DOI:10.1145/2628036

Though maximum flow algorithms have a long history, revolutionary progress is still being made.

BY ANDREW V. GOLDBERG AND ROBERT E. TARJAN

Efficient Maximum Flow Algorithms

THE MAXIMUM FLOW problem and its dual, the minimum cut problem, are classical combinatorial optimization problems with many applications in science and engineering; see, for example, Ahuja et al.[1] The problem is a special case of linear programming and can be solved using general linear programming techniques or their specializations (such as the network simplex method[9]). However, special-purpose algorithms are more efficient. Moreover, algorithm design techniques and data structures developed to compute maximum flows are useful for other problems as well. Although a special case of linear programming, the maximum flow problem is general enough so several important problems (such as the maximum bipartite matching problem) reduce to it.

Here, we survey basic techniques behind efficient maximum flow algorithms, starting with the history and basic ideas behind the fundamental maximum flow algorithms, then explore the algorithms in more detail. We restrict ourselves to basic maximum flow algorithms and do not cover interesting special cases (such as undirected graphs, planar graphs, and bipartite matchings) or generalizations (such as minimum-cost and multi-commodity flow problems).

Before formally defining the maximum flow and the minimum cut problems, we give a simple example of each problem: For the maximum flow example, suppose we have a graph that represents an oil pipeline network from an oil well to an oil depot. Each arc has a capacity, or maximum number of liters per second that can flow through the corresponding pipe. The goal is to find the maximum number of liters per second (maximum flow) that can be shipped from well to depot. For the minimum cut problem, we want to find the set of pipes of the smallest total capacity such that removing the pipes disconnects the oil well from the oil depot (minimum cut).

The maximum flow, minimum cut theorem says the maximum flow value is equal to the minimum cut capacity. This fundamental theorem has many applications, particularly in the design of maximum flow algorithms.

We distinguish between flow algorithms that are polynomial or strongly polynomial. We denote the number of vertices and arcs in the input network by n and m, respectively. For polynomial algorithms, the arc capacities are integral, with U denoting the largest capacity; capacities can be represented by $O(\log U)$-bit integers. (In practice, it is reasonable to assume capacities are integral; as implemented by computer hardware, even floating point

» key insights

- The idea of augmenting along shortest paths leads to polynomial-time algorithms.
- Data structures and fine-grain operations lead to faster algorithms.
- Discriminating based on residual capacities when assigning arc lengths leads to improved time bounds.

ILLUSTRATION BY JURGEN ZIEWE

numbers are represented as integers.) A polynomial algorithm is one with a worst-case time bound polynomial in n, m, and log U. For many applications, algorithms manipulate numbers that fit in a machine word, and elementary arithmetic operations take unit time. We assume this is the case when stating polynomial bounds. A strongly polynomial algorithm is one with a worst-case time bound polynomial in n and m, even if capacities are arbitrary real numbers, assuming a computational model in which elementary arithmetic operations on real numbers take unit time. Strongly polynomial algorithms are more natural from a combinatorial point of view, as only their arithmetic operation complexity depends on the input number size, and other operation counts are independent of the size.

The first special-purpose algorithm for the maximum flow problem was the augmenting path method developed by Ford and Fulkerson.[14] This method is, in general, not polynomial time but can be made so. One way to do this is through scaling, as introduced by Dinic.[11]

Edmonds and Karp[12] introduced the shortest augmenting path method, making the Ford-Fulkerson method strongly polynomial. To define path lengths, the Edmonds-Karp method uses the unit length function, which sets the length of each arc to one. Edmonds and Karp note that other length functions can be used but do not seem to lead to better time bounds. The key to the analysis is the observation that the shortest augmenting path length is non-decreasing and must eventually increase.

The blocking flow method augments along a maximal set of shortest paths by finding a blocking flow. Such augmentation increases the shortest augmenting path length and thereby speeds up the shortest augmenting path method. Blocking flows are implicit in Dinic's algorithm[10] and made explicit by Karzanov,[23] who also introduced a relaxation of flow called a "preflow" that allows an algorithm to change the flow on a single arc instead of on an entire augmenting path. Arc flow is updated through a push operation. Preflows allow faster algorithms for finding blocking flows.

The maximum flow, minimum cut theorem says the maximum flow value is equal to the minimum cut capacity.

An interesting special case of the maximum flow problem involves all arcs having unit capacities. As shown independently by Karzanov[22] and Even and Tarjan,[13] the blocking flow algorithm in this case achieves better time bounds than in the general case for two reasons: the number of blocking flow computations is reduced, and the computations are faster—linear time in the graph size.

The operations of a blocking flow algorithm can be divided into two parts: those that manipulate distances and those that manipulate flows. In theory, the latter dominate, motivating development of data structures that allow changing flow values on a path more efficiently than one arc at a time. The first such data structure was developed by Galil and Naamad.[15] A few years later, Sleator and Tarjan[29,30] introduced the dynamic tree data structure, allowing changing flow values on a path with k arcs in $O(\log k)$ time. This led to improvement in the theoretical time bound for finding a blocking flow, making it almost linear.

Goldberg and Tarjan[18] developed the push-relabel method as an alternative to the blocking flow method.[a] It maintains a preflow and updates it through push operations. It introduces the relabel operation to perform fine-grain updates of the vertex distances. Push and relabel operations are local; that is, they apply to a single arc and vertex, respectively. These fine-grain operations provide additional flexibility that can be used to design faster algorithms. The fastest general-purpose maximum flow codes are based on the push-relabel method.[7,16]

For arbitrary real-valued capacities, the blocking flow problem can be solved in $O(m \log(n^2/m))$ time,[19] giving an $O(nm \log(n^2/m))$ bound for the maximum flow algorithm. Note a decomposition of flows into paths can have a size of $\Omega(nm)$. This makes $O(nm)$ a natural target bound. In 2013, Orlin[27] developed an algorithm that achieves this bound.

The flow decomposition size is not a lower bound for computing maximum flows. A flow can be represented in $O(m)$ space, and dynamic trees can be used to augment flow on a path in logarithmic time. Furthermore, the unit capacity problem on a graph with no parallel arcs can be solved in $O(\min(n^{2/3}, \sqrt{m})m)$ time,[13,22] which is much better than $O(nm)$. For a quarter century, there was a big gap between the unit capacity case and the general case. The gap was narrowed by Goldberg and Rao,[17] who obtained an $O(\min(n^{2/3}, \sqrt{m})m \log(n^2/m) \log U)$-time algorithm for the problem with integral capacities.

To achieve this bound, Goldberg and Rao used a non-unit length function. In combination with new design and analysis techniques, this leads to the binary blocking flow algorithm that achieves the bound mentioned earlier. As the name implies, the algorithm is based on blocking flows. No comparable bound for the push-relabel method is known. This fact revives the theoretical importance of the blocking flow method.

Here, we assume familiarity with basic graph algorithms, including breadth- and depth-first search and have organized the article as follows: After introducing basic definitions, we discuss the algorithms. Our presentation is informal, including intuitive algorithm descriptions and the corresponding time bounds, but omits technical details, which can be found in the references.

Background

The input to the maximum flow problem is (G, s, t, u), where $G = (V, A)$ is a directed graph with vertex set V and arc set A, $s \in V$ is the source, $t \in V$ is the sink (with $s \neq t$), and $u : A \Rightarrow \mathbf{R}^+$ is the strictly positive capacity function. We sometimes assume capacities are integers and denote the largest capacity by U.

A flow f is a function on A that satisfies capacity constraints on all arcs and conservation constraints at all vertices except s and t. The capacity constraint for $a \in A$ is $0 \leq f(a) \leq u(a)$ (flow does not exceed capacity). The conservation constraint for v is

[a] The push-relabel method is sometimes called the preflow-push method, which is misleading, as Karzanov's algorithm uses preflows and the push operation but does not use the relabel operation and is therefore not a push-relabel algorithm.

$\sum_{(u,v)\in A} f(u,v) = \sum_{(v,w)\in A} f(v,w)$ (the incoming flow is equal to the outgoing flow). The flow value is the net flow into the sink: $|f| = \sum_{(v,t)\in A} f(v,t) - \sum_{(t,v)\in A} f(t,v)$. If $|f|$ is as large as possible, f is a maximum flow. A cut is a bipartition of the vertices $S \cup T = V$ with $s \in S, t \in T$.[b] The capacity of a cut is defined by $u(S,T) = \sum_{v\in S, w\in T, (v,w)\in A} u(S,T)$ (the sum of capacities of arcs from S to T). The max-flow/min-cut theorem[14] says the maximum flow value is equal to the minimum cut capacity. Figures 1 and 2 give an input network and a maximum flow on it, respectively.

Without loss of generality, we assume G is connected. Then $m \geq n - 1$ and therefore $n + m = O(m)$. We can also assume the graph has no parallel arcs, since we can combine parallel arcs and add their capacities.

Residual Graph and Augmenting Paths

An important notion for flow algorithms is a residual graph, encoding the possible changes of flow on arcs in a way that facilitates algorithm design. Suppose we have an arc $a = (v, w)$ with $u(a) = 9$ and $f(a) = 4$. We can then increase the flow on a by up to five units without violating the capacity constraint. Furthermore, we can decrease the flow on a by up to four units. We would like to interpret decreasing flow on an arc $a = (v.w)$ as increasing flow on the reverse arc $a^R = (w, v)$.

Given a flow f in G, we define the residual graph $G_f = (V, A_f)$ as follows: A_f contains arcs $a \in A$ such that $f(a) < u(a)$ and arcs $a^R : a \in A$ such that $f(a) > 0$. We call these forward and reverse residual arcs, respectively. We define the residual capacity u_f to be $u(a) - f(a)$ for the former and $f(a^R)$ for the latter. For every arc $a \in A$, G_f contains the forward arc, the reverse arc, or both. Figure 3 gives the residual graph for the flow in Figure 2. Note the residual graph can have parallel arcs even if the input graph is simple, as it can contain both an arc and its reversal.

A flow g in G_f defines the flow f' in G as follows: For a forward arc $a \in G_f$, $f'(a) = f(a) + g(a)$; for a reverse arc $a \in G_f$, $f'(a^R) = f(a^R) - g(a)$. Seeing that f' is a valid flow is straightforward.

An augmenting path is a path from s to t in G_f. Given an augmenting path P, we can augment f as follows: Let δ be the minimum residual capacity of the arcs on P and g be the flow of value on P. The corresponding flow f' on G has $|f'| = |f| + \delta > |f|$. An augmenting path can be found in $O(m)$ time (such as by using breadth- or depth-first search).

Note that during an augmentation, at least one arc of P has residual capacity δ before the augmentation and zero after the augmentation. We say such an arc is saturated by the augmentation. Saturated arcs are deleted from G_f. An arc a is added to G_f if $u_f(a)$ is zero before the augmentation, and the augmentation increases the flow on a^R.

Using the max-flow/min-cut theorem, one can show a flow f has maximum value if and only if G_f does not contain an augmenting path. This motivates the augmenting path algorithm: while G_f contains an augmenting path, find such a path and augment the flow on it.

If capacities are integral, the augmenting path algorithm always terminates, since each augmentation increases the flow value by at least one. This observation, and the fact that the capacity of the cut $(\{s\}, V - \{s\})$ is $O(nU)$, gives a pseudo-polynomial bound of $O(nmU)$ on the algorithm's running time. The bound is not polynomial because U can be exponential in the size of the problem input. If the capacities are real-valued, the algorithm need not terminate. As we shall see later, variants of this algorithm do run in polynomial time.

Scaling

Scaling is one way to make the augmenting path algorithm polynomial-time if the capacities are integral.

Recall that U is the largest arc capacity and let $k = \lceil \log_2 U \rceil + 1$, the number of bits needed to represent capacities. For $i = 0,\ldots,k$, define $u_i(a) = \lfloor u(a)/2^{k-i} \rfloor$. Note $u_0 \equiv 0$, and for $i > 0$, u_i is defined by the i most significant bits of u. The zero flow is maximum for u_0.

Given a maximum flow f_i for capacities u_i $(0 \leq i < k)$, the algorithm computes a maximum flow f_{i+1} for capacities u_{i+1} as follows. Note $u_{i+1} = 2u_i + b_{i+1}$, where $b_{i+1}(a)$ is the $(i+1)$-st most significant bit of $u(a)$. Thus $f = 2f_i$ is a feasible flow for capacities u_{i+1}.

Figure 1. Input example.

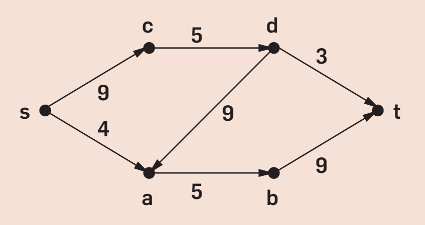


Figure 2. Maximum flow (capacity/flow) and minimum cut.

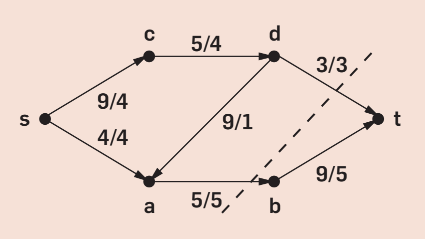
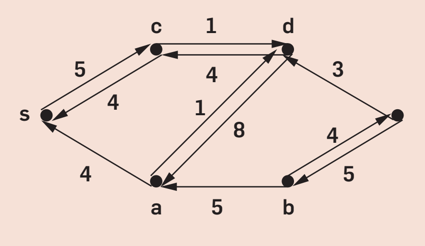


Figure 3. Residual graph and residual capacities corresponding to Figure 2.



We start with f and apply the augmenting path algorithm to compute f_{i+1}.

To bound the number of augmentations, consider a minimum cut (S, T) for capacities u_i. Since f_i is a maximum flow, for every arc a from S to T we have $u_i(a) = f_i(a)$, and thus for the initial flow f, we have $u_{i+1}(a) - f(a) \leq 1$. Therefore $|f|$ is within m of maximum, and we need at most m augmentations to compute f_{i+1} from f_i. The running time of the scaling algorithm is thus $O(m^2 \log U)$.

Shortest Augmenting Paths

Define the length of every arc in G_f to be one, and suppose we always choose a shortest augmenting path. This is natural, since breadth-first search finds shortest augmenting paths and takes linear time.

Consider a shortest-path augmentation. Let $d(v)$ denote the distance from a vertex v to t in G_f, and let $k = d(s)$. For an arc (v, w) on the augmenting path, we have $d(v) = d(w) + 1$. Therefore the reverse arc (w, v) is not on a path from s to t of length k or less. The augmentation deletes at least one arc on a path

[b] Formally, this defines an s-t cut, though, here, we deal only with s-t cuts; in the literature, a minimum cut may also refer to the minimum cut value over all s, t pairs of minimum s-t cuts.

from s to t of length k and does not add any arcs on paths from s to t of length k or less. This observation leads to the key monotonicity property of the shortest augmenting path algorithm: For every vertex v, residual graph distances from s to v and from v to t are non-decreasing.

The monotonicity property yields a strongly polynomial time bound. Each augmentation saturates an arc on a path of the current shortest length. Therefore, after at most m augmentations, the distance from s to t must increase. Initially, the distance is at least one, and if t is reachable from s, the distance is at most $n - 1$. The total number of augmentations is thus $O(nm)$. The time for one augmentation is $O(m)$ to find the augmenting path and proportional to the path length; that is, $O(n) = O(m)$ to modify the flow. This gives an $O(nm^2)$ bound on the running time.

Blocking Flow Method

Given a network G with arc capacities, a flow f in G is blocking if every s-to-t path in G contains a saturated arc. Note f need not be a maximum flow, as there can be an s-to-t path in G_f that will contain the reverse of an arc of G (see Figure 4). But a maximum flow is always a blocking flow. As we shall see later, in an acyclic graph, blocking flows can be found more quickly than maximum flows.

Figure 4. Example of a blocking flow that is not a maximum flow.

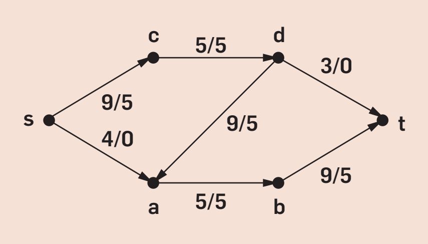


Figure 5. Push operation example: before (left) and after (right); zero excesses and non-residual arcs not shown.

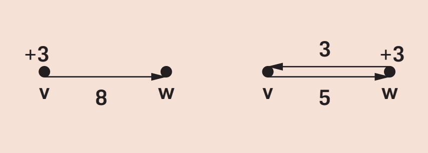
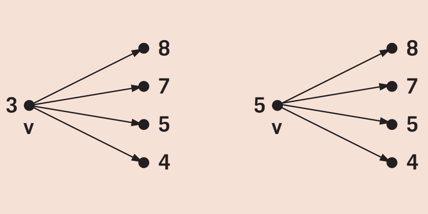


Figure 6. Relabel operation example: before (left) and after (right).



The blocking flow algorithm constructs an auxiliary network $G'_f = (V, A'_f)$ where A'_f contains all residual arcs belonging to some shortest s-to-t path. Note if $(v, w) \in A'_f$, then $d(v) = d(w) + 1$, so G'_f is acyclic. G'_f can be constructed in $O(m)$ time using breadth-first search. Suppose we compute a blocking flow g in G'_f. Then $f + g$ is a feasible flow in G. Furthermore, one can show the s-to-t distance in G_{f+g} is greater than that in G_f. It follows that a maximum flow can be computed in at most $n - 1$ iterations, where the time for an iteration is dominated by the blocking flow computation.

Dinic[10] introduced an algorithm for finding blocking flows in acyclic graphs, using depth-first search to find an augmenting path in G'_f that augments along the path and deletes saturated arcs from G'_f. The key to the analysis is the observation that if depth-first search retreats from a vertex, there is no path from the vertex to t in G'_f and the vertex can be deleted. One can use this observation to show the running time of the algorithm is proportional to n plus the total length of the augmenting paths found; the total length term dominates. As an augmenting path has $O(n)$ arcs and each augmentation saturates an arc, the running time of the blocking flow algorithm is $O(nm)$. This gives an $O(n^2m)$ bound for Dinic's maximum flow algorithm.

Using Dynamic Trees

The running time of the blocking flow algorithm is dominated by changes of arc flows that do not saturate the arc. A natural approach to improving the running time bound is to use a data structure that allows one to make several such changes in one data structure operation. This can be achieved by using a data structure to remember non-saturated portions of the augmenting paths. The dynamic tree data structure[29,30] was developed for this purpose.

Intuitively, the dynamic tree blocking flow algorithm uses the data structure to remember non-saturated portions of augmenting paths that may be reused later. In particular, the saved paths are linked during the augmenting path search, work that is amortized over the search for augmenting paths. Flow augmentation is performed using dynamic tree operations, at the cost of the logarithm of the corresponding augmenting path length. This application of dynamic trees reduces the running time of the blocking flow algorithm from $O(nm)$ to $O(m \log n)$. By restricting the maximum tree size and using additional data structures, this bound can be further improved to $O(m \log(n^2/m))$,[19] yielding an $O(nm \log(n^2/m))$ maximum flow algorithm.

Although dynamic trees yield the best worst-case bounds, they have so far not been used in practical implementations because most practical instances are relatively easy, and the constant factors in dynamic tree implementations are relatively large.

Push-Relabel Method

The blocking flow algorithm uses global operations (such as building the auxiliary network and augmenting along a path). The push-relabel method uses local operations. These fine-grain operations give the method more flexibility, which can be used to make the method faster in practice.

Following Karzanov,[23] the push-relabel method uses preflows. Preflows are like flows, but the conservation constraints are relaxed: $\sum_{(u,v)\in A} f(u, v) \geq \sum_{(v,w)\in A} f(v, w)$ for all $v \in V - \{s, t\}$ (the incoming flow is at least the outgoing flow). We define excess by $e_f(v) = \sum_{(u,v)\in A} f(u,v) - \sum_{(v,w)\in A} f(v,w)$. A vertex with excess can push some of it to its residual neighbor. Intuitively, we want to push only to a neighbor that is closer to the sink. Karzanov[23] uses distances in the auxiliary network to determine where to push flow.

The push-relabel method replaces the distances by a valid labeling, a relaxation of distances that can be updated locally. Given a flow f, we say a function $d : V \rightarrow \mathcal{N}$ is a valid labeling if $d(t) = 0$ and for every $(v, w) \in A_f$, we have $d(v) \leq d(w) + 1$. One can show a valid labeling gives lower bounds on distances to t. In particular, if $d(v) \geq n$, then there is no path from v to t in G_f, meaning v is on the source side of some minimum cut.

The push-relabel method maintains a preflow f and a valid distance labeling

d and updates them using operations push and relabel, respectively. We describe these operations next; a full description of the algorithm can be found in Goldberg and Tarjan.[18]

An interesting special case of the maximum flow problem involves all arcs having unit capacities.

One way to start the push-relabel method is to saturate all arcs out of the source by setting their flow values to the corresponding capacity values, and to set $d(s) = n$ (t is not reachable from s) and $d(v) = 0$ for $v \neq s$. This creates initial flow excesses on vertices adjacent to the source. Intuitively, the method pushes flow excesses toward the sink and relabels vertices with excess if the excess cannot be pushed toward the sink. We say a vertex v is active if $v \neq t$ and $e_f(v) > 0$.

The push operation applies to an arc (v, w) if v is active and $d(w) < d(v)$, or w is closer to t according to d. The operation determines the maximum amount of flow that can be pushed, $\delta = \min(e_f(v); u_f(v, w))$ and pushes this amount of flow along (v, w) by setting $u_f(v, w) = u_f(v, w) - \delta$, $u_f(w, v) = u_f(w, v) + \delta$, $e_f(v) = e_f(v) - \delta$, and $e_f(w) = e_f(w) + \delta$. We say a push is saturating if $\delta = u_f(v, w)$ and non-saturating otherwise. Note that a non-saturating push gets rid of all the excess of v; see Figure 5 for an example of a non-saturating push operation.

The relabel operation applies to an active vertex v such that no push operation applies to an arc (v, w), or for all $(v, w) \in A_f$, $d(v) \leq d(w)$. The operation sets $d(v) = \min\{n, 1 + \min_{(v,w)\in A_f} d(w)\}$. (A vertex with excess always has an outgoing residual arc.) Note the relabel operation always increases $d(v)$; see Figure 6 for an example of a relabel operation.

The time complexity of the push-relabel method is as follows. The total time for relabeling operations is $O(nm)$, and the time for saturating pushes is $O(nm)$ as well. The time for non-saturating pushes is $O(n^2m)$; these operations dominate the running time bound, which is also $O(n^2m)$.

Note our description of the push-relabel method is generic; we have not specified the rule to select the next active vertex to process. Some operation orderings lead to better bounds. In particular, for the highest label push-relabel algorithm, which always selects an active vertex with the highest distance label to process next, the time for non-saturating pushes and the overall time bound are $O(n^2\sqrt{m})$.[6] Using dynamic trees, one can get an $O(nm\log(n^2/m))$ bound[18] more simply than through the blocking flow method.

The highest-label algorithm is also one of the most practical variants of the push-relabel method. However, robust practical performance requires additional heuristics. The push relabel method is very flexible, making it easy for the algorithm designer to add heuristics. For example, one can restrict active vertices to those with $d(v) < n$ and do post-processing to compute the final flow. One can also do periodic backward breadth-first searches to maximize $d(v)$ values. See, for example, Cherkassky and Goldberg[7] and Goldberg.[16]

Unit Capacities

Now consider the special case of the maximum flow problem in which all input arc capacities are one. Since merging parallel arcs results in non-unit capacities, we cannot assume the graph has no parallel arcs, so we consider two cases—parallel arcs and no parallel arcs—in both of which one obtains better bounds for Dinic's algorithm.

First, note that after an augmentation, all arcs on the augmenting path are saturated. Therefore, an arc participates in at most one augmentation per blocking flow, and the blocking flow algorithm runs in $O(m)$ time.

Moreover, one can show the number of blocking flow computations is $O(\sqrt{m})$. To prove this bound, we divide the maximum flow computation into two phases. In the first phase, the s-to-t distance is less than $\sqrt{m}$. Since an augmentation by a blocking flow increases the distance, the first phase consists of at most $\sqrt{m}$ augmentations. One can show if the s-to-t distance is at least $\sqrt{m}$, the residual flow value is $O(\sqrt{m})$. Since an augmentation decreases the value, the number of augmentations in the second phase is $O(\sqrt{m})$.

This analysis implies an $O(m^{3/2})$ bound for the unit capacity problem. If G has no parallel arcs, one can also obtain an $O(n^{2/3}m)$ bound, which is better for dense graphs.

Binary Blocking Flow Algorithm

The time bounds for the blocking flow

and push-relabel algorithms are $\Omega(nm)$ in the general case, while for unit capacities, the algorithm of Dinic runs in $O(\min(n^{2/3}, \sqrt{m})m)$ time. Here, we discuss the intuition behind the binary blocking flow algorithm of Goldberg and Rao,[17] which narrows the gap for the integral capacity case.

Instead of assigning unit length to every residual arc, the binary blocking flow algorithm uses a zero-one length function, assigning zero length to the arcs with large residual capacity and unit length to the arcs with small residual capacity. The fact that arcs with unit length have small residual capacity allows the algorithm to come close to the unit capacity time bound.

The algorithm maintains a flow f and an upper bound F on the difference between the maximum flow value and the current flow value $|f|$. The algorithm proceeds in phases; each phase decreases F by a factor of two. In a phase, the value of F remains constant except for the very end of the phase, when it is decreased. A threshold parameter Δ, which is a function of F and thus remains constant during a phase, determines whether residual arcs are large or small; large arcs have a residual capacity of at least 3Δ, and the remaining ones are small.

As in the case of the unit length function, we define the auxiliary network G'_f to be the graph induced by the arcs on shortest s-to-t paths. The algorithm repeatedly computes a blocking flow in G'_f, updating G'_f before each computation, until F decreases by a factor of two. The decrease in F happens if one either increases the flow by $F/2$ or the s-to-t distance becomes sufficiently large. As in the unit capacity case, a large s-t distance implies a bound on the residual flow value.

This use of the binary length function leads to two problems that must be addressed: First, G'_f need not be acyclic (it can contain cycles of zero-length arcs), and, second, an arc length can decrease from one to zero, and, as a side effect, the s-to-t distance may fail to increase after a blocking flow augmentation.

To deal with the first problem, the algorithm contracts strongly connected components of G'_f, looks for a blocking flow in the resulting acyclic graph, and stops the blocking flow computation if the value of the flow being computed reaches Δ. One can show that since each strongly connected component is induced by large-capacity arcs, one can always route a flow of value Δ through it. At the end of each iteration, we expand G'_f and extend the flow we found to G_f. One can show that a blocking flow in G'_f extends to a blocking flow in G_f. This gives us two types of iterations: ones that find a blocking flow and ones that find a flow of value Δ. The s-to-t distance increases in the former case and does not decrease in the latter case.

The maximum flow problem is far from being completely understood, and new and improved algorithms continue to be discovered.

To deal with the second problem, one can show the arcs (v, w) that may have their length decrease (the special arcs) have the property that the residual capacity of (v, w) is at least 3Δ and the residual capacity of (w, v) at least 2Δ. The algorithm contracts such arcs, assuring that after an augmentation by a blocking flow, the s-to-t distance increases even if these arc lengths decrease.

Using the dynamic-tree data structure, the binary flow algorithm runs in $O(\min(n^{2/3}, \sqrt{m})m \log(n^2/m) \log U)$ time, which is within a $\log(n^2/m) \log U$ factor of the best known upper bound for the unit capacity problem with no parallel arcs.

Conclusion

As mentioned here, a 2013 algorithm of Orlin[27] achieves an $O(nm)$ strongly polynomial bound for the maximum flow problem, as well as an $O(n^2/\log n)$ bound for $m = O(n)$. This result is quite sophisticated and uses a combination of ideas from maximum flow, minimum-cost flow, and dynamic connectivity algorithms. In particular, Orlin uses the binary blocking flow algorithm as a subroutine. His result closes a longstanding open problem of the existence of an $O(nm)$ maximum flow algorithm. However, the binary blocking flow algorithm bounds suggest an $O(nm/n^{\epsilon})$ strongly polynomial algorithm may exist.

The maximum flow problem is far from being completely understood, and new and improved algorithms continue to be discovered. We would like to mention four intriguing directions that have yielded new results: The first is to generalize the push-relabel approach to allow the flow excess (incoming minus outgoing flow) at a vertex to be arbitrary—positive, negative, or zero. Given a residual arc (v, w) such that the excess at v exceeds the excess at w, one can balance the arc by increasing its flow to either saturate the arc or equalize the excesses at v and w. The flow balancing algorithm[31] starts with some initial flow (such as zero flow), dummy excesses of plus infinity at s and minus infinity at t, and repeats arc-balancing steps until all such steps move a sufficiently small amount of flow, then rounds the flow to obtain an exact maximum flow. Although the running time of this algorithm ($O(n^2 m \log U)$) is not competitive with that of the best algorithms, the method is simple and extends to give a very simple and practical algorithm for a parametric version of the maximum flow algorithm.[2,31]

Another approach that yields a fast practical algorithm for maximum flow problems in computer-vision applications is that of Boykov and Kolmogorov,[5] improving the basic augmenting path method by using bidirectional search to find augmenting paths, in combination with a clever method for retaining information from previous searches to speed up future ones. The Boykov-Kolmogorov method does not augment on shortest paths and has not been proved to be polynomial but can be modified to find exact shortest paths and to be polynomial without sacrificing its practical performance, indeed improving it in many cases. The resulting algorithm[20] computes shortest augmenting paths incrementally, using information from previous searches. Special techniques have yielded fast maximum flow algorithms for planar graphs and for undirected graphs; for the latest results, see Borradaile and Klein,[3] Borradaile et al.,[4] and Karger and Levine.[21]

A recent series of papers, including Christiano et al.,[8] Kelner et al.,[24] Lee et al.,[25] and Sherman,[28] have studied the problem of finding an approximately maximum flow (within a factor of $1 + \epsilon$ of maximum) in undirected graphs and culminates in a near-linear time algorithm. These papers used linear algebraic techniques and electrical flows. Building on this work, Madry[26] in 2013 obtained a breakthrough result, an exact algorithm for unit capacity flows in directed graphs running in $\tilde{O}(m^{10/7})$ time. This improves the classical $O(\min(n^{2/3}, m^{1/2})m)$ bound for the problem, suggesting that better bounds for the exact capacitated maximum flow problem in directed graphs may be possible. Whether these ideas can be used to find exact maximum flows in directed graphs with integral capacities is an intriguing open question. In summary, progress on maximum flow algorithms has been made for more than half a century, and continues.

Acknowledgment

Robert E. Tarjan was supported by National Science Foundation grant CCF-0830676 and U.S.-Israel Binational Science Foundation Grant 2006204. Some of his work was done while visiting Stanford University and supported by an Air Force Office of Scientific Research Multidisciplinary University Research Initiative grant.

References

1. Ahuja, R.K., Magnanti, T.L., and Orlin, J.B. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1993.
2. Babenko, M.A., Derryberry, J., Goldberg, A.V., Tarjan, R.E., and Zhou, Y. Experimental evaluation of parametric max-flow algorithms. In *Proceedings of the Sixth Workshop on Experimental Algorithms, Lecture Notes in Computer Science*. Springer, Heidelberg, Germany, 2007, 256–269.
3. Borradaile, G. and Klein, P.N. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *Journal of the ACM 56,* 2 (2009), 1–34.
4. Borradaile, G., Klein, P.N., Mozes, S., Nussbaum, Y., and Wulff-Nilsen, C. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science*. IEEE Press, New York, 2011,170–179.
5. Boykov, Y. and Kolmogorov, V. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26,* 9 (2004), 1124–1137.
6. Cheriyan, J. and Maheshwari, S.N. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing 18,* 6 (1989), 1057–1086.
7. Cherkassky, B.V. and Goldberg, A.V. On implementing push-relabel method for the maximum flow problem. *Algorithmica 19,* 4 (1997), 390–410.
8. Christiano, P., Kelner, J.A., Madry, A., Spielman, D.A., and Teng, S-H. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the Annual ACM Symposium on Theory of Computing*. ACM Press, New York, 2011, 273–282.
9. Dantzig, G.B. Application of the simplex method to a transportation problem. In *Activity Analysis and Production and Allocation,* T.C. Koopmans, Ed. John Wiley & Sons, Inc., New York, 1951, 359–373.
10. Dinic, E.A. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematical Docladi 11* (1970), 1277–1280.
11. Dinic, E.A. Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi [Excess scaling and transportation problems]. In *Issledovaniya po Diskretnoi*. Matematike Nauka, Moscow, Russia, 1973.
12. Edmonds, J. and Karp, R.M. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM 19,* 2 (1972), 248–264.
13. Even, S. and Tarjan, R.E. Network flow and testing graph connectivity. *SIAM Journal on Computing 4,* 4 (1975), 507–518.
14. Ford, Jr., L.R. and Fulkerson, D.R. Maximal flow through a network. *Canadian Journal of Mathematics 8* (1956), 399–404.
15. Galil, Z. and Naamad, A. An $O(EV \log^2 V)$ algorithm for the maximal flow problem. *Journal of Computer and System Sciences 21,* 2 (1980), 203–217.
16. Goldberg, A.V. Two-level push-relabel algorithm for the maximum flow problem. In *Proceedings of the Fifth Conference on Algorithmic Aspects in Information Management, Volume 5564 of Lecture Notes in Computer Science*. Springer, Heidelberg, Germany, 2009, 212–225.
17. Goldberg, A.V. and Rao, S. Beyond the flow decomposition barrier. *Journal of the ACM 45,* 5 (1998), 753–782.
18. Goldberg, A.V. and Tarjan, R.E. A new approach to the maximum flow problem. *Journal of the ACM 35,* 4 (1988), 921–940.
19. Goldberg, A.V. and Tarjan, R.E. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research 15,* 3 (1990), 430–466.
20. Goldberg, A.V., Hed, S., Kaplan, H., Tarjan, R.E., and Werneck, R.F. Maximum flows by incremental breadth-first search. In *Proceedings of the 19th European Symposium on Algorithms*. Springer-Verlag, Heidelberg, Germany, 2011, 457–468.
21. Karger, D.R. and Levine, M. Finding maximum flows in undirected graphs seems easier than bipartite matching. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, 1997.
22. Karzanov, A.V. Tochnaya otzenka algoritma nakhojdeniya maksimalnogo potoka, primenennogo k aadache 'o predstavitelyakh' [The exact time bound for a maximum flow algorithm applied to the set representatives problem]. In *Problems in Cibernetics 5* (1973), 66–70.
23. Karzanov, A.V. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematical Dokladi 15* (1974), 434–437.
24. Kelner, A., Lee, Y.T., Orecchia, L., and Sidford, A. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. SIAM, Philadelphia, 2014, 217–226.
25. Lee, T., Rao, S., and Srivastava, N. A new approach to computing maximum flows using electrical flows. In *Proceedings of the Annual ACM Symposium on Theory of Computing*. ACM Press, New York, 2013, 755–764.
26. Madry, A. Navigating central path with electrical flows: From flows to matchings, and back. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science*. IEEE Press, New York, 2013, 253–262.
27. Orlin, J.B. Max Flows in $O(nm)$ time, or better. In *Proceedings of the Annual ACM Symposium on Theory of Computing*. ACM Press, New York, 765–774.
28. Sherman, J. Nearly maximum flows in nearly linear time. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science*. IEEE Press, New York, 2013, 263–269.
29. Sleator, D.D. and Tarjan, R.E. A data structure for dynamic trees. *Journal of Computer and System Sciences 26,* 3 (1983), 362–391.
30. Sleator, D.D. and Tarjan, R.E. Self-adjusting binary search trees. *Journal of the ACM 32,* 3 (1985), 652–686.
31. Tarjan, R.E., Ward, J., Zhang, B., Zhou, Y., and Mao, J. Balancing applied to maximum network flow problems. In *Proceedings of the 14th European Symposium on Algorithms*. Springer-Verlag, Berlin, 2006, 612–623.

Andrew V. Goldberg (goldberg@microsoft.com) is a principal researcher at Microsoft Research Silicon Valley Lab, Mountain View, CA.

Robert E. Tarjan (ret@cs.princeton.edu) is the James S. McDonnell Distinguished University Professor of Computer Science at Princeton University, Princeton, NJ, and a visiting researcher at Microsoft Research Silicon Valley Lab, Mountain View, CA.