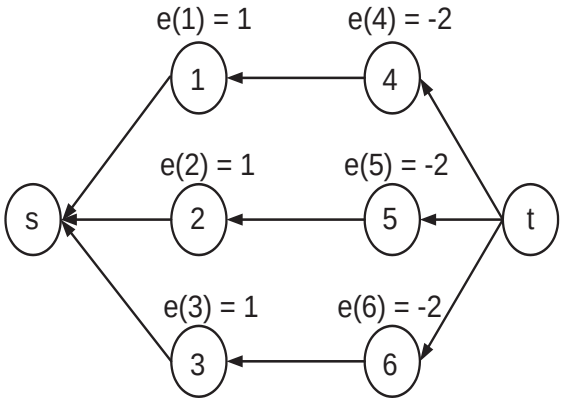
An s-blocked maximum pseudoflow
(residual network)



A t-blocked maximum pseudoflow
(residual network)



An st-blocked maximum pseudoflow
(residual network)



A blocked maximum pseudoflow
(residual network)

Figure 1: An example network and s-blocked, t-blocked, st-blocked, and blocked pseudoflows

and Tarjan [14, 15]. Let $d : N \rightarrow Z^+ \cup \{0\}$ be a *distance function* with respect to the residual network $G(x)$. Then, d is called *valid* if it satisfies: (i) $d(i) = 0 \quad \forall i \in \{i|e(i) < 0\}$ and (ii) $d(i) \leq d(j) + 1, \quad \forall (i,j) \in G(x)$. We then call $d(i)$ the *distance label* of node i. $d(i)$ is a lower bound on the length of the shortest path from i to a deficit node in $G(x)$. If $d(s) \geq n$, then there is no balancing path in $G(x)$. If $d(i)$ for all $i \in N$ is equal to the shortest path length from i to a deficit node in $G(x)$, then d is called *exact*. An arc $(i,j) \in G(x)$ is called *admissible* if $d(i) = d(j) + 1$. A path in $G(x)$ consisting entirely of admissible arcs is called an *admissible path*. If d is valid, an admissible path from s to a deficit node is a shortest balancing path. The algorithm has the same complexity as the shortest augmenting path algorithm: $O(n^2m)$. The running time will be $O(nm \log U)$ if it is combined with capacity scaling.

ARC-BASED ALGORITHMS

In this section, we propose an algorithmic framework that uses arc-based flow pushes. The algorithm starts with a pseudoflow that is obtained by saturating all source-adjoint arcs. The algorithm repeatedly sends flow from an excess node to a neighbor that is closer to a deficit node. The algorithm consists of two stages: balancing and flow cancelling. Distance labels are defined as in the previous section. This approach has at least as good a time bound as the preflow-push algorithm of Goldberg [13]. Depending on how excess nodes are selected and how pushes are performed, the time complexity could be improved beyond that of the generic algorithm.

It can be shown that any given pseudoflow can be used as the initial solution for all presented algorithms. If the network has some known special structure and the maximum flow can be approximated by a pseudoflow that can be constructed quickly, that pseudoflow can be used as the initial solution and this could save significant CPU time in practice. The algorithms that we propose can also be used to solve for reoptimizing a network if some of the arcs are removed or if the capacities of some arcs are decreased.

EXPERIMENTAL RESULTS

We used the path-based capacity scaling algorithm in our experiments. We coded the algorithm in C++ and ran the experiments on a Core i5 processor with 4.0 GB of memory. We generated the random networks by using a program that we developed. The program takes as parameters the number of nodes (n), the number of networks to be generated, the lower and upper bounds on arc capacities, density $d = m/n(n-1)$, and the seed for the random number generator. The seed is automatically incremented by one before creating the next random network. The program generates random networks as follows: (1) randomly assigns s and t from $\{1, \ldots, n\}$ (2) randomly generates a path from s to t, and the capacities for the generated arcs from a uniform distribution between u_{LB} and u_{UB} $(unif(u_{LB}, u_{UB}))$(3) randomly connects each node that is not on the path to the ones on it by an arc from a path node to the new node, and by an arc from the new node to another path node (path nodes are randomly picked and the new arc capacities are generated from $unif(u_{LB}, u_{UB})$ (4) if $m/n(n-1) < d$ at this point, the desired density is achieved by randomly generating

additional arcs with capacities from $unif(u_{LB}, u_{UB})$. We generated 10 random instances for each combination of parameters and display the average values of the results in the table below.

n	d	Sat. None	Sat. S	Sat. T	Sat. S+T	Sat. All	Random Flows
50	0.2	0.01	0.01	0.02	0.02	0.05	0.03
50	0.5	0.08	0.09	0.04	0.03	0.10	0.05
50	0.8	0.17	0.18	0.07	0.06	0.16	0.08
100	0.2	0.11	0.10	0.06	0.07	0.34	0.15
100	0.5	0.36	0.37	0.18	0.21	0.79	0.26
100	0.8	0.88	0.93	0.35	0.33	1.23	0.41
500	0.2	4.41	7.05	2.35	2.54	43.12	5.33
500	0.5	22.31	34.07	12.01	11.05	142.11	15.08
500	0.8	61.22	75.12	25.01	19.15	217.236	24.98

Figure 2: The CPU times for the balancing path capacity scaling algorithm with different initialization options and various size networks

We tested various initialization options: (1) do not saturate any arcs which is essentially the same as the capacity scaling algorithm of Ahuja and Orlin [1]; (2) saturate source-adjoint arcs (S); (3) saturate sink-adjoint arcs (T); (4) saturate both source and and sink adjoint arcs; (5) saturate all arcs; and (6) randomly assign flows to all arcs. As can be seen in Table 2, saturating sink-adjoint arcs and saturating both source and sink-adjoint arcs resulted in the fastest times. They both outperformed the regular capacity scaling algorithm by a significant margin. An interesting result was that assigning random flows resulted in almost as fast times as saturating T and saturating S & T. Saturating all arcs resulted in the worst computational time. This was because the capacities were distributed in a wide range and the pseudoflow obtained by saturating all arcs was heavily unbalanced, making it far from optimal and difficult to balance.

CONCLUSIONS AND FUTURE RESEARCH

In this research, we proposed new fast polynomial algorithms for the maximum flow problem. The theoretical complexity of the proposed algorithms are at least as good as the previous algorithms that are path and arc based. Preliminary computational experiments showed that the proposed algorithms can result in faster times in practice. Further computational testing is needed to test the performance of the other algorithms proposed in this paper. Another future research direction is to find better computational bounds by using different strategies to pick nodes and push flows, and using special data structures. Other future research directions include developing variations of the algorithmic framework proposed in this paper for the minimum cost network flow problem and its special cases.

REFERENCES

[1] R.K. Ahuja and J.B. Orlin. A fast and simple algorithm for the maximum flow problem. Operations Research, 37:748–759, 1989.

[2] R.K. Ahuja and J.B. Orlin. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. Naval Research Logistics, 38(3):413–430, 1991.

[3] R.K. Ahuja, J.B. Orlin, and R.E. Tarjan. Improved time bounds for the maximum flow problem. SIAM Journal on Computing, 18:939–954, 1989.

[4] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Network flows: Theory, algorithms and applications. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[5] B.G. Chandran and D.S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum-flow problem. Operations Research, 57(2): 358–376, 2009.

[6] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. Soviet Mathematics Doklady, 11:1277–1280, 1970.

[7] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM, 19:248–264, 1972.

[8] L.R. Ford and D.R. Fulkerson. Maximum flow through a network. Canadian Journal of Mathematics, 8:399–404, 1956.

[9] S. Fujishige. A maximum flow algorithms using MA ordering. Operations Research Letters, 31(3):176–178, 2003.

[10] S Fujishige and S. Isotani. New maximum flow algorithms by MA orderings and scaling. Journal of the Operations Research Society of Japan, 46(3):243–250, 2003.

[11] D.R. Fulkerson and G.B. Dantzig. Computation of maximum flow in networks. Naval Research Logistics Quarterly, 2:277–283, 1955.

[12] H.N. Gabow. Scaling algorithms for network problems. Journal of Computer and System Sciences, 31:148–168, 1985.

[13] A.V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, MA, 1985.

[14] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. In Proc. 18th ACM Symp. Theory Comput., pages 136–146, 1986.

[15] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. Journal of the ACM, 35(4):921–940, 1988.

[16] D.S. Hochbaum. The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem. Lecture Notes in Computer Science, 1412:325–337, 1998.

[17] D.S. Hochbaum. The pseudoflow algorithm: a new algorithm for the maximum-flow problem. Operations Research, 56(4):992–1009, 2008.

[18] A.V. Karzanov. Determining the maximal flow in a network by the method of preflows. Soviet Mathematics Doklady, 15:434–437, 1974.

[19] James B. Orlin. Max flows in o(nm) time, or better. In Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing, STOC '13, pages 765–774, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2029-0. doi: 10.1145/2488608.2488705. URL *http://doi.acm.org/10.1145/2488608.2488705*.

[20] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. Journal of Algorithms, 3:128–146, 1982.

[21] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. Journal of Computer and System Sciences, 24:362–391, 1983.