6.854 Advanced Algorithms

Lecture 11: 10/4/2006 Lecturer: David Karger

Scribes: Salman Abolfathe, Jaime Quinonez

Blocking flows

11.1 Blocking flows in unit-capacity graphs

Here, we start with blocking flows in unit-capacity graphs and then generalize for arbitrary capacities. The basic idea is that in order to find a path from s to t, we first augment along those paths that have minimum length. In order to formalize this idea we need some definitions.

Definition 1 *A* **layered graph** *is a graph whose vertices are arranged according to distances from s. The first layer contains vertices whose minimum distance from s is 1, the second layer contains vertices whose minimum distance from s is 2, and so on.*

Definition 2 *An* **admissible arc** *is an arc on a shortest path from s to t. That is, these are the arcs that traverse adjacent layers in a layered graph.*

Definition 3 *An* **admissible path** *is a path from s to t made of only admissible arcs.*

Definition 4 *A* **blocking flow** *is a union of flows along admissible paths that saturate at least one arc on every admissible path.*

The **max flow algorithm** is thus to find a blocking flow, augment along those paths, and repeat until no more augmenting paths are found.

To see the runtime of this algorithm, first establish the following lemma:

Lemma 1 *Augmenting along a complete blocking flow increases the distance between s and t.*

Proof: Suppose there is a path in the residual graph from s to t with the same length as the distance from s to t in the original graph. Then this path is an admissible path in the graph. Thus, a valid blocking flow for the original graph must saturate at least one of its arcs. Therefore, that one saturated arc cannot appear in the residual graph. Thus, there is no such path in residual graph from s to t with the same length as the distance from s to t in the original graph. ■

Using this lemma, since each blocking flow increases the $s-t$ distance by at least one, and the length of each path from s to t is at most n we conclude

Corollary 1 *n blocking flows yield a max flow.*

To find a blocking flow, start at vertex s use the following operations:

Advance: Follow some edge forward from current vertex, adding the traversed edge to current path from s.

Retreat: If there is no outgoing edge from the current vertex, go back along the last edge on the current path and delete that edge.

By applying advance and retreat operations, we eventually reach t. Therefore, the current path becomes an augmenting path for one unit of flow. Augment along this path, then delete all edges on this path from the graph, and start process again at vertex s. Keep applying advance and retreats until all of the edges coming out of s are deleted. You then know that no more flow can be sent out of s, and you have a blocking flow.

In this algorithm, each of the advance, retreat and augment operations is applied at most once per edge. So finding a blocking and augmenting requires $O(m)$ time. By collary 1, n blocking flows suffice to find a max. flow. Therefore, the running time of this max flow algorithm using blocking flows is $O(mn)$.

11.2 Other bounds for blocking flow algorithm in unit-capacity graphs

In the previous section, we used corollary (1) to bound the number of blocking flows required at n. Here we state another bound for the number of blocking flows required to find a max flow, and thus reduce the running time of the max flow algorithm using blocking flows.

Lemma 2 *If s and t are a distance d apart, the residual max flow is at most m/d*

Proof: The residual max flow can be decomposed into paths that must contain at least d edges. Since this is a unit-capacity graph, each edge can contain at most one unit of flow. Since the flow that passes through each path is one unit, each edge is part of at most one path. Thus, there can be at most m/d paths of flow, and since each one contributes one unit to the max flow, the max flow is at most m/d. ■

Now suppose we have already run d blocking flows. By lemma (1), the distance between s and t is at least d. By lemma (2), the maximum residual flow is m/d. Since each blocking flow adds at least one unit to the flow, only m/d more blocking flows are required. Thus, $d + m/d$ total blocking flows will find a max flow. Note that d was an arbitrary parameter introduced for the analysis. By choosing a value of d that minimizes $d + m/d$, we can show a smaller bound on the number of blocking flows required. By balancing and solving $d = m/d$, we get an minimum value at $d = m^{1/2}$, which means $d + m/d = m^{1/2} + \frac{m}{m^{1/2}} = m^{1/2}$ blocking flows suffice to find a max flow. Since each blocking flow takes $O(m)$ time, the total running time for max flow is $O(m^{3/2})$.

This analysis is an example of improving the bounds on the running time of an algorithm by doing a better analysis on the exact same algorithm for max flow on unit capacity graphs. A different analysis can show an upper bound of $O(mn^{2/3})$ on the running time of max flow using blocking flows on a unit-capacity graph. If the graph is a **unit graph**, where every vertex has either in-degree or out-degree equal to one, then one can show that $O(\sqrt{n})$ blocking flows suffice to find a max flow, and thus bipartite matching can be done in $O(m\sqrt{n})$ time.

11.3 Capacitated graphs

To analyze blocking flow algorithms on capacitated graphs, look at how many of three operations, advances, retreats and augments need to be performed.

Since each advance will lead to either a retreat or augment, you can say that the advances are all paid for by the future retreat and augments.

Furthermore, if you retreat along an edge, you destroy it, and thus each blocking flow you can only do at most m retreats.

For augment operations, at least one edge needs to be saturated along each admissible path, so at least one edge gets destroyed per admissible path in the blocking flow. The maximum path length is n (the simple path that goes through all nodes. Thus, for every n augments you do, you destroy at least one edge. Thus, the total augment work per blocking flow is $\leq mn$.

As before, each blocking flow increases the s-t distance by 1, so you only need n blocking flows to find a max flow. This bound leads to a running time of $O(mn^2)$ for max flow using max flows on capacitated graphs.

Note that this is a better time bound than $O(m^2n)$ for shortest augmenting paths max flow. The intuition behind this improvement is that shortest augmenting paths requires searching the graph to find each augmenting path, whereas a blocking flow finds several augmenting paths while searching the graph once per blocking flow.

11.4 Scaling Blocking Flows

In order to scale blocking flows, you want scaling as the outer loop. The algorithm is thus to scale in the capacities bit by bit. After scaling in a bit, run a full blocking flow to find a new max flow. Each blocking flow consists of finding an admissible path, augmenting along it, and repeating until no more augmenting paths can be found, just as before.

For this algorithm you can't say how much any one blocking flow will take. However, you can perform an aggregate analysis. In total, you can compute the max flow over all blocking phases. There are only m augments and each takes n work. Therefore, only $O(mn)$ augment work is required for blocking flows for each bit. Since there are $log\ U$ scaling phases, where U is the maximum capacity of an edge, the total running time of max flow for scaling blocking flows is $O(mn\ log\ U)$

For unit-capacity graphs, Goldberg-Rao showed running times of $O(m^{3/2}\ log\ U)$ and $O(mn^{2/3}\ log\ U)$

for scaling blocking flows to find max flow.

11.5 Improving max flow with data structures

One of the problems with scaling blocking flows is than once you augment along an edge and destroy that edge, you lose an admissible path, and you also lose the partial paths that have been built as the rest of that admissible path. Thus there are many partial paths that get ignored.

One way to view this problem is that there are a collection of trees, they get cut/linked every once in a while, and you want to be able to jump to the head of a tree quickly. You want a data structure that supports the following five operations: link two trees together, cut a tree at a link, jump to the head of a tree, find the minimum edge on a path, and decrease the capacities on all edges in a path. All three of the primitive operations can be expressed using these five tree operations. An advance is to link current node to next node, and then jump to the head of the tree containing the next node. A retreat is to cut a dead node from the previous node. An augment is finding the minimum edge capacity on a path, adding that path, and then cutting the saturated edge.

Sleator and Tarjan invented **dynamic trees**, which accomplish all of the above operations in $O(log\ n)$ amortized time. Using these trees, on can accomplish max flow in $O(mn\ log\ n)$ time.

The best bounds proven so far for scaling max flow are $O(m^{3/2}\ log\ U)$ by Goldberg-Rao and $O(mn\ log_{m/n}\ n)$ by Goldberg and Tarjan. In practice, the push-relable algorithm achieves a time nearly linear in m.

For undirected graphs, Benezur and Karger showed you can approximate the max flow to within a factor of $1-\epsilon$ in $O(\frac{n^{3/2}}{\epsilon^2})$ time. For a unit-capacity graph, Levine and Karger demonstrated a $O(nf)$ time bound.