MATH 409 LECTURE 11
THE MOORE-BELLMAN-FORD ALGORITHM

REKHA THOMAS

We now study a second algorithm for finding shortest paths in a digraph that works for any edge costs (possibly negative). The digraph will however still have all the assumptions from Lecture 9, which in particular assumes that the graph has no negative cost cycles. This material is taken from [2].

Moore-Bellman-Ford Algorithm

Input: a digraph G without negative cycles, edge costs $c_e \in \mathbb{R}$ for all $e \in E(G)$ and a start vertex $r \in V(G)$.
Output: for all $v \in V(G)$,
— $l(v)$ – the length of a shortest path from r to v
— $p(v)$ – the previous node to v on such a path
If v is not reachable from r, then get $l(v) = \infty$ and $p(v)$ is undefined.

(1) Set $l(r) := 0$ and $l(v) = \infty$ for all $v \in V(G)\backslash\{r\}$.
(2) For $i = 1, \ldots, n-1$ do
  For each $vw \in E(G)$ do:
    if $l(w) > l(v)+c_{vw}$ then set $l(w) = l(v)+c_{vw}$ and $p(w) := v$.

Theorem 1. *The Moore-Bellman-Ford Algorithm runs in $O(nm)$-time.*

Proof. The outer loop takes $O(n)$ time since there are $n-1$ iterations. The inner loop takes $O(m)$ time as it checks the l-value at the head vertex of every edge in the graph. This gives an $O(nm)$-algorithm. (Convince yourself that all the other work done such as comparing $l(w)$ to $l(v) + c_{vw}$ only contribute constant factors to the complexity of the algorithm.) □

Theorem 2. *The Moore-Bellman-Ford algorithm works correctly.*

Proof. At any stage of the algorithm let $R := \{v \in V(G) : l(v) < \infty\}$ and $F := \{xy \in E(G) : x = p(y)\}$. Recall that $l(v) < \infty$ just means that $l(v)$ is finite. Also note that F may not include all the edges of G.

Date: April 22, 2010.

Further, every vertex $x \in R$ has a unique previous vertex $p(x)$ assigned to it by the algorithm even though there maybe many vertices y in $V(G)$ such that $yx \in E(G)$.

We first argue the following:

(a) $l(y) \geq l(x) + c_{xy}$ for all $xy \in F$.
(b) If F contains a circuit C, then C has a negative cost.
(c) If G has no negative cost cycles then R and F together form an **arborescence** rooted at r.

(a) Note that throughout the algorithm, no l-value ever increases. If $xy \in F$ then at some stage of the algorithm, $p(y)$ is set to x and $l(y)$ is set to $l(x) + c_{xy}$. As the algorithm proceeds, $l(x)$ never increases and although $l(y)$ maybe set to $l(x) + c_{xy}$ again, $l(y)$ is always at least $l(x) + c_{xy}$.

(b) Suppose at some stage, a circuit C in F was created by setting $p(y) := x$. Then before that insertion we had $l(y) > l(x) + c_{xy}$ as well as $l(w) \geq l(v) + c_{vw}$ for all $vw \in E(C)\backslash\{xy\}$ by (a). Summing these inequalities as you go around the edges of C, shows that C has negative total cost. More elaborately, if the vertices in C (in order) are $y, x, x_1, x_2, \ldots, x_k$, then we have $l(y) > l(x) + c_{xy}$, $l(x) \geq l(x_1) + c_{x_1x}$, $l(x_1) \geq l(x_2) + c_{x_2x_1}, \ldots, l(x_k) \geq l(y) + c_{yx_k}$. If you sum these inequalities then all the $l(\cdot)$ terms cancel and we get $0 > c_{xy} + c_{x_1x} + c_{x_2x_1} + \ldots + c_{yx_k} = c(C)$.

(c) Since our graph has no negative cost cycles, (b) implies that F has no directed circuits. Further, $x \in R\backslash\{r\}$ implies that $p(x) \in R$ since $x \in R$ implies that $l(x)$ is finite which means that $l(p(x))$ is also finite and $p(x) \in R$. So R and F together form a directed graph rooted at r where every node has precisely one previous node which implies that every node in R has at most one entering edge. This last property implies that the underlying graph of (R, F) is acyclic: We have already ruled out a directed circuit, so if there was a circuit in the undirected graph, then some node in R would have two edges entering it. A connected acyclic digraph (i.e., the underlying undirected graph is connected and acyclic) in which every node has at most one entering edge is called an **arborescence**. So we have shown that (R, F) is an arborescence with root r.

We are now ready to prove the theorem. Before we start, note that $l(x)$ is at least the length of the (r, x)-path in (R, F) for any $x \in R$ at any stage of the algorithm. This is because, $l(x) = l(p(x)) + c_{p(x)x} = l(p(p(x))) + c_{p(p(x))p(x)} + c_{p(x)x} = \ldots =$ the length of the$(r, x) -$ path in (R, F).

We now prove the following statement by induction:

After k iterations of the algorithm, $l(x)$ is at most the length of a shortest (r, x)-path with at most k edges.

Why does this help? Suppose we can prove this. Then this statement will hold for $k = n-1$ and we will have that at the end of the algorithm (which has $n-1$ iterations), $l(x)$ is at most the length of a shortest (r, x)-path with at most $n-1$ edges. However, as we noted before, $l(x)$ is also at least the length of the (r, x)-path in (R, F) at the end of the algorithm and the algorithm gives us an (r, x)-path of length $l(x)$. Putting this together with the observation that no shortest path has more than $n-1$ edges, we will obtain the correctness of the algorithm.

Proof of the statement: First check that the statement is true for $k = 0$ and then assume that its true up to $k-1$. Let P be a shortest (r, x)-path with at most k edges and wx be the last edge of P. Then $P_{[r,w]}$ must be a shortest (r, w)-path with at most $k-1$ edges, and by the induction hypothesis we have $l(w) \leq c(P_{[r,w]})$ after $k-1$ iterations. But in the k-th iteration, wx is also examined, after which $l(x) \leq l(w) + c_{wx} \leq c(P_{[r,w]}) + c_{wx} = c(P)$.

□

Exercise 3. Run the Moore-Bellman-Ford algorithm on the digraph in Exercise 5 of Lecture 10.

References

[1] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics, 1998.

[2] B. Korte and J. Vygen. *Combinatorial Optimization*. Springer, Berlin, 2000.