6.854 Advanced Algorithms

Lecture 14: October 26, 1999 Lecturer: David Karger

Scribes: Brian Dean, John Jannotti

14.1 Minimum Cost Flow Algorithms

14.1.1 Shortest Augmenting Paths (Unit-Capacity Case)

The shortest augmenting path algorithm for solving the minimum cost max flow problem is the natural generalization of the shortest augmenting path algorithm for the max flow problem. To simplify things at first, we'll make a few assumptions:

- All arcs have unit capacity. We'll show later how to scale the algorithm to work in a general network.
- There are no negative-cost cycles. This means the minimum-cost circulation will be zero, but the minimum-cost max flow is still interesting.

Rather than actual arc costs, our algorithm will deal with reduced arc costs. Given a price function $p()$ on the nodes, recall that the reduced cost of an arc (i,j) is $c_p(i,j) = c(i,j) + p(i) - p(j)$. Furthermore, we have seen that if the price function is chosen as the distance from the source node s, then all reduced costs will be non-negative and the reduced costs of shortest paths from s will be zero. When computing shortest paths through the network, it makes no difference if we use the original arc costs or the reduced arc costs.

To initially compute reduced arc costs, we'll need to run a single-source shortest path algorithm from s to compute the initial node prices. Since there may be initially be negative-cost arcs (but fortunately no negative cost cycles), we need to use a shortest path algorithm which accomodates these, such as:

- The $O(mn)$ Bellman-Ford algorithm, or
- A scaling algorithm by Goldberg that runs in $O(m\sqrt{n}\log C)$ time.

After this initial computation, all reduced arc costs will be non-negative. Therefore, subsequent shortest path computations can be performed more efficiently with Dijkstra's algorithm.

The shortest augmenting path algorithm is then to repeatedly compute shortest paths from s and to augment along a shortest path from s to t. Each shortest path calculation using Dijkstra's algorithm takes $O(m\log n)$ time, and since the network is unit capacity there will be at most n augmentations. The total running time is therefore $O(mn\log n)$. If there happen to be parallel arcs in the network, then each shortest path computation will require $O(m^2 \log n)$ time and there will be potentially $O(m)$ augmentations, for a running time of $O(m^3 \log n)$.

Claim: The shortest augmenting path algorithm computes a minimum cost max flow from s to t.

Proof: We maintain as an invariant the fact that there will never be a negative reduced-cost arc in the residual network. Initially, we got rid of all negative reduced-cost arcs by computing shortest paths from s. The shortest augmenting path algorithm then performs two types of operations: recomputing shortest paths in the residual network from s, which doesn't introduce negative reduced-cost arcs, and augmenting along shortest paths from s to t. However, since all arcs in a shortest path from s to t have zero reduced cost, all new residual arcs introduced by this augmentation will also have zero reduced cost. The only way to introduce a negative reduced-cost arc into the residual network is to augment along a positive reduced-cost arc, which is never done. Therefore no negative reduced cost arcs, and therefore no negative reduced-cost cycles, will ever appear in the residual network. When we reach a state when no augmenting path exists, we know that we've found a max flow, and since there will be no negative-cost residual cycles, we know this flow will be a minimum-cost flow. ■

As an extra bonus, the shortest augmenting path algorithm actually computes a minimum-cost flow for every single flow value up to the max flow value.

14.1.2 Finding a Minimum-Cost Circulation

In order to use the previous shortest augmenting path algorithm to find a minimum-cost circulation, we need to satisfy our assumption that there are no negative-cost cycles. This is done by saturating all negative-cost arcs, leaving no negative-cost arcs in the residual network. Unfortunately, this operation potentially introduces excesses and deficits at nodes. However, this problem with excesses and deficits can be solved by creating a source node s connected to all excesses and a sink node t connected to all deficits, and by solving a minimum-cost max flow in the resulting network. We know that a feasible solution exists, since we could just send all excess flow back to the deficit from which it originated. Furthermore, there will be no negative-cost cycles in the residual network, so we can solve the problem using the shortest augmenting path algorithm. Running time will be identical to that of the SAP algorithm.

14.1.3 Minimum-Cost Flow in Unit-Capacity Networks

Given that we know how to find a minimum-cost circulation, how can we find a minimum-cost max flow? Recall that all $s - t$ maximum flows in a network are equivalent to within addition of a circulation. Therefore, we can transform any max flow into a minimum-cost max flow by adding to it a minimum-cost circulation. Hence, we can compute a minimum-cost flow by first finding any max flow and then finding a minimum-cost circulation in its residual network. Running time is still identical to that of the SAP algorithm.

14.1.4 Minimum-Cost Flow in General Networks by Capacity-Scaling

It is possible to adapt the shortest augmenting path algorithm to general-capacity networks by scaling capacities. During each scaling phase, we shift in one bit of each arc capacity, for a total of $O(\log U)$ phases.

In each phase, we shift in at most one unit of capacity on each arc. This gives a unit-capacity problem which can be solved using the preceding techniques. The total running time over all phases will be $O(m^3 \log n \log U)$.

Note that in any phase we may shift in capacity on negative reduced-cost arcs, potentially creating negative reduced-cost cycles. This is not a problem, however, for the preceding methods.

14.1.5 Minimum-Cost Flow in General Networks by Cost-Scaling

An alternative method of solving for a minimum-cost max flow in a general network is by scaling costs rather than capacities. This is useful if capacities are non-integral quantities but costs are integers. In the shortest augmenting path algorithm, we maintained the invariant that there were no negative reduced-cost arcs in the residual network. Let's relax this condition a bit.

Definition : An residual arc (i,j) is said to be ϵ*-optimal* if $c_p(i,j) \geq -\epsilon$. A flow is said to be ϵ-optimal if all its residual arcs are ϵ-optimal.

Initially, we'll start with a zero price function and any max flow, which will be C-optimal. During each scaling phase, we'll go from an ϵ-optimal max flow to an $(\epsilon/2)$-optimal max flow. When can we terminate the algorithm?

Lemma : A $\frac{1}{n+1}$-optimal max flow is optimal; that is, it represents a minimum-cost max flow.

Proof: The reduced cost of any residual cycle in such a network is at least -$\frac{n}{n+1}$. Since the reduced cost of a cycle is the same as the actual cost of the cycle, we know the actual cost of all cycles in the residual network are at least -$\frac{n}{n+1}$, and therefore strictly larger than -1. However, since actual arc costs are integers, this means that all residual cycles must have non-negative cost. ■

We therefore will have $O(\log(nC))$ scaling phases.

To start each scaling phase, we'll saturate all negative-cost residual arcs. This makes all residual arcs have non-negative reduced cost, but introduces excesses and deficits into the network. The capacity scaling algorithm then operates much like the push/re-label algorithm. It will attempt to push excess flow back toward the deficits. Each phase will therefore only circulate flow around, and since we started with a max flow, we'll end up with a max flow.

Definition : An residual arc is *admissible* iff it has negative reduced cost.

During each phase, we'll only do *push* operations along admissible arcs. If there are no admissible arcs (for example, when a phase begins), we do a *relabel* operation. A relabel operation on a node v reduces its price $p(v)$, making its outgoing arcs cheaper. If any outgoing residual arcs drop to a negative reduced cost, they will become admissible. It is always possible to do either a push or a relabel operation, and we'll keep doing these until the excesses flow back to the deficits.

During each phase, we start with a flow that is ϵ-optimal, and we maintain the invariant that our flow is $(\epsilon/2)$-optimal. Since we saturate all negative reduced-cost residual arcs at the beginning of a phase, the invariant will initially hold; we just need to make sure it is maintained under pushes and relabels:

- A push will never cause the invariant to fail. Since we only push along negative reduced-cost residual arcs, a push can only introduce positive reduced-cost residual arcs, which satisfy the invariant.
- A relabel has the potential to break the invariant. We can fix this by reducing a node's label by only $\epsilon/2$ when relabeling it. Since a node is relabeled only if it has no outgoing residual admissible arc, this means that prior to the relabel all outgoing arcs have non-negative reduced costs. Reducing these costs further by no more than $\epsilon/2$ will keep them $(\epsilon/2)$-optimal.

How many push and relabel operations can we do?

Lemma : Each node is relabeled at most $3n$ times.

Proof: At the beginning of a phase, we saturate all negative reduced-cost residual arcs, creating excesses and deficits. By sending flow along a path P, we create not only a deficit at the start of the path and an excess at the end of the path, but also we introduce a residual path P' along which we can send the flow back. Consider any such path P'. Since we started the phase with an ϵ-optimal flow, prior to saturation each arc had cost no less than $-\epsilon$, so each residual arc in P' will have cost no larger than ϵ. The total cost of the path P' will be no more than $n\epsilon$. Each time we relabel a node with some excess (these are the only nodes we ever relabel), we decrease the cost of every such residual path departing from the node by at least $\epsilon/2$. Thus, after $3n$ total relabels of any particular node, the cost of any departing residual path must be less than $-n\epsilon/2$. However, this can't happen if the flow is to remains $(\epsilon/2)$-optimal. We therefore have a bound of $3n$ relabels per node. ■

Given the bound of $O(n)$ relabels/node, the same analysis as with the original push/relabel algorithm applies. There will be $O(mn)$ saturating pushes and $O(n^2m)$ non-saturating pushes, for a total running time of $O(n^2m)$ per phase. The entire scaling algorithm therefore runs in $O(n^2m\log(nC))$ time.

14.1.6 State of the Art

Although we have not seen them, there exists strongly polynomial min-cost flow algorithms.

In 1985 Tardos, using a technique called "minimum mean cost cycles" gave bounds of the form $O(m^2 \textit{ polylog } m)$. The algorithm proceeded by finding the negative cycles in which the average cost per edge was most strongly negative. Thus short cycles of a particular negativity are preferred over long. The algorithm used a cost scaling technique using the ideas of ϵ-optimality, however, after every m negative cycle saturations an edge is "frozen", that is, it's flow value never again changes.

The best known scaling algorithm gives bounds of $O(mn\log\log U\log C)$. It is an open problem to find strongly polynomial bound of the form $O(mn \textit{ polylog } m)$.