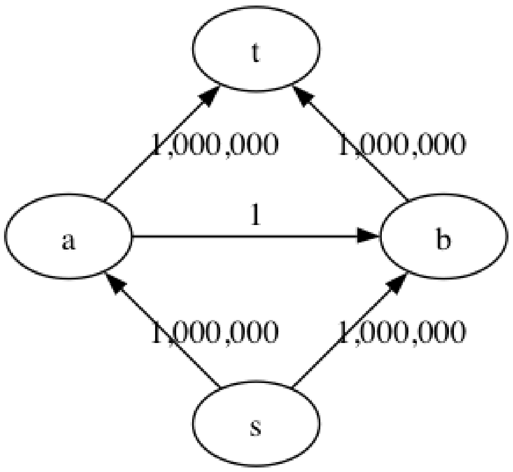
Stanford University — CS261: Optimization Handout 10
Luca Trevisan February 3, 2011

Lecture 10

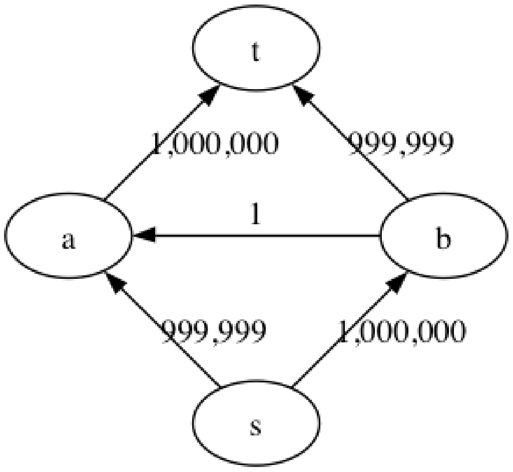
In which we discuss the worst-case running of the Ford-Fulkerson algorithm, discuss plausible heuristics to choose an augmenting path in a good way, and begin analyzing the "fattest path" heuristic.

In the last lecture we proved the Max-Flow Min-Cut theorem in a way that also established the optimality of the Ford-Fulkerson algorithm: if we iteratively find an augmenting path in the residual network and push more flow along that path, as allowed by the capacity constraints, we will eventually find a flow for which no augmenting path exists, and we proved that such a flow must be optimal.

Each iteration of the algorithm takes linear time in the size of the network: the augmenting path can be found via a DFS of the residual network, for example. The problem is that, in certain cases, the algorithm might take a very long time to finish. Consider, for example, the following network.



Suppose that, at the first step, we pick the augmenting path $s \to a \to b \to t$. We can only push one unit of flow along that path. After this first step, our residual network (not showing edges out of t and into s, which are never used in an augmenting path) is



Now it is possible that the algorithm picks the augmenting path $s \to b \to a \to t$ along which, again, only one unit of flow can be routed. We see that, indeed, it is possible for the algorithm to keep picking augmenting paths that involve a link between a and b, so that only one extra unit of flow is routed at each step.

The problem of very slow convergence times as in the above example can be avoided if, at each iteration, we choose more carefully which augmenting path to use. One reasonable heuristic is that it makes sense to pick the augmenting path along which the most flow can be routed in one step. If we had used such an heuristic in the above example, we would have found the optimum in two steps. Another, alternative, heuristic is to pick the shortest augmenting path, that is, the augmenting path that uses the fewest edges; this is reasonable because in this way we are going to use the capacity of fewer edges and keep more residual capacity for later iterations. The use of this heuristic would have also resulted in a two-iterations running time in the above example.

1 The "fattest" augmenting path heuristic

We begin by studying the first heuristic: that is we consider an implementation of the Ford-Fulkerson algorithm in which, at every iteration, we pick a *fattest* augmenting path in the residual network, where the fatness of a path in a capacitated network is the minimum capacity of the edges in the path. In the network of our previous example, the paths $s \to a \to t$ and $s \to b \to t$ have fatness $1,000,000$, while the path $s \to a \to b \to t$ has fatness 1.

How do we find a fattest augmenting path? We will show that it can be found with a simple modification of Dijkstra's algorithm for finding shortest paths.

1.1 Dijkstra's algorithm

Let us first quickly recall how Dijkstra's algorithm works. Suppose that we have a graph in which each edge (u, v) has a length $\ell(u, v)$ and, for two given vertices s, t, we want to find the path of minimal length from s to t, where the length of a path is the sum of the lengths of the edges in the path. The algorithm will solve, for free, the more general problem of computing the length of the shortest path from s to v for every vertex v. In the algorithm, the data structure that holds information about a vertex v has two fields: $v.dist$, which will eventually contain the length of the shortest path from s to v, and $v.pred$ which will contain the *predecessor* of v in a shortest path from s to v, that is, the identity of the vertex that comes immediately before v in such a path.

The distances are initialized to $+\infty$, except for $s.dist$ which is initialized to zero. The algorithm initially puts all vertices in a *priority queue* Q. Recall that a priority queue is a data structure that contains elements which have a numerical field called a *key* (in this case the key is the *dist* field), and that supports the operation of inserting an element in the queue, of finding and removing from the queue the element of minimal key value, and of reducing the key field of a given element.

The algorithm works as follows:

```
Algorithm Dijkstra

• Input: graph $G = (V, E)$, vertex $s \in V$, non-negative edge lengths $\ell(\cdot, \cdot)$
• for each $v \in V - \{s\}$, let $v.dist = \infty$
• $s.dist = 0$
• insert all vertices in a priority queue $Q$ keyed by the dist field
• while $Q$ is not empty
    – find and remove vertex $u$ in $Q$ whose field $u.dist$ is smallest among queue
      elements
    – for all vertices $v$ such that $(u, v) \in E$
        ∗ if $v.dist > u.dist + \ell(u, v)$ then
            · $v.dist := u.dist + \ell(u, v)$
            · update $Q$ to reflect changed value of $v.dist$
            · $u.pred := v$
```

The running time is equal to whatever time it takes to execute $|V|$ *insert* operations, $|V|$ *remove-min* operations, and $|E|$ *reduce-key* operations in the priority queue. The simple implementation of a priority queue via a binary heap gives $O(\log |V|)$ running time for each operation, and a total running time of $O((|E| + |V|) \cdot \log |V|)$. A more elaborate data structure called a *Fibonacci heap* implements *insert* and *remove-min* in $O(\log |V|)$ time, and is such that k *decrease-key* operations always take at most $O(k)$ time overall, so that the total running time is $O(|V| \log |V| + |E|)$.

Regarding correctness, we can prove by induction that the algorithm maintains the following invariant: at the beginning of each iteration of the *while* loop, the vertices x which are *not* in the queue are such that $x.dist$ is the correct value of the shortest path length from s to x and such a shortest path can be realized by combining a shortest path from s to $x.pred$ and then continuing with the edge $(x.pred, x)$. This is certainly true at the beginning, because the first vertex to be removed is s, which is at distance $s.dist = 0$ from itself, and if it is true at the end, when the queue is empty, it means that at the end of the algorithm all vertices get their correct values of $x.dist$ and $x.pred$. So we need to show that if the invariant is true at a certain step then it is true at the following step.

Basically, all we need to prove is that, at the beginning of each iteration, the vertex u that we remove from the queue has correct values of $u.dist$ and $u.pred$. If we call $x := u.pred$, then x is a vertex that was removed from the queue at an earlier iteration and so, by the inductive hypothesis, is such that $x.dist$ is the correct shortest path distance from s to x; if $x = u.pred$ we also have $u.dist = x.dist + \ell(u, v)$, which means that there is indeed a path of length $u.dist$ from s to u in which x is the predecessor of u. We need to prove that this path is a shortest path. So suppose toward a contradiction that there is a shorter path p of length $< u.dist$. The path p starts at s, which is outside the queue, and ends at u, which is in the queue, so at some point the path must have an edge (y, z) such that y is outside the queue and z is inside. This also means that when y was removed from the queue it had the correct value $y.dist$, and after we processed the neighbors of y we had $z.dist \leq y.dist + \ell(u, v)$. But this would mean that $z.dist$ is at most the length of the path p, while $u.dist$ is bigger than the length of the path p, which is impossible because u was chosen to be the element with the smallest $u.dist$ among elements of the queue.

1.2 Adaptation to find a fattest path

What would be the most straightforward adaptation of Dijkstra's algorithm to the problem of finding a fattest path? In the shortest path problem, the length of a path is the *sum* of the *lengths* of the edges of the path, and we want to find a path of *minimal* length; in the fattest path problem, the fatness of a path is the *minimum* of the *capacities* of the edges of the path, and we want to find a path of *maximal* fatness. So we just change sums to min, lengths to capacities, and minimization to maximization.

```
Algorithm Dijkstra-F

  • Input: graph $G = (V, E)$, vertex $s \in V$, non-negative edge capacities $c(\cdot, \cdot)$
  • for each $v \in V - \{s\}$, let $v.fat = 0$
  • $s.dist = \infty$
  • insert all vertices in a priority queue $Q$ keyed by the dist field
  • while $Q$ is not empty
      – find and remove vertex $u$ in $Q$ whose field $u.fat$ is largest among queue elements
      – for all vertices $v$ such that $(u, v) \in E$
          ∗ if $v.fat < \min\{u.fat, c(u, v)\}$ then
              · $v.fat := \min\{u.fat, c(u, v)\}$
              · update $Q$ to reflect changed value of $v.dist$
              · $u.pred := v$
```

The running time is the same and, quite amazingly, the proof of correctness is also essentially the same. (Try writing it up.)

Remark 1 *A useful feature of Dijkstra's algorithm (and other shortest path algorithms) is that it works to find "best" paths for a lot of different measures of "cost" for a path, besides length and fatness. Basically, the only requirements to implement the algorithm and prove correctness are:*

- *The cost of a path $u_1 \to u_2 \to \cdots u_t$ is no better than the cost of an initial segment $u_1 \to u_2 \to \cdots u_k$, $k < t$ of the path. That is, if we are trying to maximize the cost, we need the property that the cost of a path is at most the cost of any initial segment (e.g., the fatness of a path is at most the fatness of any initial segment, because in the former case we are taking the minimum over a larger set of capacities); if we are trying to minimize the cost, we need the property that the cost of a path is at least the cost of any initial segment.*
- *The cost of a path $u_1 \to u_2 \to \cdots \to u_{t-1} \to u_t$ can be determined by only knowing the cost of the path $u_1 \to u_2 \to \cdots \to u_{t-1}$ and the cost of the edge (u_{t-1}, u_t).*

1.3 Analysis of the fattest augmenting path heuristic

In the next lecture, we will prove the following result.

Theorem 2 *If $(G = (V, E), s, t, c)$ is a network in which the optimal flow has cost opt, then there is a path from s to t of fatness $\geq opt/m$.*

From the above theorem, we se that if we implement the Ford-Fulkerson algorithm with the fattest-path heuristic, then, after we have found t augmenting paths, we have a solution such that, in the residual network, the optimum flow has cost at most $opt \cdot \left(1 - \frac{1}{2|E|}\right)^t$.

To see why, call $flow_i$ the cost of the flow found by the algorithm after i iterations, and res_i the optimum of the residual network after i iterations of the algorithm. Clearly we have $res_i = opt - flow_i$.

The theorem tells us that at iteration $i+1$ we are going to find an augmenting path of fatness at least $res_i \cdot \frac{1}{2|E|}$. (Because of the "virtual capacities," the residual network could have as many as twice the number of edges of the original network, but no more.) This means that the cost of the flow at the end of the $(i+1)$-th iteration is going to be $flow_{i+1} \geq flow_i + res_i \cdot \frac{1}{2|E|}$, which means that the residual optimum is going to be

$$res_{i+1} = opt - flow_{i+1} \leq opt - \left(flow_i - res_i \cdot \frac{1}{2|E|}\right) = res_i \cdot \left(1 - \frac{1}{2|E|}\right)$$

We started with $flow_0 = 0$ and $res_0 = opt$, and so we must have $res_t \leq opt \cdot \left(1 - \frac{1}{2|E|}\right)^t$.

If the capacities are integers, then if the residual network has an optimum less than 1, its optimum must be zero. Recalling that $1 - x \leq e^{-x}$,

$$res_t \leq opt \left(1 - \frac{1}{2|E|}\right)^t \leq opte^{-t/2|E|} = e^{\ln opt - t/2|E|}$$

This means that if $t > 2|E| \ln opt$, then $res_t < 1$, which implies $res_t = 0$ and so it means that, within the first $1 + 2|E| \ln opt$ steps, the algorithm reaches a point in which the residual network has no augmenting path and it stops.

We said that, using the simple binary heap implementation of Dijkstra's algorithm, the running time of one iteration is $O((|V|+|E|) \cdot \log |V|)$, and so we have the following analysis.

Theorem 3 *The fattest-path implementation of the Ford-Fulkerson algorithm, given in input a network with integer capacities whose optimal flow has cost opt, runs in time at most*

$$O((|V| + |E|) \cdot |E| \cdot \log |V| \cdot \log opt)$$

To complete the above running time analysis, we need to prove Theorem 1, which we will do next time.