The Design and Analysis of Algorithms

Dexter C. Kozen

With 72 Illustrations

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

Dexter C. Kozen
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Series Editor:
David Gries
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Library of Congress Cataloging-in-Publication Data
Kozen, Dexter, 1951–
The design and analysis of algorithms / Dexter C. Kozen.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-97687-6
1. Computer algorithms. I. Title.
QA76.9.A43K69 1991
005.1—dc20 91-38759

Printed on acid-free paper.

Production managed by Bill Imbornoni; manufacturing supervised by Jacqui Ashri.
Photocomposed from a LaTeX file.
Printed and bound by R.R. Donnelley & Sons, Inc., Harrisonburg, VA.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-97687-6 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-97687-6 Springer-Verlag Berlin Heidelberg New York

Lecture 16 Max Flow

Suppose we are given a tuple $G = (V, c, s, t)$, where V is a set of *vertices*, $s, t \in V$ are distinguished vertices called the *source* and *sink* respectively, and c is a function $c : V^2 \to \mathcal{R}_+$ assigning a nonnegative real *capacity* to each pair of vertices. We make G into a directed graph by defining the set of directed edges

$$E \;=\; \{(u, v) \mid c(u, v) > 0\} \; .$$

Intuitively, we can think of the edges as wires or pipes along which electric current or fluid can flow; the capacity $c(e)$ represents the carrying capacity of the wire or pipe, say in amps or gallons per minute. The *max flow problem* is to determine the maximum possible flow that can be pushed from s to t, and to find a routing that achieves this maximum. The following definition is intended to capture the intuitive idea of a *flow*.

Definition 16.1 A function $f : V^2 \to R$ is called a *flow* if the following three conditions are satisfied:

(a) *skew symmetry*: for all $u, v \in V$,

$$f(u, v) \;=\; -f(v, u) \; ;$$

(b) *conservation of flow at interior vertices*: for all vertices u not in $\{s, t\}$,

$$\sum_{v \in V} f(u, v) \;=\; 0 \; ;$$

(c) *capacity constraints*: $f \leq c$ pointwise; *i.e.*, for all u, v,

$$f(u,v) \leq c(u,v) .$$

We say that (u,v) is *saturated* if $f(u,v) = c(u,v)$. □

If we think of edges (u,v) for which $f(u,v) > 0$ as carrying flow *out of* u, and edges (u,v) for which $f(u,v) < 0$ (or equivalently by (a), $f(v,u) > 0$) as carrying flow *into* u, then condition (b) says that the total flow out of any interior vertex is equal to the total flow into that vertex, or in other words, the *net flow* (total flow out minus total flow in) at any interior vertex is 0.

It follows from (a) that $f(u,u) = 0$ for any vertex u.

Figure 1 illustrates a graph with capacities c (ordinary typeface) and a flow f on that graph (italic). Edges not shown have a capacity of 0 and a flow that is the negative of the flow in the opposite direction; *e.g.*, $c(u,s) = 0$ and $f(u,s) = -4$. If neither an edge nor its opposite is shown (*e.g.* (s,t)), then the capacities and flows in both directions are 0.

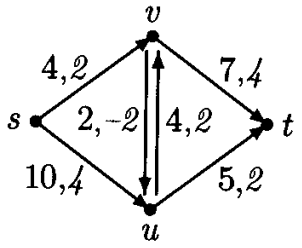


Figure 1

Definition 16.2 An *s,t-cut* (or just *cut*, when s,t are understood) is a pair A, B of disjoint subsets of V whose union is V such that $s \in A$, $t \in B$. The *capacity* of the cut A, B, denoted $c(A,B)$, is

$$c(A,B) = \sum_{u \in A,\ v \in B} c(u,v) ,$$

i.e., the total capacity of the edges from A to B. If f is a flow, we define the *flow across the cut* A, B to be

$$f(A,B) = \sum_{u \in A,\ v \in B} f(u,v) .$$

□

Note that by condition (a) of Definition 16.1, $f(A,B)$ gives the *net flow* across the cut from A to B; that is, the sum of the positive flow values on edges from A to B minus the sum of the positive flow values on edges from B to A.

Definition 16.3 The *value* of a flow f, denoted $|f|$, is defined to be

$$\begin{aligned} |f| &= f(\{s\}, V - \{s\}) \\ &= \sum_{v \in V} f(s, v) \ , \end{aligned}$$

or in other words the net flow out of s. □

In the example of Figure 1, $|f| = 6$.

Although Definition 16.3 defines the value of the flow f with respect to the cut $\{s\}, V - \{s\}$, the flow value will be the same no matter where it is measured:

Lemma 16.4 *For any s,t-cut A, B and flow f,*

$$|f| = f(A, B) \ .$$

Proof. Induction on the cardinality of A, using condition (b) of Definition 16.1. □

In particular,

$$f(\{s\}, V - \{s\}) = f(V - \{t\}, \{t\}) \ ,$$

which says that the net flow out of s equals the net flow into t.

The flow across any cut surely cannot exceed the capacity of the cut. This is expressed in the following lemma:

Lemma 16.5 *For any s,t-cut A, B and flow f,*

$$|f| \leq c(A, B) \ .$$

Proof. Lemma 16.4 and condition (c). □

The main result of this lecture will be the *Max Flow-Min Cut Theorem*, which states that the minimum cut capacity is achieved by some flow; *i.e.*, the inequality in Lemma 16.5 is an equality for some cut A, B and some flow f^*. The flow f^* necessarily has maximum value among all flows on G by Lemma 16.5, and is called a *max flow*. The flow f^* is not unique, but its value is.

16.1 Residual Capacity

Definition 16.6 Given a flow f on G with capacities c, we define the *residual capacity function* $r : V^2 \to R$ to be the pointwise difference

$$r = c - f \ .$$

The *residual graph* associated with $G = (V, E, c)$ and flow f is the graph $G_f = (V, E_f, r)$, where

$$E_f = \{(u, v) \mid r(u, v) > 0\} \ .$$

□

The residual capacity $r(u,v)$ represents the amount of additional flow that could be pushed along the edge (u,v) without violating the capacity constraint (c) of Definition 16.1. In case the flow $f(u,v)$ is negative, this "additional flow" could involve backing off the positive flow from v to u. For example, if $c(u,v) = 8$ and $f(u,v) = 6$, and $(v,u) \notin G$ so that $c(v,u) = 0$, then $r(u,v) = 2$ and $r(v,u) = c(v,u) - f(v,u) = 0 - (-6) = 6$. The residual graph for the flow in Figure 1 is given in Figure 2 below.

Note that the residual graph G_f can have an edge where there was none in G. However, G_f has no edges (u,v) where neither (u,v) nor (v,u) were present in G, so $|E_f| \leq 2 \cdot |E|$.

Intuitively, the formation of the residual graph translates the problem by making f the new origin (zero flow). Solving the residual flow problem is tantamount to solving the original flow problem; a solution to the residual flow problem can be added to f to obtain a solution to the original problem. This observation is formalized in the following lemma.

Lemma 16.7 *Let f be a flow in G, and let G_f be its residual graph.*

(a) The function f' is a flow in G_f iff $f + f'$ is a flow in G.

(b) The function f' is a max flow in G_f if $f + f'$ is a max flow in G.

(c) The value function is additive; i.e., $|f + f'| = |f| + |f'|$ and $|f - f'| = |f| - |f'|$.

(d) If f is any flow and f^* a max flow in G, then the value of a max flow in G_f is $|f^*| - |f|$.

Proof.

(a) Since f is a flow, it satisfies skew symmetry ($f(u,v) = -f(v,u)$) and conservation at interior vertices ($\sum_v f(u,v) = 0$). Thus f' satisfies these properties iff $f + f'$ does. To show that the capacity constraints are satisfied, recall that the capacities of G_f are given by $r = c - f$, where c is the capacity function of G. Then

$$\begin{aligned} f' \leq r \quad &\text{iff} \quad f' \leq c - f \\ &\text{iff} \quad f + f' \leq c \,. \end{aligned}$$

(b) This follows directly from (a).

(c) By the definition of flow value,

$$\begin{aligned} |f \pm f'| &= \sum_v (f(s,v) \pm f'(s,v)) \\ &= \sum_v f(s,v) \pm \sum_v f'(s,v) \\ &= |f| \pm |f'| \,. \end{aligned}$$

(d) This follows directly from (b) and (c).

□

16.2 Augmenting Paths

Definition 16.8 Given G and flow f on G, An *augmenting path* is a directed path from s to t in the residual graph G_f. □

An augmenting path represents a sequence of edges on which the capacity exceeds the flow, *i.e.*, on which the flow can be increased. As observed above, on some edges this "increase" may actually involve decreasing a positive flow in the opposite direction.

Figure 2 illustrates the residual graph associated with the flow in the example of Figure 1 and an augmenting path. The minimum capacity of any edge in this path is 2, so the flow can be increased on these edges by 2, resulting in a new flow in the original graph with value 2 greater than that of $|f|$. Note that the "increase" on (u, v) is essentially a decrease of a positive flow on (v, u).

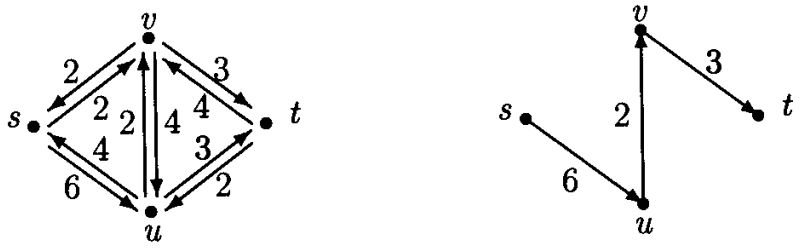


Figure 2

We are now ready to state and prove the main theorem of this lecture:

Theorem 16.9 (Max Flow-Min Cut Theorem [34]) *The following three statements are equivalent:*

(a) f is a max flow in $G = (V, E, c)$;

(b) there is an s,t-cut A, B with $c(A, B) = |f|$;

(c) there does not exist an augmenting path.

Proof.

(b) → (a) This is immediate from Lemma 16.5.

(a) → (c) Suppose there is an augmenting path $u_0, u_1, \ldots, u_n$ with $s = u_0$ and $t = u_n$. Let

$$d \;=\; \min\{r(u_i, u_{i+1}) \mid 0 \le i < n\} \;>\; 0\,.$$

The quantity d is the smallest residual capacity along the augmenting path and is called the *bottleneck capacity*. An edge along the augmenting path with that capacity is called a *bottleneck edge*. Define the following flow g in the residual graph G_f:

$$\begin{aligned} g(u_i, u_{i+1}) &= \;\;d, \;\; 0 \le i < n \\ g(u_{i+1}, u_i) &= -d, \;\; 0 \le i < n \\ g(u, v) &= \;\;0, \;\; \text{for all other pairs } (u, v). \end{aligned}$$

Then g is a flow in G_f with value d. By Lemma 16.7, $f + g$ is a flow in G and $|f + g| = |f| + |g| = |f| + d$.

(c) $\rightarrow$ (b) Assume there is no augmenting path. Let A consist of all vertices reachable from s by paths in the residual graph. Let $B = V - A$. There are no edges in the residual graph from A to B; thus in G, all edges from A to B are saturated, *i.e.* $f(u, v) = c(u, v)$. It follows from Lemma 16.4 that $c(A, B) = |f|$. □

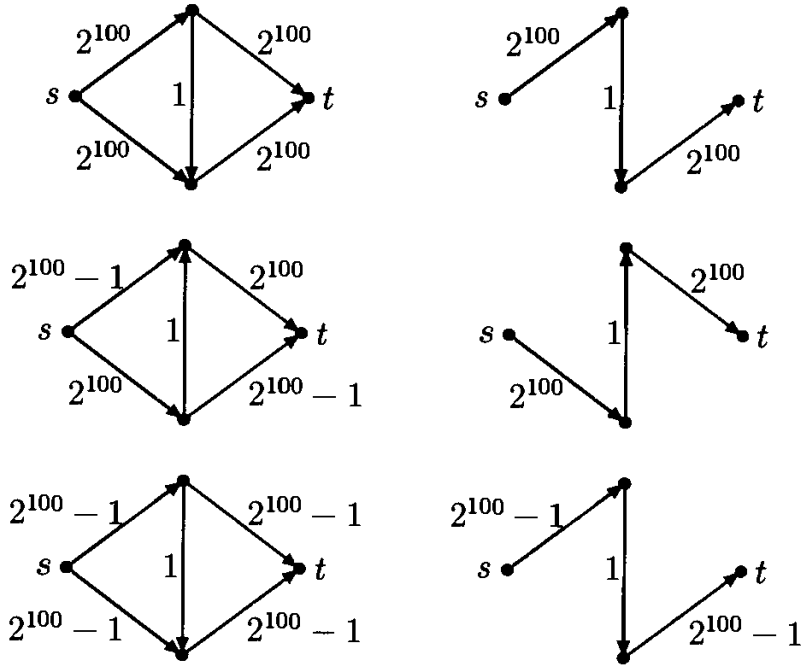
Lecture 17 More on Max Flow

The Max Flow-Min Cut Theorem gives an algorithm for finding a flow with maximum value in a given network as long as the capacities are rational numbers. This algorithm was first published in 1956 by Ford and Fulkerson [34].

The algorithm works as follows. We begin with the zero flow, then repeatedly find an augmenting path p and push d additional units of flow along p from s to t, where $d > 0$ is the bottleneck capacity of p (minimum edge capacity along p). We continue until it is no longer possible to find an augmenting path, *i.e.* until the residual graph has no path from s to t. We know at that point by the Max Flow-Min Cut Theorem that we have a max flow.

If the edge capacities are integers, this algorithm increases the flow value by at least 1 with each augmentation, hence achieves a maximum flow after at most $|f^*|$ augmentations. Moreover, each augmentation increases the flow by an integral amount, so $|f^*|$ is an integer. Unfortunately, $|f^*|$ can be exponential in the representation of the problem, and the algorithm can run for this long if the augmenting paths are not chosen with some care.

Example 17.1 The following diagram illustrates the first few augmentations in a flow problem with large capacities. The residual graphs are shown on the left-hand side and the augmenting paths on the right. This sequence of augmentations will take 2^{101} steps to converge to a max flow, which has value 2^{101}.



□

In fact, if the capacities are irrational, the process of repeated augmentation along indiscriminately chosen augmenting paths may not produce a max flow after a finite time, as the following example shows.

Example 17.2 Let r be the positive root of the quadratic $x^2 + x - 1$:

$$r = \frac{-1+\sqrt{5}}{2} \approx .618\ldots$$

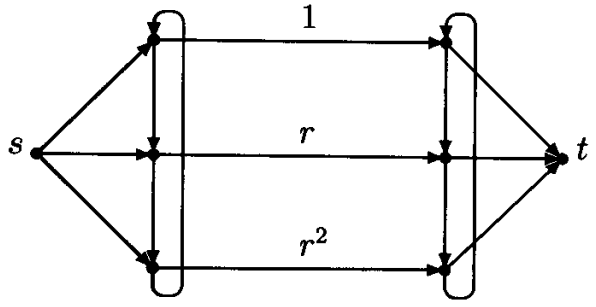
Then $r^2 = 1 - r$, and more generally, $r^{n+2} = r^n - r^{n+1}$ for any $n \geq 0$. Also, since $0 < r < 1$,

$$1 > r > r^2 > r^3 > \cdots > 0\,.$$

Note

$$r + 2 = \frac{1}{1-r} = \sum_{n=0}^{\infty} r^n\,.$$

Consider the following flow network:



The three horizontal interior edges (call them the *flumes*) have the capacities shown, and all other edges have capacity $r + 2$. The max flow value is $1 + r + r^2 = 2$, since this is the minimum cut capacity obtained by cutting the flumes; any other cut has capacity at least $r + 2 > 2$.

Suppose that in the first augmenting step, we push one unit of flow directly from s to t along the top flume. This leaves residual capacities of $0, r$, and r^2 on the flumes.

Now we perform the following loop, which after n iterations will result in the flumes having residual capacities 0, r^{n+1}, and r^{n+2} in some order: choose the flume with minimum nonzero residual capacity, say d, and push d units of flow from s forward along that flume, back through the saturated flume, and then forward through the remaining flume to t. Suppose that we start with residual capacities $0, r^n$, and r^{n+1} on the flumes. The minimum nonzero residual capacity is r^{n+1}, and the new residual capacities will be r^{n+1}, $r^n - r^{n+1} = r^{n+2}$, and 0, respectively. The situation is the same as before, only rotated.

The loop can be repeated indefinitely, leaving ever higher powers of r on the flumes. We always have sufficient residual capacity on the non-flumes. The residual capacities tend to 0, so the flow value tends to the maximum flow value 2.

With irrational capacities, the sequence of augmentations need not even converge to the maximum flow value. An example of this behavior can be obtained from the graph above by adding an edge (s, t) of weight 1. The same infinite sequence of augmentations converges to a flow of value 2, but the maximum flow value is 3. □

17.1 Edmonds and Karp's First Heuristic

Edmonds and Karp [30] suggested two heuristics to improve this situation. The first is the following:

Always augment by a path of maximum bottleneck capacity.

Definition 17.3 A *path flow* in G is a flow f that takes nonzero values only on some simple path from s to t. In other words, there exist a number d and a simple path $u_0, u_1, \ldots, u_k$ with $s = u_0$, $t = u_k$, and such that

$$\begin{aligned} f(u_i, u_{i+1}) &= d, \quad 0 \le i \le k-1 \\ f(u_{i+1}, u_i) &= -d, \quad 0 \le i \le k-1 \\ f(u, v) &= 0, \quad \text{for all other } (u, v). \end{aligned}$$

□

Lemma 17.4 *Any flow in G can be expressed as a sum of at most m path flows in G and a flow in G of value 0, where m is the number of edges of G.*

Proof. Let f be a flow in G. If $|f| = 0$, we are done. Otherwise, assume $|f| > 0$ (the argument for $|f| < 0$ is symmetric, interchanging the roles of s and t). Define a new capacity function $c'(e) = \max\{f(e), 0\}$ and let G' be the graph with these capacities. Then f is still a flow in G', and since $c' \leq c$, any flow in G' is also a flow in G. By the Max Flow-Min Cut Theorem, the null flow in G' must have an augmenting path, which is a path from s to t with positive capacities; by construction of G', every edge on this path is saturated by f. Take p to be the path flow on that path whose value is the bottleneck capacity. Then the two flows p and $f - p$ are both flows in G', and at least one edge on the path (the bottleneck edge) is saturated by p.

Now we repeat the process with $f - p$ to get $c'' \leq c'$ and G'', and so on. Note that G'' has strictly fewer edges than G', since at least the bottleneck edge of p has disappeared. This process can therefore be repeated at most m times before the flow value vanishes. The original f is then the sum of the remaining flow of value 0 and the path flows found in each step. □

We now consider the complexity of maximum-capacity augmentation.

Theorem 17.5 *If the edge capacities are integers, then the heuristic of augmentation by augmenting paths of maximum bottleneck capacity results in a maximum flow f^* in at most $O(m \log |f^*|)$ augmenting steps.*

Proof. By Lemma 17.4, f^* is a sum of at most m path flows and a flow of value 0, therefore one of the path flows must be of value $|f^*|/m$ or greater. An augmenting path of maximum bottleneck capacity must have at least this capacity. Augmenting by such a path therefore increases the flow value by at least $|f^*|/m$, so by Lemma 16.7(d) of the previous lecture, the max flow in the residual graph has value at most $|f^*| - |f^*|/m = |f^*|(\frac{m-1}{m})$. Thus after k augmenting steps, the max flow in the residual graph has value at most $|f^*|(\frac{m-1}{m})^k$. Hence the number of augmenting steps required to achieve a max flow is no more than the least number k such that

$$|f^*|(\frac{m-1}{m})^k \;<\; 1\,.$$

Using the estimate

$$\log m - \log(m-1) \;=\; \Theta(\frac{1}{m})\,, \tag{24}$$

we obtain $k = \Theta(m \log |f^*|)$. The estimate (24) follows from the limit

$$\lim_{n\to\infty}(1 - \frac{1}{n})^n \;=\; \frac{1}{e}\,.$$

□

Finding a maximum capacity augmenting path can be done efficiently using a modification of Dijkstra's algorithm (Homework 5).

17.2 Edmonds and Karp's Second Heuristic

The method described above is still less than completely satisfactory, since the complexity depends on the capacities. It would be nice to have an algorithm whose asymptotic worst-case complexity is a small polynomial in m and n alone.

The following algorithm produces a max flow in time independent of the edge capacities. This algorithm is also due to Edmonds and Karp [30]. It uses the following heuristic to achieve an $O(m^2n)$ running time:

Always choose an augmenting path of minimum length.

Definition 17.6 The *level graph* L_G of G is the directed breadth-first search graph of G with root s with sideways and back edges deleted. The *level* of a vertex u is the length of a shortest path from s to u in G. □

Note that the level graph has no edges from level i to level j for $j \geq i+2$. This says that any shortest path from s to any other vertex is a path in the level graph. Any path with either a back or sideways edge of the breadth-first search graph would be strictly longer, since it must contain at least one edge per level anyway.

Lemma 17.7 *(a) Let p be an augmenting path of minimum length in G, let G' be the residual graph obtained by augmenting along p, and let q be an augmenting path of minimum length in G'. Then $|q| \geq |p|$. Thus the length of shortest augmenting paths cannot decrease by applying the above heuristic.*

(b) We can augment along shortest paths of the same length at most $m = |E|$ times before the length of the shortest augmenting path must increase strictly.

Proof. Choose any path p from s to t in the level graph and augment along p by the bottleneck capacity. After this augmentation, at least one edge of p will be saturated (the bottleneck edge) and will disappear in the residual graph, and at most $|p|$ new edges will appear in the residual graph. All these new edges are back edges and cannot contribute to a shortest path from s to t as long as t is still reachable from s in the level graph. We continue finding paths in the level graph and augmenting by them as long as t is reachable from s. This can occur at most m times, since each time an edge in the level graph disappears. When t is no longer reachable from s in the level graph, then any augmenting path must use a back or side edge, hence must be strictly longer. □

This gives rise to the following algorithm:

Algorithm 17.8 (Edmonds and Karp [30]) Find the level graph L_G. Repeatedly augment along paths in L_G, updating residual capacities and deleting edges with zero capacity until t is no longer reachable from s. Then calculate a new level graph from the residual graph at that point and repeat. Continue as long as t is reachable from s.

With each level graph calculation, the distance from s to t increases by at least 1 by Lemma 17.7(a), so there are at most n level graph calculations. For each level graph calculation, there are at most m augmentations by Lemma 17.7(b). Thus there are at most mn augmentations in all. Each augmentation requires time $O(m)$ by DFS or BFS, or $O(m^2n)$ in all. It takes time $O(m)$ to calculate the level graphs by BFS, or $O(mn)$ time in all. Therefore the running time of the entire algorithm is $O(m^2n)$.

Lecture 18 Still More on Max Flow

18.1 Dinic's Algorithm

We follow Tarjan's presentation [100]. In the Edmonds-Karp algorithm, we continue to augment by path flows along paths in the level graph L_G until every path from s to t in L_G contains at least one saturated edge. The flow at that point is called a *blocking flow*. The following modification, which improves the running time to $O(mn^2)$, was given by Dinic in 1970 [29]. Rather than constructing a blocking flow path by path, the algorithm constructs a blocking flow all at once by finding a maximal set of minimum-length augmenting paths. Each such construction is called a *phase*.

The following algorithm describes one phase. As in Edmonds-Karp, there are at most n phases, because with each phase the minimum distance from s to t in the residual graph increases by at least one. We traverse the level graph from source to sink in a depth-first fashion, advancing whenever possible and keeping track of the path from s to the current vertex. If we get all the way to t, we have found an augmenting path, and we augment by that path. If we get to a vertex with no outgoing edges, we delete that vertex (there is no path to t through it) and retreat.

In the following, u denotes the vertex currently being visited and p is a path from s to u.

Algorithm 18.1 (Dinic [29])

Initialize. Construct a new level graph L_G. Set $u := s$ and $p := [s]$. Go to **Advance**.

Advance. If there is no edge out of u, go to **Retreat**. Otherwise, let (u, v) be such an edge. Set $p := p \cdot [v]$ and $u := v$. If $v \neq t$ then go to **Advance**. If $v = t$ then go to **Augment**.

Retreat. If $u = s$ then halt. Otherwise, delete u and all adjacent edges from L_G and remove u from the end of p. Set $u :=$ the last vertex on p. Go to **Advance**.

Augment. Let Δ be the bottleneck capacity along p. Augment by the path flow along p of value Δ, adjusting residual capacities along p. Delete newly saturated edges. Set $u :=$ the last vertex on the path p reachable from s along unsaturated edges of p; that is, the start vertex of the first newly saturated edge on p. Set $p :=$ the portion of p up to and including u. Go to **Advance**.

We now discuss the complexity of these operations.

Initialize. This is executed only once per phase and takes $O(m)$ time using BFS.

Advance. There are at most $2mn$ advances in each phase, because there can be at most n advances before an augment or retreat, and there are at most m augments and m retreats. Each advance takes constant time, so the total time for all advances is $O(mn)$.

Retreat. There are at most n retreats in each phase, because at least one vertex is deleted in each retreat. Each retreat takes $O(1)$ time plus the time to delete edges, which in all is $O(m)$; thus the time taken by all retreats in a phase is $O(m + n)$.

Augment. There are at most m augments in each phase, because at least one edge is deleted each time. Each augment takes $O(n)$ time, or $O(mn)$ time in all.

Each phase then requires $O(mn)$ time. Because there are at most n phases, the total running time is $O(mn^2)$.

18.2 The MPM Algorithm

The following algorithm given by Malhotra, Pramodh-Kumar, and Maheshwari in 1978 [77] produces a max flow in $O(n^3)$ time. The overall structure is

similar to the Edmonds-Karp or Dinic algorithms. Blocking flows are found for level graphs of increasing depth. The algorithm's superior time bound is due a faster ($O(n^2)$) method for producing a blocking flow.

For this algorithm, we need to consider the capacity of a vertex as opposed to the capacity of an edge. Intuitively, the capacity of a vertex is the maximum amount of commodity that can be pushed through that vertex.

Definition 18.2 The *capacity* $c(v)$ of a vertex v is the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:

$$c(v) = \min\{\sum_{u \in V} c(u,v), \sum_{u \in V} c(v,u)\} .$$

□

This definition applies as well to residual capacities obtained by subtracting a nonzero flow.

The MPM algorithm proceeds in phases. In each phase, the residual graph is computed for the current flow, and the level graph L is computed. If t does not appear in L, we are done. Otherwise, all vertices not on a path from s to t in the level graph are deleted.

Now we repeat the following steps until a blocking flow is achieved:

1. Find a vertex v of minimum capacity d according to Definition 18.2. If $d = 0$, do step 2. If $d \neq 0$, do step 3.
2. Delete v and all incident edges and update the capacities of the neighboring vertices. Go to 1.
3. Push d units of flow from v to the sink and pull d units of flow from the source to v to increase the flow through v by d. This is done as follows:

   Push to sink. The outgoing edges of v are saturated in order, leaving at most one partially saturated edge. All edges that become saturated during this process are deleted. This process is then repeated on each vertex that received flow during the saturation of the edges out of v, and so on all the way to t. It is always possible to push all d units of flow all the way to t, since every vertex has capacity at least d.

   Pull from source. The incoming edges of v are saturated in order, leaving at most one partially saturated edge. All edges that become saturated by this process are deleted. This process is then repeated on each vertex from which flow was taken during the saturation of the edges into v, and so on all the way back to s. It is always possible to pull all d units of flow all the way back to s, since every vertex has capacity at least d.

   Either all incoming edges of v or all outgoing edges of v are saturated and hence deleted, so v and all its remaining incident edges can be deleted from the level graph, and the capacities of the neighbors updated. Go to 1.

It takes $O(m)$ time to compute the residual graph for the current flow and level graph using BFS. Using Fibonacci heaps, it takes $O(n \log n)$ time amortized over all iterations of the loop to find and delete a vertex of minimum capacity. It takes $O(m)$ time over all iterations of the loop to delete all the fully saturated edges, since we spend $O(1)$ time for each such edge. It takes $O(n^2)$ time over all iterations of the loop to do the partial saturations, because it is done at most once in step 3 at each vertex for each choice of v in step 1.

Note that when we delete edges, we must decrement the capacities of neighboring vertices; this is done using the decrement facility of Fibonacci heaps.

The loop thus achieves a blocking flow in $O(n^2)$ time. As before, at most n blocking flows have to be computed, because the distance from s to t in the level graph increases by at least one each time. This gives an overall worst-case time bound of $O(n^3)$.

The max flow problem is still an active topic of research. Although $O(n^3)$

remains the best known time bound for general graphs, new approaches to the max flow problem and better time bounds for sparse graphs have appeared more recently [38, 98, 4, 41, 95, 37].

18.3 Applications of Max Flow

Bipartite Matching

Definition 18.3 A *matching* M of a graph G is a subset of edges such that no two edges in M share a vertex. We denote the size of M by $|M|$. A *maximum matching* is one of maximum size. □

We can use any max flow algorithm to produce a maximum matching in a bipartite graph $G = (U, V, E)$ as follows. Add a new source vertex s and a new sink vertex t, connect s to every vertex in U, and connect every vertex in V to t. Assign every edge capacity 1. The edges from U to V used by a maximum integral flow give a maximum matching.

Minimum Connectivity

Let $G = (V, E)$ be a connected undirected graph. What is the least number of edges we need to remove in order to disconnect G? This is known as the *minimum connectivity problem.*

The minimum connectivity problem can be solved by solving $n - 1$ max flow problems. Replace each undirected edge with two directed edges, one in each direction. Assign capacity 1 to each edge. Let s be a fixed vertex in V and let t range over all other vertices. Find the max flow for each value of t, and take the minimum over all choices of t. This also gives a minimum cut, which gives a solution to the minimum connectivity problem.