Maximum Flow with a Faster Way of Computing a Blocking Flow[*]

Yijie Han
School of Computing and Engineering
University of Missouri at Kansas City
Kansas City, MO 64110, USA
hanyij@umkc.edu

Abstract

We present a maximum flow algorithm with time complexity $O((\min(n^{2/3}, m^{1/2})m \log m \log U)$ for a network of n vertices, m arcs and integral arc capacities in the range $\{1, 2, ..., U\}$. The same algorithm is also shown to run in $O(\min(m^{1/2}(m + nm^{1/4} \log m)$, $n^{2/3}(m + n^{2/3}m^{1/2} \log m)) \log U)$ time. Thus our algorithm runs in $O(\min(m^{1/2}, n^{2/3})m \log U)$ time for $n^{4/3} \log^2 m \leq m \leq n^2$ which is faster than the best previous result of $O(\min(n^{2/3}, m^{1/2})m \log(n^2/m) \log U)$. For other range of m our algorithm runs as good as the best previous result (in $O((\min(n^{2/3}, m^{1/2})m \log m \log U)$ time). Our algorithm can be further improved if a faster Union-Split-Find algorithm becomes available.

Keywords: Algorithms, maximum flow, Union-Split-Find.

1 Introduction

The maximum flow problem is a classical problem with wide applications. It has been studied by many researchers. Depending on the arc capacity being unity, integral or real usually three versions of the maximum flow problem are studied. In this paper we study the integral version of the problem. We assume that a flow network is given as input with n vertices, m arcs and arc capacities in $\{1, 2, ..., U\}$.

Maximum flow problem has a long history with an early version traces back to 1951[3]. Goldberg and Rao [4] have a table listing the published results on the maximum flow problem. The current best result when $\log U = O(\log n)$ is published by Goldberg and Rao[4] with time complexity of $O(\min(n^{2/3}, m^{1/2})m \log(n^2/m) \log U)$. Results published before Goldberg and Rao's paper have time complexity essentially at least $O(mn)$ (for dense graphs the result of Cheriyan et al. [2] is an exception with time complexity $O(n^3/ \log n)$).

In this paper we show a maximum flow algorithm with time complexity $O((\min(n^{2/3}, m^{1/2})$ $m \log m \log U)$. The same algorithm is also shown to run in $O(\min(m^{1/2}(m + nm^{1/4} \log m)$, $n^{2/3}(m + n^{2/3}m^{1/2} \log m)) \log U)$ time. This complexity improves the time complexity of Goldberg and Rao's result[4] for $n^{4/3} \log^2 m \leq m < n^2$. In this range of m our algorithm runs in $O(\min(m^{1/2}, n^{2/3})m \log U)$ time. For other range of m our algorithm runs in the same time as Goldberg and Rao's algorithm (in $O(\min(n^{2/3}, m^{1/2})m \log m \log U)$ time). The factor $\log m$ comes from the complexity for a graph based version of the Union-Split-Find algorithm. The time complexity of our algorithm can be improved further if a faster algorithm for the Union-Split-Find becomes available.

Our algorithm is built on the framework of Goldberg and Rao[4]. However, instead of using Goldberg and Tarjan's blocking flow algorithm[5] in the design we construct our own blocking flow algorithm. By applying this blocking flow algorithm we are able to control the complexity of the whole algorithm for the maximum flow.

The rest of the paper is organized as follows. In section 2 preliminaries are given. In section 3 we introduce the Reach problem for graphs and demonstrate a solution to this Reach problem with an Union-Split-Find algorithm. This Union-Split-Find algorithm will be used in later constructions. In section 4 we briefly review Goldberg and Rao's framework[4]. In section 5 our algorithm is presented. Section 6 concludes the whole paper.

2 Preliminaries

We use $\mathbf{N}$ to denote the set of nonnegative integers.

A flow network consists of a directed graph $G = (V, E)$ of vertex set V and arc set E, a source vertex $s \in V$, a sink vertex $t \in V$, and an integral capacity function $u : E \rightarrow \{1, 2, ..., U\}$. $u(a)$ is called the

[*]Research supported in part by NSF grant 0310245.

capacity of arc a.

A flow in a flow network is a function $f : E \rightarrow \{0, 1, 2, ..., U\}$ such that
(1). $f(a) \leq u(a)$ for each arc a and
(2). $\sum_{(j,k)} f(j, k) - \sum_{(i,j)} f(i, j) = 0$ for each vertex $j \in V - \{s, t\}$.

(1) is called the capacity constraint and (2) is called the conservation constraint. The flow of the network is $|f| = \sum_{(j,t)} f(j, t)$. $f(a)$ is called the flow on arc a.

The residual capacity of arc (i, j) is $u_f(i, j) = u(i, j) - f(i, j) + f(j, i)$. (i, j) is a residual arc if $u_f(i, j) > 0$. E_f denotes the set of all residual arcs of the network. The residual graph is $G_f = (V, E_f)$. A residual flow is the difference between a maximum flow f^* and the current flow f. If $f^*(a) \geq f(a)$ the residual flow on a is $f^*(a) - f(a)$, otherwise the residual flow on a^R is $f(a) - f^*(a)$.

A blocking flow on a directed flow network is a flow f where every directed $s - t$ path contains an arc with zero residual capacity. A maximum flow is a blocking flow but not vice versa.

A binary length function is $l : E \rightarrow \{0, 1\}$. A function $d_l : V \rightarrow \mathbf{N}$ is a distance function if $d_l(i)$ is the distance from i to t with respect to l.

Given a flow f, a length function l and the distance function d_l, a residual arc (i, j) is admissible if $d(i) = d(j) + l(i, j)$. The set of all admissible arcs is denoted by $A(f, l, d)$ and the induced admissible graph is denoted by $G_A = (V, A(f, l, d))$.

3 The Reach Problem and the Union-Split-Find Algorithm

Given a acyclic directed graph $G = (V, E)$ (for graphs having cycles we can contract each of its strongly connected component into a supervertex and then working on the contracted graph which is acyclic) and two sets of vertices A and B, where $A \subseteq V$, $B \subseteq V$, $A \cap B = \phi$, we intend to perform online operations for a sequence of inputs:

$v_1, v_2, v_3, ..., v_p$

where each $v_i \in A \cup B$. If $v_i \in A$ we want to get an answer $v_j \in B$ where v_j is not yet deleted from B and V and v_j can be reached by a path in G from v_i. If $v_i \in B$ then we delete v_i from B and V. We call this problem the Reach problem.

Although the Reach problem can be solved by computing the transitive closure between vertices in A and the vertices in B, such computation is too costly. We use an Union-Split-Find algorithm on an ordered set to solve this problem.

When we are given $v \in A$, we use depth-first search to find a vertex $u \in B$. Suppose the path on the depth-first search is $v = v_1, v_2, ..., v_j = u$, then we Union all $v_1, v_2, ..., v_j$ into one set R. Within R, the vertices are ordered. Here we order the vertices as $v_1 < v_2 < v_3 < ... < v_j$. We make this set pointing to v_j. We also build a tree T. The T has root v_j and v_i's parent is v_{i+1}, $1 \leq i < j$. If next time again v is given as input, then we do not need to trace the path in tree T, we simply use a Find operation to find the node the unioned set R points to. Suppose now we are given $v' \in A$ as input. If while during depth-first search we meet at vertex v_g, i.e. the depth-first search path is $v' = v'_1, v'_2, ..., v'_p = v_g$, then we augment the tree T with this path. We view vertices as ordered as $v_{g-1} < v'_1 < v'_2 < ... < v'_{p-1} < v_g$. We Union $v'_1, v'_2, ..., v'_{p-1}$ into one set R_1. We then perform a Split operation on R to get $R_2 = \{v_1, v_2, ..., v_{g-1}\}$ and $R_3 = \{v_g, ..., v_j\}$. We then do Union to get $R_4 = R_2 \cup R_1$ and then Union again to get $R_5 = R_4 \cup R_3$. Next time if v' is input again we execute a Find operation to return u.

When we are given $u \in B$, we simply delete u and arcs incident to u. This deletion operation may require us to do set Split operations as in the tree T several vertices may have u as the parent. Next time when we are given v and we use a Find operation to find the root r of the tree T containing v, r may be not in B. Thus we have to start depth-first search at r. During depth-first search we may backtrack. If we backtrack to r and then to r's children we may need to do more set Split operations.

Thus it can be seen that for each input vertex, we have to perform $O(m + pt)$ steps where m is the total number of edges visited, p is the number of Union-Split-Find operations and t is the time for one Union-Split-Find operations. We can use a B-tree of degree 4 to represent each set. Then it is not difficult to see that each Union and each Find operation takes $O(\log m)$ time. Each Split operation can also be implemented in $O(\log m)$ time by splitting a B-tree into two B-trees and then adjusting the nodes in the B-trees bottom-up. Although there are known algorithms[1][6][7][8] for Union-Split-Find using $O(\log \log n)$ time, these algorithms do not seem to work in our case. We have:

Theorem 1: For an input of q vertices in the Reach problem we have to take $O((m + q) \log m)$ time.

Proof: If we performed h Find operations for one input vertex (for each set encountered we need only one Find operation), then we must have executed $h-1$ Union operations to union $h - 1$ sets together. Each Union operation can be associated with an arc a of the graph. If a later Split operation is performed on a (because backtrack has determined to delete the vertex a is pointing to) a is permanently deleted. Thus we can perform at most m Union and m Split operations. Since each Union-Split-Find operation takes $O(\log m)$ time, this Theorem holds. □

We are in search for faster algorithms for the Union-Split-Find operations in our case. If an Union-Split-Find algorithm with time t for each operation is found then it will be seen that we can derive a maximal flow algorithm with time $O(\min(n^{2/3}, m^{1/2})mt \log U)$.

4 Goldberg and Rao's Framework

Golgberg and Rao's framework[4] is explained in this section.

An upper bound F on the residual flow is maintained. Initially $F = nU$. F is updated every time it is reduced by a factor of 2. When $F < 1$ the maximum flow is obtained since arc capacities and flows are integral. The computation for F to be reduced by a factor of 2 is called a phase. Each phase has several update steps. Within each update step either a flow of value Δ is found or a blocking flow of value $< \Delta$ is found, where Δ depends on F. The binary length function for the phase is defined to be:

$$l(a) = \begin{cases} 0 & \text{if } u_f(a) \geq 3\Delta \\ 1 & \text{otherwise.} \end{cases}$$

Length of certain arcs are modified in order to prove that the s to t distance increases after an augmentation of a blocking flow. Arc (i, j) is special if $2\Delta \leq u_f(i, j) < 3\Delta$, $d_l(i) = d_l(j)$, and $u_f(j, i) \geq 3\Delta$. A new length function $\bar{l}$ is defined to be equal to zero for special arcs and equal to l for other arcs. Distance is not changed, i.e. $d_l = d_{\bar{l}}$.

At the beginning of each update step, d_l and $\bar{l}$ is computed. The admissible graph can have cycles of zero-length arcs. Strongly connected components of the graph induced by zero-length arcs are contracted into a single node. It is shown [4] that any flow of value $\leq \Delta$ in the contracted graph corresponds to a flow of the same value in the original graph and this conversion can be done in $O(m)$ time.

Let $A(f, l, d_l)$ be the admissible graph with flow f, length function l and distance labeling d_l. Suppose after an augmentation of f by a blocking flow in $A(f, l, d_l)$ and let f' and l' be the flow and the length function immediately after. Then the following theorem is proved in [4].

Theorem 2 (Theorem 4.3 in [4]): Suppose $f' - f$ is a blocking flow in $A(f, l, d_l)$. Then $d_l(s) < d_{l'}(s)$.

Goldberg and Rao then used the blocking flow algorithm in [5] which takes $O(m \log(n^2/m))$ time in their maximum flow algorithm. The factor $m \log(n^2/m)$ in the time complexity $O(\min(n^{2/3}, m^{1/2})m \log(n^2/m) \log U)$ represents the time complexity of this blocking flow algorithm.

The distance $d_l(s)$ in their algorithm is bounded by $O(\min(n^{2/3}, m^{1/2}))$. The distance bound of $n^{2/3}$ $(m^{1/2})$ corresponds to the time complexity of $O(n^{2/3}m \log(n^2/m) \log U)$
$(O(m^{1/2}m \log(n^2/) \log U))$ in their algorithm.

5 Our Algorithm

We design our own blocking flow algorithm. In order to apply our blocking flow algorithm we have to modify Goldberg-Rao algorithm[4].

We first give the following lemma:

Lemma 1: If the capacity of each arc is increased by d or if we introduce $O(m)$ new arcs with each arc of capacity no more than d, then the maximum flow value can be increased by at most dm.

Let $h = \min\{k | 2^k \geq F/(8m)\}$. When the upper bound on residual flow is F we consider only the set C of arcs of capacity $\geq 2^h$ since arcs of lower capacity can contribute at most $2^h m \leq F/4$ to the total flow. That is, we delete arcs of capacity $< 2^h$. For each arc a let $x(a) = u_f(a) \bmod 2^h$. We decrease $u_f(a)$ by $x(a)$. Thus if later we add deleted arcs and increase the arc capacity of arcs in C to their original value we can increase the flow by no more than $2^h m \leq F/4$. Let the admissible graph obtained be $A_1(f, l, d_l)$.

Now the capacity of the arcs in $A_1(f, l, d_l)$ are multiples of 2^h. They range from 2^h to $8m2^h$. We intend to find a flow of value $\Delta = \min(8m^{1/2}2^h, 8m2^h/n^{2/3})$ or a blocking flow of value no more than Δ.

Since $A_1(f, l, d_l)$ is an acyclic graph (because strongly connected components of zero-length arcs have been contracted), we may partition arcs to sets $U_i = \{(a, b) | d_l(a) = d_l(b) = i\}$ and $V_i = \{(a, b) | d_l(a) = d_l(b) + 1 = i\}$.

When we are searching for an augmenting path we start from s and use depth-first search. We go through several arcs in U_i and an arc in V_i. Since the flow is no more than Δ we can use our algorithm for the Reach problem when we go through arcs in U_i. Here $A = \{b|$ there is an arc (a, b) with $l(a, b) = 1$ and $d_l(b) = i\}$ and $B = \{a|$ there is an arc (a, b) with $l(a, b) = 1$ and $d_l(a) = i\}$.

Because we can use depth-first search to find an augmenting path, when we are backtracking the vertex we back from is deleted. If we reach t we can augmenting the flow by at least 2^h. Thus we need to reach t no more than $\Delta/2^h = \min(8m^{1/2}, 8m/n^{2/3})$ times in order to find a Δ flow. We use our Union-Split-Find algorithm for the Reach problem. In each U_i we attribute the first Find operation to the search operation, the search on an unexplored arc to the arc itself (this gives at most $O(m)$ time), the subsequent Union-Split-Find operations to the arc on which the Union-Split-Find is executed (this gives at most $O(m \log m)$ time). Attributed to each search operation is the time to visit arcs in V_i (constant time for each search and arc) and then the first Find operation in U_i. This gives $O(\min(m^{1/2}, n^{2/3}) \log m)$ operations per search because $d_l(s)$ is bounded by $O(\min(m^{1/2}, n^{2/3}))$. Because we have to search $\min(8m^{1/2}, 8m/n^{2/3})$ times the time attributed to search operations in $O(m \log m)$.

Thus we have showed that a flow of value Δ or a blocking flow can be found in $O(m \log m)$ time. This accomplishes one update. As shown in [4] after $O(\min(m^{1/2}, n^{2/3}))$ updates F is reduced to $F/4$. Now we increase the capacity of arcs to their original value and add the deleted arcs back. Because this adds at most $F/4$ to the maximum flow we have reduced F to $F/2$. This allows our algorithm to proceed as in [4].

Thus we have

Theorem 3: A maximum flow can be found in time $O(\min(m^{1/2}, n^{2/3}) m \log m \log U)$ time.

Note that the $\log m$ factor in the time complexity comes from the Union-Split-Find operation. Although there are $O(\log \log n)$ time algorithms for Union-Split-Find[1][6][7][8], we found that these algorithms do not apply to our case. However, this does not preclude the future discovery of faster algorithms for Union-Split-Find for our case. Thus although this algorithm does not perform better than previous best result, it provides an approach which is of hope of further improvement.

We give another analysis showing that our algorithm runs in $O(\min(m^{1/2}(m+nm^{1/4}\log m), n^{2/3}(m+n^{2/3}m^{1/2}\log m))\log U)$ time. Thus it is better than the best previous result [4] when $n^{4/3}\log^2 m \leq m < n^2$. For $m < n^{4/3}\log^2 m$ because of Theorem 3 our algorithm runs in the same time as the algorithm in [4].

We note that we use Union-Split-Find to go through vertices in U_i. However, when a tree T_1 is to be combined into tree T_2 which results in the Union operation, the root of T_2 becomes an ancestor of the vertices in T_1. For any two vertices v_1, v_2 in U_i this ancestor-descendant relation (where v_1 is the descendant and v_2 is the ancestor) can be built only once. If this relation is destroyed then v_2 is deleted and v_1 will never become a descendant of v_2 again. If there are n_i vertices in U_i the total number of ancestor-descendant relations is no more than n_i^2. Suppose a search of augmenting path goes through vertices in U_i by executing e Union operations (without backtracking), then this builds ce^2 ancestor-descendant relations for a constant c. Now consider backtracking. The search for augmenting path with backtracking within U_i forms a tree. Viewing that the tree branching from the root to the leaves, only the last branch forms the successful augmenting path. We call this last branch the main branch. For the other branches when we backtrack from a vertex v we delete v because it cannot reach the sink t. Thus the Union operation we executed when we are on the search path to reach v (the last Union operation when we merge into the tree rooted at v) and the Split operation we backtrack from v can be attributed to vertex v. This entails at most $O(n \log m)$ time. Plus we have to do several Split operations for the subtrees rooted at the children of v. However, each such a Split operation is executed on the main branch of an earlier search path and we attribute the Split operation with the corresponding Union operation we executed on that main branch of the earlier search path. Thus we have shown that the total operations for Union-Split-Find is $O(n \log m)$ plus the Union operations on the main branches.

We use the following technique to prevent repeated executing of the Find operations on an already built ancestor-descendant relation. For each vertex in A (see section 3) we add an ancestor link. If $v \in A$ is given as an input vertex and we find $u \in B$ as an ancestor of v, then we set the ancestor link of v to point to u. If next time v is input again we first use v's ancestor link to find u if u has not been deleted. This takes constant time which corresponds to building no ancestor-descendant relations. If u has been deleted, then we will execute $h > 1$ Find operations and $h-1$ Union operations to go through h sets to find a vertex in B. Thus we expend $O(h \log m)$ time which corresponds to building at least $h(h-1)/2$ ancestor-descendant relations.

Because each augmenting path increases the flow by at least 2^h. We have at most $g \leq \Delta/2^h$ main branches in each U_i. Suppose in U_i for the j-th main branch we execute e_{ij} Union operations, then the total ancestor-descendant relations we build for all main branches are $c\sum_{j=1}^{g} e_{ij}^2$ for a constant c. Since there are n_i vertices in U_i we have $c\sum_{j=1}^{g} e_{ij}^2 \leq n_i^2$. Sum for $d_l(s)$ U_i's we have:

$$c\sum_{j=1}^{g} e_{ij}^2 \leq n_i^2, i = 1, 2, ..., d_l(s).$$

$$\sum_{i=1}^{d_l(s)} n_i = n$$

Combine the above two formulae, we have:

$$\sum_{i=1}^{d_l(s)} \sqrt{c \sum_{j=1}^{g} e_{ij}^2} \leq n$$

We are seeking the maximum value of

$$M = (\log m) \sum_{i=1}^{d_l(s)} \sum_{j=1}^{g} e_{ij}$$

.

We enlarge the above formulae and write:

Find the maximum value of

$$M = (\log m) \sum_{i=1}^{d_l(s)} \sum_{j=1}^{g} e_{ij}$$

Subject to constraint

$$\sum_{i=1}^{d_l(s)} \sqrt{c \sum_{j=1}^{g} e_{ij}^2} = n$$

The maximum is attained when all e_{ij}'s are equal. Thus we have

$$d_l(s)\sqrt{cg}e_{ij} = n \text{ or } e_{ij} = n/(d_l(s)\sqrt{cg})$$

and therefore $M = (\log m)d_l(s)gn/(d_l(s)\sqrt{cg}) = n \log m\sqrt{g/c} \leq n \log m\sqrt{\Delta/(c2^h)}$
$= O(\log m \min(nm^{1/4},\ n^{2/3}m^{1/2}))$

Plus the $O(n \log m)$ time attributed to the vertices and $O(m)$ time needed simply by visiting all arcs we have time $O(m + \log m \min(nm^{1/4}, n^{2/3}m^{1/2}))$ for the blocking flow algorithm.

Since the $O(m + nm^{1/4} \log m)$ time for a Δ flow or a blocking flow corresponds to $d_l(s) = O(m^{1/2})$, according to [4] we have a maximum flow algorithm running in $O(m^{1/2}(m + nm^{1/4} \log m) \log U)$ time.

Time $O(m + n^{2/3}m^{1/2} \log m)$ for a Δ flow or a blocking flow corresponds to $d_l(s) = O(n^{2/3})$. By [4] we have a maximum flow algorithm running in $O(n^{2/3}(m + n^{2/3}m^{1/2} \log m) \log U)$ time.

Combined with Theorem 3 we have

Theorem 4: A maximum flow problem can be computed in $O(\min(m^{1/2}(m + nm^{1/4} \log m),\ n^{2/3}(m + n^{2/3}m^{1/2} \log m),\ m^{1/2}m \log m,\ n^{2/3}m \log m) \log U)$ time.

Corollary: When $m \geq n^{4/3} \log^2 m$ the maximum flow can be computed in $O(\min(m^{1/2},\ n^{2/3})m \log U)$ time.

6 Conclusion

Although removing the dependency on U in the time complexity will be very exciting, currently we think that $O(\min(m^{1/2}, n^{2/3})m \log U)$ is probably the "right" bound. We are only able to reach it for $m \geq n^{4/3} \log^2 m$. It would also be interesting to find a faster Union-Split-Find algorithm for the Reach problem.

References

[1] P.v. Emde Boas, R. Kaas, E. Zijlstra. Design and implementation of anefficient priority queue. Math. Systems Theory, 10(1977). pp. 99-127.

[2] J. Cheriyan, T. Hagerup, K. Mehlhorn. $O(n^3)$-time maximum-flow algorithm. *SIAM J. Comput. 45*, 1144-1170(1996).

[3] G. B. Dantzig. Application of the simplex method to a transportation problem. In *Activity Analysis and Production and Allocation.* Wiley, New York, 359-373(1951).

[4] A. V. Goldberg, S. Rao. Beyond the flow decomposition barrier. *J. ACM*, Vol. 45, No. 5,,, 783-797(September 1998).

[5] A. V. Goldberg, R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res. 15*, 430-466(1990).

[6] R. G. Karlsson. Algorithms in a restricted universe. Report CS-84-50, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1984.

[7] K. Mehlhorn. Datenstrukturen und Algorithmen 1, Teubner, 1986.

[8] S. Näher. Dynamic fractional Cascading oder die Verwaltung vieler linearer Listen. Dissertation, University des Saarlandes, Saarbrüchen, 1987.