A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

— The First Law of Mentat, in Frank Herbert's *Dune* (1965)

There's a difference between knowing the path and walking the path.

— Morpheus [Laurence Fishburne], *The Matrix* (1999)

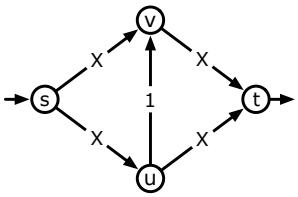
22 Max-Flow Algorithms

22.1 Ford and Fulkerson's augmenting paths

Ford and Fulkerson's proof of the Maxflow-Mincut Theorem, described in the previous lecture note, translates immediately to an algorithm to compute maximum flows: Starting with the zero flow, repeatedly augment the flow along **any** path $s \rightsquigarrow t$ in the residual graph, until there is no such path.

If every edge capacity is an integer, then every augmentation step increases the value of the flow by a positive integer. Thus, the algorithm halts after $|f^*|$ iterations, where f^* is the actual maximum flow. Each iteration requires $O(E)$ time, to create the residual graph G_f and perform a whatever-first-search to find an augmenting path. Thus the Ford-Fulkerson algorithm runs in $\boldsymbol{O(E|f^*|)}$ time in the worst case.

The following example shows that this running time analysis is essentially tight. Consider the 4-node network illustrated below, where X is some large integer. The maximum flow in this network is clearly $2X$. However, Ford-Fulkerson might alternate between pushing 1 unit of flow along the augmenting path $s \to u \to v \to t$ and then pushing 1 unit of flow along the augmenting path $s \to v \to u \to t$, leading to a running time of $\Theta(X) = \Omega(|f^*|)$.



A bad example for the Ford-Fulkerson algorithm.

Ford and Fulkerson's algorithm works quite well in many practical situations, or in settings where the maximum flow value $|f^*|$ is small, but without further constraints on the augmenting paths, this is *not* an efficient algorithm in general. The example network above can be described using only $O(\log X)$ bits; thus, the running time of Ford-Fulkerson is *exponential* in the input size.

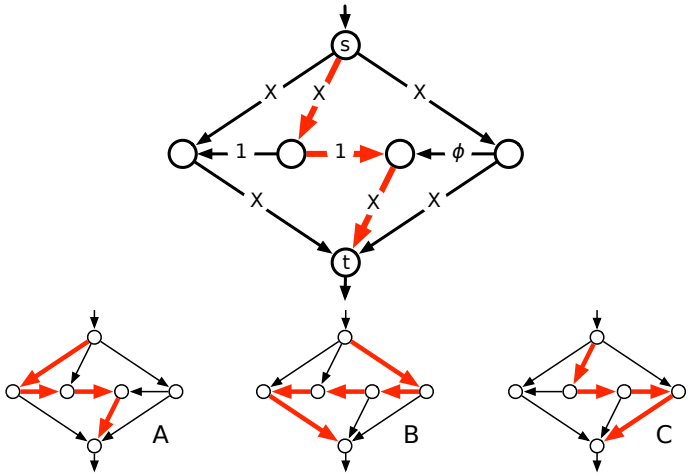
22.2 Irrational Capacities

If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts, although still in exponential time.

However, if we allow *irrational* capacities, the algorithm can actually loop forever, always finding smaller and smaller augmenting paths! Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow, or even to a significant fraction of the maximum flow! Perhaps the simplest example of this effect was discovered by Uri Zwick.

Consider the six-node network shown on the next page. Six of the nine edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5} - 1)/2 \approx 0.618034$, chosen so

that $1 - \phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we can watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X - 3$.)



Uri Zwick's non-terminating flow example, and three augmenting paths.

Suppose the Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown in the large figure above. The three horizontal edges, in order from left to right, now have residual capacities 1, 0, and ϕ. Suppose inductively that the horizontal residual capacities are ϕ^{k-1}, 0, ϕ^k for some non-negative integer k.

1. Augment along B, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, \phi^k, 0$.
2. Augment along C, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^k$.
3. Augment along B, adding ϕ^{k+1} to the flow; the residual capacities are now $0, \phi^{k+1}, \phi^{k+2}$.
4. Augment along A, adding ϕ^{k+1} to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^{k+2}$.

It follows by induction that after $4n+1$ augmentation steps, the horizontal edges have residual capacities $\phi^{2n-2}, 0, \phi^{2n-1}$. As the number of augmentations grows to infinity, the value of the flow converges to

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

even though the maximum flow value is clearly $2X + 1 \gg 7$.

Picky students might wonder at this point why we care about irrational capacities; after all, computers can't represent anything but (small) integers or (dyadic) rationals exactly. Good question! One reason is that the integer restriction is literally *artificial*; it's an *artifact* of actual computational hardware[1], not an inherent feature of the abstract mathematical problem. Another reason, which is probably more convincing to most practical computer scientists, is that the behavior of the algorithm with irrational inputs tells us something about its worst-case behavior *in practice* given floating-point capacities—terrible! Even with very reasonable capacities, a careless implementation of Ford-Fulkerson could enter an infinite loop simply because of round-off error.

[1]...or perhaps the laws of physics. Yeah, whatever. Like *reality* actually matters in this class.

22.3 Edmonds-Karp: Fat Pipes

The Ford-Fulkerson algorithm does not specify which alternating path to use if there is more than one. In 1972, Jack Edmonds and Richard Karp analyzed two natural heuristics for choosing the path. The first is essentially a greedy algorithm:

> Choose the augmenting path with largest bottleneck value.

It's a fairly easy to show that the maximum-bottleneck (s,t)-path in a directed graph can be computed in $O(E \log V)$ time using a variant of Jarník's minimum-spanning-tree algorithm, or of Dijkstra's shortest path algorithm. Simply grow a directed spanning tree T, rooted at s. Repeatedly find the highest-capacity edge leaving T and add it to T, until T contains a path from s to t. Alternately, one could emulate Kruskal's algorithm—insert edges one at a time in decreasing capacity order until there is a path from s to t—although this is less efficient.

We can now analyze the algorithm in terms of the value of the maximum flow f^*. Let f be any flow in G, and let f' be the maximum flow *in the current residual graph* G_f. (At the beginning of the algorithm, $G_f = G$ and $f' = f^*$.) Let e be the bottleneck edge in the next augmenting path. Let S be the set of vertices reachable from s through edges in G with capacity greater than $c(e)$ and let $T = V \setminus S$. By construction, T is non-empty, and every edge from S to T has capacity at most $c(e)$. Thus, the capacity of the cut (S,T) is at most $c(e) \cdot E$. On the other hand, the maxflow-mincut theorem implies that $\|S,T\| \geq |f|$. We conclude that $c(e) \geq |f|/E$.

The preceding argument implies that augmenting f along the maximum-bottleneck path in G_f multiplies the maximum flow value in G_f by a factor of at most $1 - 1/E$. In other words, the residual flow *decays exponentially* with the number of iterations. After $E \cdot \ln|f^*|$ iterations, the maximum flow value in G_f is at most

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln|f^*|} < |f^*|\, e^{-\ln|f^*|} = 1.$$

(That's Euler's constant e, not the edge e. Sorry.) In particular, *if all the capacities are integers*, then after $E \cdot \ln|f^*|$ iterations, the maximum capacity of the residual graph is *zero* and f is a maximum flow.

We conclude that for graphs with integer capacities, the Edmonds-Karp 'fat pipe' algorithm runs in $\boldsymbol{O(E^2 \log E \log|f^*|)}$ time, which is actually a polynomial function of the input size.

22.4 Dinits/Edmonds-Karp: Short Pipes

The second Edmonds-Karp heuristic was actually proposed by Ford and Fulkerson in their original max-flow paper, and first analyzed by the Russian mathematician Dinits (sometimes transliterated Dinic) in 1970. Edmonds and Karp published their independent and slightly weaker analysis in 1972. So naturally, almost everyone refers to this algorithm as 'Edmonds-Karp'.[2]

> Choose the augmenting path with fewest edges.

The correct path can be found in $O(E)$ time by running breadth-first search in the residual graph. More surprisingly, the algorithm halts after a polynomial number of iterations, independent of the actual edge capacities!

[2]To be fair, Edmonds and Karp discovered their algorithm a few years before publication—getting ideas into print takes time, especially in the early 1970s—which is why some authors believe they deserve priority. I don't buy it; Dinits *also* presumably discovered his algorithm a few years before *its* publication. (In Soviet Union, result publish you.) On the gripping hand, Dinits's paper also described an improvement to the algorithm presented here that runs in $O(V^2E)$ time instead of $O(VE^2)$, so maybe *that* ought to be called Dinits's algorithm.

The proof of this upper bound relies on two observations about the evolution of the residual graph. Let f_i be the current flow after i augmentation steps, let G_i be the corresponding residual graph. In particular, f_0 is zero everywhere and $G_0 = G$. For each vertex v, let $level_i(v)$ denote the unweighted shortest path distance from s to v in G_i, or equivalently, the *level* of v in a breadth-first search tree of G_i rooted at s.

Our first observation is that these levels can only increase over time.

Lemma 1. *$level_{i+1}(v) \geq level_i(v)$ for all vertices v and integers i.*

Proof: The claim is trivial for $v = s$, since $level_i(s) = 0$ for all i. Choose an arbitrary vertex $v \neq s$, and let $s \to \cdots \to u \to v$ be a shortest path from s to v in G_{i+1}. (If there is no such path, then $level_{i+1}(v) = \infty$, and we're done.) Because this is a shortest path, we have $level_{i+1}(v) = level_{i+1}(u) + 1$, and the inductive hypothesis implies that $level_{i+1}(u) \geq level_i(u)$.

We now have two cases to consider. If $u \to v$ is an edge in G_i, then $level_i(v) \leq level_i(u) + 1$, because the levels are defined by breadth-first traversal.

On the other hand, if $u \to v$ is not an edge in G_i, then $v \to u$ must be an edge in the ith augmenting path. Thus, $v \to u$ must lie on the shortest path from s to t in G_i, which implies that $level_i(v) = level_i(u) - 1 \leq level_i(u) + 1$.

In both cases, we have $level_{i+1}(v) = level_{i+1}(u) + 1 \geq level_i(u) + 1 \geq level_i(v)$. □

Whenever we augment the flow, the bottleneck edge in the augmenting path disappears from the residual graph, and some other edge in the *reversal* of the augmenting path may (re-)appear. Our second observation is that an edge cannot appear or disappear too many times.

Lemma 2. *During the execution of the Dinits/Edmonds-Karp algorithm, any edge $u \to v$ disappears from the residual graph G_f at most $V/2$ times.*

Proof: Suppose $u \to v$ is in two residual graphs G_i and G_{j+1}, but not in any of the intermediate residual graphs $G_{i+1}, \ldots, G_j$, for some $i < j$. Then $u \to v$ must be in the ith augmenting path, so $level_i(v) = level_i(u) + 1$, and $v \to u$ must be on the jth augmenting path, so $level_j(v) = level_j(u) - 1$. By the previous lemma, we have

$$level_j(u) = level_j(v) + 1 \geq level_i(v) + 1 = level_i(u) + 2.$$

In other words, the distance from s to u increased by at least 2 between the disappearance and reappearance of $u \to v$. Since every level is either less than V or infinite, the number of disappearances is at most $V/2$. □

Now we can derive an upper bound on the number of iterations. Since each edge can disappear at most $V/2$ times, there are at most $EV/2$ edge disappearances overall. But at least one edge disappears on each iteration, so the algorithm must halt after at most $EV/2$ iterations. Finally, since each iteration requires $O(E)$ time, Dinits' algorithm runs in $\boldsymbol{O(VE^2)}$ ***time*** overall.

22.5 Further Progress

This is nowhere near the end of the story for maximum-flow algorithms. Decades of further research have led to a number of even faster algorithms, some of which are summarized in the table below.[3] All of the algorithms listed below compute a maximum flow in several iterations. Each algorithm has two

[3]To keep the table short, I have deliberately omitted algorithms whose running time depends on the maximum capacity, the sum of the capacities, or the maximum flow value. Even with this restriction, the table is incomplete!

variants: a simpler version that performs each iteration by brute force, and a faster variant that uses sophisticated data structures to maintain a spanning tree of the flow network, so that each iteration can be performed (and the spanning tree updated) in logarithmic time. There is no reason to believe that the best algorithms known so far are optimal; indeed, maximum flows are still a very active area of research.

Technique	Direct	With dynamic trees	Sources
Blocking flow	$O(V^3)$	$O(VE \log V)$	[Dinits; Sleator and Tarjan]
Network simplex	$O(V^2E)$	$O(VE \log V)$	[Dantzig; Goldfarb and Hao; Goldberg, Grigoriadis, and Tarjan]
Push-relabel (generic)	$O(V^2E)$	—	[Goldberg and Tarjan]
Push-relabel (FIFO)	$O(V^3)$	$O(V^2 \log(V^2/E))$	[Goldberg and Tarjan]
Push-relabel (highest label)	$O(V^2\sqrt{E})$	—	[Cheriyan and Maheshwari; Tunçel]
Pseudoflow	$O(V^2E)$	$O(VE \log V)$	[Hochbaum]
Compact abundance graphs		$O(VE)$	[Orlin 2012]

Several purely combinatorial maximum-flow algorithms and their running times.

The fastest known maximum flow algorithm, announced by James Orlin in 2012, runs in ***O(VE) time***. The details of Orlin's algorithm are far beyond the scope of this course; in addition to his own new techniques, Orlin uses several existing algorithms and data structures as black boxes, most of which are themselves quite complicated. Nevertheless, for purposes of analyzing algorithms that use maximum flows, this is the time bound you should cite. So write the following sentence on your cheat sheets and cite it in your homeworks:

> ***Maximum flows can be computed in O(VE) time.***

Exercises

1. For any flow network G and any vertices u and v, let $bottleneck_G(u, v)$ denote the maximum, over all paths π in G from u to v, of the minimum-capacity edge along π.

   (a) Describe and analyze an algorithm to compute $bottleneck_G(s, t)$ in $O(E \log V)$ time.

   (b) Describe an algorithm to construct a spanning tree T of G such that $bottleneck_T(u, v) = bottleneck_G(u, v)$ for *all* vertices u and v. (Edges in T inherit their capacities from G.)

2. Describe an efficient algorithm to determine whether a given flow network contains a *unique* maximum flow.

3. Suppose you have already computed a maximum flow f^* in a flow network G with ***integer*** edge capacities.

   (a) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is increased by 1.

   (b) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is decreased by 1.

   Both algorithms should be significantly faster than recomputing the maximum flow from scratch.

4. Let G be a network with integer edge capacities. An edge in G is *upper-binding* if increasing its capacity by 1 also increases the value of the maximum flow in G. Similarly, an edge is *lower-binding* if decreasing its capacity by 1 also decreases the value of the maximum flow in G.

   (a) Does every network G have at least one upper-binding edge? Prove your answer is correct.

   (b) Does every network G have at least one lower-binding edge? Prove your answer is correct.

   (c) Describe an algorithm to find all upper-binding edges in G, given both G and a maximum flow in G as input, in $O(E)$ time.

   (d) Describe an algorithm to find all lower-binding edges in G, given both G and a maximum flow in G as input, in $O(EV)$ time.

5. A given flow network G may have more than one minimum (s,t)-cut. Let's define the ***best*** minimum (s,t)-cut to be any minimum cut with the smallest number of edges.

   (a) Describe an efficient algorithm to determine whether a given flow network contains a *unique* minimum (s,t)-cut.

   (b) Describe an efficient algorithm to find the best minimum (s,t)-cut when the capacities are integers.

   (c) Describe an efficient algorithm to find the best minimum (s,t)-cut for *arbitrary* edge capacities.

   (d) Describe an efficient algorithm to determine whether a given flow network contains a unique *best* minimum (s,t)-cut.

6. A new assistant professor, teaching maximum flows for the first time, suggests the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, just reduce the capacity of edges along the augmenting path! In particular, whenever we saturate an edge, just remove it from the graph.

```
GreedyFlow(G, c, s, t):
  for every edge e in G
      f(e) ← 0

  while there is a path from s to t
      π ← an arbitrary path from s to t
      F ← minimum capacity of any edge in π
      for every edge e in π
          f(e) ← f(e) + F
          if c(e) = F
              remove e from G
          else
              c(e) ← c(e) − F

  return f
```

   (a) Show that this algorithm does *not* always compute a maximum flow.

   (b) Prove that for any flow network, if the Greedy Path Fairy tells you precisely which path π to use at each iteration, then GreedyFlow does compute a maximum flow. (Sadly, the Greedy Path Fairy does not actually exist.)

7. We can speed up the Edmonds-Karp 'fat pipe' heuristic, at least for integer capacities, by relaxing our requirements for the next augmenting path. Instead of finding the augmenting path with maximum bottleneck capacity, we find a path whose bottleneck capacity is at least half of maximum, using the following *capacity scaling* algorithm.

   The algorithm maintains a bottleneck threshold Δ; initially, Δ is the maximum capacity among all edges in the graph. In each *phase*, the algorithm augments along paths from s to t in which every edge has residual capacity at least Δ. When there is no such path, the phase ends, we set $\Delta \leftarrow \lfloor \Delta/2 \rfloor$, and the next phase begins.

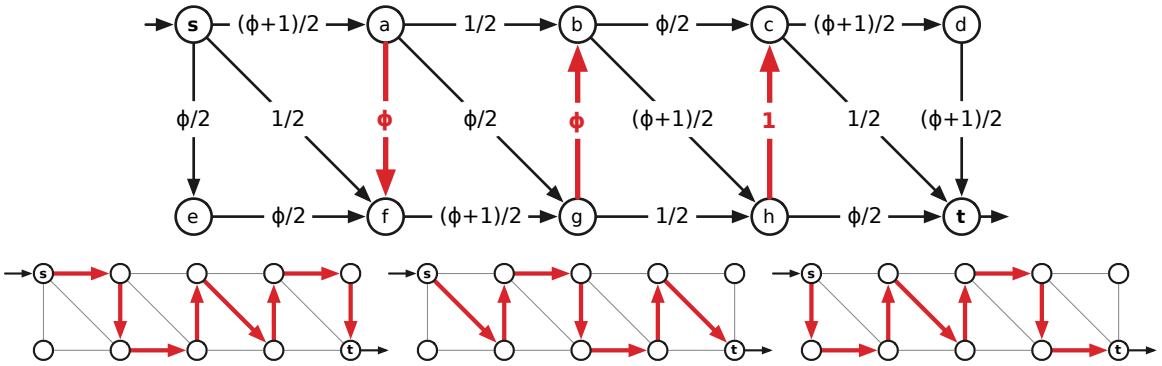
   (a) How many phases will the algorithm execute in the worst case, if the edge capacities are integers?

   (b) Let f be the flow at the end of a phase for a particular value of Δ. Let S be the nodes that are reachable from s in the residual graph G_f using only edges with residual capacity at least Δ, and let $T = V \setminus S$. Prove that the capacity (with respect to G's original edge capacities) of the cut (S, T) is at most $|f| + E \cdot \Delta$.

   (c) Prove that in each phase of the scaling algorithm, there are at most $2E$ augmentations.

   (d) What is the overall running time of the scaling algorithm, assuming all the edge capacities are integers?

8. In 1980 Maurice Queyranne published the following example of a flow network where the Edmonds-Karp 'fat pipe' heuristic does not halt. Here, as in Zwick's bad example for the original Ford-Fulkerson algorithm, ϕ denotes the inverse golden ratio $(\sqrt{5}-1)/2$. The three vertical edges play essentially the same role as the horizontal edges in Zwick's example.



Queyranne's network, and a sequence of "fat-pipe" augmentations.

   (a) Show that the following infinite sequence of path augmentations is a valid execution of the Edmonds-Karp algorithm. (See the figure above.)

```
QUEYRANNEFATPIPES:
  for i ← 1 to ∞
      push ϕ^(3i−2) units of flow along  s→a→f→g→b→h→c→d→t
      push ϕ^(3i−1) units of flow along  s→f→a→b→g→h→c→t
      push ϕ^(3i)   units of flow along  s→e→f→a→g→b→c→h→t
  forever
```

   (b) Describe a sequence of $O(1)$ path augmentations that yields a maximum flow in Queyranne's network.