ICS 260—Fall, 2000–Class Notes 4: Polynomial Time Augmenting Path Algorithms

Notes Originally Developed by Sandy Irani
Modified by Michael B. Dillencourt

1 Polynomial Time Algorithms Using Augmenting Paths

[AMO 7.1, 7.3, 7.4]

Typically, when proving worst case bounds, we are looking for an algorithm which is polynomial. But polynomial in what? If we can bound the running time of an algorithm by a polynomial in the number of numbers that is independent of the values of those numbers, then we say that the algorithm is *strongly polynomial.* In the case of network flows, this would mean an algorithm whose running time is bounded by polynomial in n and m. An algorithm that is polynomial-time, is an algorithm whose running time is bounded by a polynomial in the number of bits required to specify the input. In this case, it would mean an algorithm whose running time is bounded by a polynomial in n, m and $\log(c_{max})$, assuming that the capacities are integral. Any bound which is polynomial in c_{max} is not polynomial in the number of bits necessary to specify the input. Thus, the Ford-Fulkerson algorithm is not a polynomial-time algorithm. An algorithm whose running time is bounded by a polynomial in the number of numbers and the *value* of those numbers (like Ford-Fulkerson), is said to be a *pseudopolynomial* time algorithm. In this lecture, we will examine variations of the Ford-Fulkerson algorithm which achieve a polynomial time bound by picking the augmenting path more carefully. The last algorithm we will discuss is strongly polynomial.

What are some ways that we can pick the augmenting path more cleverly so that we can improve the running time of the basic Ford-Fulkerson algorithm? We will look at two basic means of improving the augmenting path found. These are:

- Pick an augmenting path with a large capacity, so that the current flow will be augmented by a large amount.
- Pick an augmenting path which is short, in terms of the number of edges.

1.1 Augmenting Along the Maximum Capacity Path

What if we always pick the augmenting path with the maximum capacity? How many augmenting paths must be found until the max flow is achieved? How long will it take to find the augmenting path in each iteration? We can use a variation of Dilkstra's algorithm to find the augmenting path with the maximum capacity. This will take $O(m \log n)$ or $O(n \log n + m)$, depending on which implementation is of Dijkstra's algorithm is used. How may iterations will be performed?

We will use the following *Flow Decomposition Theorem* which we will state without proof:

Theorem 1 (Flow Decomposition Theorem) *Any s-t flow f can be decomposed into at most m primitive elements, where a primitive element is either a path or a cycle.*

Suppose we have a flow f on the graph and let f^* denote the value of the maximum flow. This means that, the residual network has a flow of value $f^* - |f|$. By the decomposition theorem, this can be composed into at most m paths. Thus, there is a path in G_f with flow at least $(f^* - |f|)/m$ and this means that the maximum augmenting path will have a flow of at least $(f^* - |f|)/m$.

Now consider $2m$ successive iterations, where flow is pushed along the augmenting path of maximum capacity. If each augmenting path that is found has value at least $(f^* - |f|)/2m$, then the original flow f will have been augmented by $f^* - |f|$ and the maximum flow will have been found. If the capacity of the maximum capacity augmenting path falls below $(f^* - |f|)/2m$, then the capacity of the maximum capacity augmenting path has decreased by a factor of 2. This means that after each $2m$ iterations, if the maximum flow has not been found then the capacity of the maximum capacity augmenting path decreases by a factor of 2. Since the capacity of the max capacity augmenting path is at most c_{max} and at least 1, there will be at most $2m \log(c_{max})$ iterations.

1.2 Capacity Scaling Algorithm

Augmenting along the maximum capacity path greatly improves upon the number of iterations required. However, it takes more time to find such a path. In the following method, we will augment along a path with large (although perhaps not maximum) capacity. The path will be chosen so that it has sufficiently large capacity to bound the number of iterations in the same manner, but it will require less computation time ($O(m)$) to find the path.

The algorithm will maintain a parameter Δ. Given a network G and a flow f, we define the Δ-residual $G_f(\Delta)$ network to be the residual network G_f with only those arcs whose capacity is at least Δ. Thus, if the capacities are integral and the flow f is integral, then $G_f(1) = G_f$ and $G_f(\Delta)$ is a subgraph of $G_f(1)$.

A phase of the algorithm in which Δ remains fixed is called a *scaling phase*. During each iteration within a scaling phase, the algorithm finds any path from s to t in $G_f(\Delta)$ and augments along that path. This is done until there is no path from s to t in $G_f(\Delta)$. The Δ is halved and a new scaling phase begins. We start with $\Delta = 2^{\lfloor \log c_{max} \rfloor}$ and continue until $\Delta = 1$. Thus, there are $\log(c_{max})$ scaling phases. The code is given in Figure 1.

```
Algorithm CapacityScaling(G, s, t):
(1) f ← null flow
(2) Δ ← 2^⌊lg c_max⌋
(3) while Δ ≥ 1
(4)     while G_f(Δ) contains a path from s to t
(5)         Augment f using the path
(6)     Δ ← Δ/2
```

Figure 1: The capacity scaling algorithm

.

Finding a path from s to t in $G_f(\Delta)$, augmenting the flow and updating $G_f(\Delta)$ takes time $O(m)$ per iteration. The efficiency of the algorithm depends on the fact that there are at most $2m$ iterations per phase.

Let f be the current flow at the end of a scaling phase, and let S denote all the nodes which are reachable in $G_f(\Delta)$ from s. Let $T = S - \{s\}$. Since there is no path from s to t in $G_f(\Delta)$, all the edges from S to T in G_f must have capacity at most Δ. This means that the capacity of the (S, T) cut is at most Δm, since there can be at most m edges from S to T. Thus, the value of the maximum flow in G_f is at most Δm. In the next scaling phase, each augmenting flow will augment the current flow by at least $\Delta/2$, so there can be at most $2m$ such augmentations.

The running time of the capacity scaling algorithm is $O(m^2 \log(c_{max}))$.

1.3 Augmenting Along the Shortest Path

Finding the shortest path from the source to the sink takes only $O(m)$ time using breadth first search. However, the shortest augmenting path algorithm is typically implemented so that it is not necessary to find the shortest path from s to t from scratch every time. This is managed so that the average time to find the shortest path from s to t is $O(n)$. We will not go over the details of this method here although the ideas used are very similar to some of the ideas in the preflow-push algorithm which we will discuss in the next lecture.

However, we will establish that there are $O(nm)$ augmentations before the maximum flow is found, regardless of how the shortest paths are found. The resulting algorithm is $O(nm^2 \log m)$ if the shortest paths are computed from scratch every time and $O(mn^2)$ if the more sophisticated implementation is used. This is the first strongly polynomial algorithm we have seen so far.

The following two lemmas will be useful. For each node u, $d[u]$ will denote the length of the shortest path from u to t.

Lemma 1 *The distance from v to t in the residual graph G_f never decreases as the flow f is augmented.*

Lemma 2 *Between successive times that an edge (u, v) is saturated, the distance $d[u]$ increases by two.*

Since each $d[u]$ is either infinite or less than n, there can be at most n times that the $d[\cdot]$ value of any node increases. In each augmentation, at least one edge gets saturated. This means that there can be at most $mn/2$ saturations, and, thus, at most $mn/2$ augmentations.

We just need to prove the two lemmas.

Proof of Lemma 1. The only way the distance from a node could decrease is with the addition of a new edge. Suppose that the edge (v, u) is added because flow is pushed from u to v along an augmenting path. Let $sp[u]$ be distance from u to t and let $sp[v]$ be the distance from v to t before the edge is added. It must be the case that $sp[u] = sp[v] + 1$ because the edge (u, v) is along a shortest path from s to t. This means that it can not be the case that v finds a shorter path to t through u since $sp[u] > sp[v]$. ∎

Proof of Lemma 2. Suppose that the edge (u, v) is saturated at some time t and is saturated again at some later time t'. It must be the case that at some time t'', where $t < t'' < t'$, flow is pushed back from v to u. Let $sp[x]$ denote the length of the shorest path from x to t at time t. $sp'[\cdot]$ and $sp''[\cdot]$ represent the shorest paths at time t' and t'' respectively. Since the edge (u, v) lies along the shortest path from s to t at time t and t', it must be the case that

$$\begin{aligned} sp[u] &= sp[v] + 1 \\ sp'[u] &= sp'[v] + 1 \end{aligned}$$

The edge (v, u) must lie along the length of the shortest path from s to t at time t''. This means that

$$sp''[v] = sp''[u] + 1$$

We also know that $sp[u] \leq sp''[u] \leq sp'[u]$ and $sp[v] \leq sp''[v] \leq sp'[v]$, since the distances never decrease. Putting it all together we get

$$sp[v] = sp[u] - 1 \leq sp''[u] - 1 \leq sp''[v] - 2 \leq sp'[v] - 2,$$

so we know that $sp[v] + 2 \leq sp'[v]$. ∎