Design and Analysis of Algorithms

COMP 6006, PG CS, Fall 2005

Prof Rudolf Fleischer

Fudan University, Shanghai

January 3, 2006

Contents

6 Shortest Paths in Graphs **3**
6.1 Definitions 3
6.2 SSSP: The Bellman-Ford Algorithm 5
6.3 SSSP with Positive Edge Lengths: Dijkstra's Algorithm 6
6.4 Car Navigation Systems 7
6.5 APSP: The Floyd-Warshall Algorithm 8
6.6 APSP: Matrix Multiplication 9

Chapter 6

Shortest Paths in Graphs

6.1 Definitions

Definition 6.1.1 (Shortest Path Tree).
Let $G = (V, E)$ be a directed graph. Let $c : E \to \mathbb{R}$ be a *cost function* on the edges. Let s be some fixed node of G (the *source*).

(a) The *length* (or *cost*) $c(p)$ of a path $p = [e_1, \ldots, e_k]$ in G is $c(p) = \sum_{i=1}^{k} c(e_i)$.

(b) For a node v, a *shortest path* $sp(s, v)$ from s to v is a minimum cost path from s to v in G. The *distance* $\mu_s(v)$ (or just $\mu(v)$ if it is clear which node s we mean) of v from s is the length of a shortest path from s to v. If v cannot be reached from s then it has distance $\mu(v) = \infty$. If v can be reached from s on a path containing a *negative cycle* (a cycle with negative cost) then is has distance $\mu(v) = -\infty$.

(c) A tree T in G is a *shortest path tree (SPT)* of G at s if T contains all nodes in G with finite distance from s and all paths from s to a node in T are shortest paths in G. □

Example 6.1.2. Figure 6.1 shows a graph and an SPT at s. Since there can be many different minimum cost paths from s to a node v, there can be many different SPTs; in Figure 6.1 for example, we could replace edge (s, c) by edge (b, c) and obtain another SPT. □

Lemma 6.1.3. *If $p = [v_0, \ldots, v_k]$ is a shortest path from v_0 to v_k then $[v_0, \ldots, v_i]$ is a shortest path from v_0 to v_i for $i = 0, \ldots, k$.*

Proof. Assume there is a shorter path p' from v_0 to v_i than $[v_0, \ldots, v_i]$. Then $p' + [v_{i+1}, \ldots, v_k]$ is a shorter path from v_0 to v_k than p, a contradiction. □

As a corollary we obtain that there is always a simple shortest path if the distance is finite (if the path contains a node twice the cycle between the two occurrences must have total cost zero and can thus be omitted). The lemma also immediately implies that SPTs always exist.

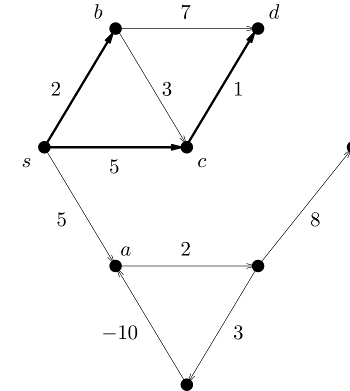

Figure 6.1: The thick edges are an SPT at s. We have $\mu(b) = 2$, $\mu(c) = 5$, and $\mu(d) = 6$. Since a lies on a negative cycle, all the nodes on this cycle and beyond have distance $-\infty$.

Theorem 6.1.4. *Let $G = (V, E)$ be a directed graph with edge costs $c : E \to \mathbb{R}$ and let $s \in V$ be the source. Then there is an SPT of G at s.*

Proof. We choose for any node with finite distance value a shortest path from s and mark the edges of the path. If the subset of marked edges forms a tree, it is an SPT by definition and we are done.

Otherwise, let v be a node with two (or more) incoming marked edges. This means that v was contained in the shortest paths p_1 and p_2 from s to two nodes v_1 and v_2 (one of them could be v), and these two paths reached v on different edges. But then by Lemma 6.1.3 both subpaths of p_1 and p_2 up to v are shortest paths to v, so they must have the same length. Thus, we can choose arbitrarily one of the two subpaths and use it in both p_1 and p_2. This reduces v's indegree (of marked edges) by 1. We do this until all nodes have indegree 1 (of marked edges), and then the subset of marked edges forms a tree. □

The *Single Source Shortest Path Problem (SSSP)* is the problem of computing a SPT for a given graph G and start node s. In the next section we will see the Bellman-Ford algorithm for graphs with arbitrary edge costs. In the important special case of SSSP with positive edge costs, Dijkstra's algorithm can solve the problem much faster. An application are shortest travel route computations in car navigation systems (where Dijkstra's algorithm is still too slow). At the end, we will discuss the Floyd-Warshall algorithm for the *All Pairs Shortest Path Problem (APSP)*, where we want to compute a shortest path between any two nodes of a graph.

6.2 SSSP: The Bellman-Ford Algorithm

Theorem 6.1.4 suggests the following generic algorithm for computing SPTs.

```
(1)   SPT(G, s)
(2)       T = ∅ ;
(3)       while  T is not an SPT  do
(4)           Find e ∈ E such that T + e is a subset of some SPT ;
(5)           T = T + e ;
(6)       od ;
```

So we construct an SPT by growing a tree similar to growing a MST in Prim's algorithm. Since we know that an SPT exists we know that in each iteration of the loop there *exists* an edge we could choose such that $T + e$ remains the subset of some SPT (namely one of the edges in an SPT containing T crossing the cut T). Unfortunately, in contrast to Prim's algorithm there is no local greedy criterium to identify such an edge. Therefore we use a brute-force approach to find such an edge. We reformulate the greedy algorithm as follows.

```
(1)   SPT(G, s)
(2)       d(s) = 0 ;
(3)       π(s) = nil ;
(4)       for all  v ∈ V − s  do
(5)           d(v) = ∞ ;
(6)           π(v) = nil ;
(7)       od ;
(8)       while  there is an edge e = (v, w) such that d(w) > d(v) + c(e)  do
(9)           Choose an arbitrary such edge e = (v, w) ;
(10)          d(w) = d(v) + c(e) ;
(11)          π(w) = e ;
(12)      od ;
```

Lines (10)-(11) are called *relaxation* of edge e. At the end, $d(v)$ should be $\mu(v)$ and $\pi(v)$ is the last edge on a shortest path from s to v, for all nodes v (so we get the edges of a shortest path from s to v in reverse order by simply following the π-links until we reach s). Obviously, at any time $d(v)$ is the length of some path from s to v (with last edge $\pi(v)$ at that time), for all nodes v, so we always have $d(v) \geq \mu(v)$. Since $d(v)$ can only decrease over time (it is only changed in line (11)) we conclude that we never relax an edge (v, w) with $d(w) = \mu(w)$, i.e., once the d-value of a node has reached its correct value (the distance of the node) we never need to consider it again for relaxation. Note that this algorithm runs forever when it hits a negative cycle (and it is not immediately clear that it ever stops if there are no negative cycles).

But if we grow an SPT by always relaxing an SPT edge with one endpoint in the already known SPT subtree, then we would be done after only $n - 1$ relaxation steps. The problem is that we cannot easily identify SPT edges (remember, the SSSP problem is the problem of computing such edges). Therefore,

Bellman and Ford suggested to find the right edge to relax by relaxing *all* edges, i.e., by replacing lines (9)-(11) by something like "relax all edges of G". Then we have relaxed the one SPT edge we were hoping to relax. But we do not know which edge it was because even after relaxing all edges we have no criterium that would tell us which node's d-value reached its final value (the distance of that node). But relaxing the right edge unknowingly is sufficient, so after $n-1$ iterations we have computed an SPT, at least if there are no negative cycles. To deal with negative cycles, we do another iteration of relaxing all edges, and all nodes whose d-value changes must have $\mu(v) = -\infty$; it is then easy to find all other nodes that can be reached from these nodes (these must also have distance $-\infty$).

Theorem 6.2.1. *The Bellman-Ford algorithm computes an SPT of any graph in time $O(nm)$.*

Proof. We have just argued that the algorithm is correct. And the running time is $O(nm)$ because we do n iterations, where each iteration costs time $O(m)$ for relaxing m edges. □

6.3 SSSP with Positive Edge Lengths: Dijkstra's Algorithm

If all edge costs are positive we can easily identify the next edge we should relax because it belongs to an SPT tree. We assume we grow the SPT node by node. Then at any time, we should pick the node not in T with smallest d-value. This algorithm is called *Dijkstra's algorithm*. We realize $Q = V - T$ as a priority queue with the current d-value as priority value (because we always want to find the node v outside of T with smallest d-value).

```
(1)   Dijkstra(G, s)
(2)       d(s) = 0 ;
(3)       π(s) = nil ;
(4)       for all  v ∈ V − s  do
(5)           d(v) = ∞ ;
(6)       od ;
(7)       T = ∅ ;
(8)       Q = V ;
(9)       while  Q ≠ ∅  do
(10)          v = Q.find − min() ;
(11)          Q = Q − v ;
(12)          T = T + v ;
(13)          for all  e = (v, w) ∈ E  do
(14)              if  d(v) + c(e) < d(w)  then
(15)                  d(w) = d(v) + c(e) ;
(16)                  π(w) = e ;
(17)              fi ;
(18)          od ;
```

(19) **od** ;

Theorem 6.3.1. *If all edge costs are positive then Dijkstra's algorithm computes an SPT. The running time is $O(m + n \log n)$ if we use a Fibonacci Heap as the priority queue.*

Proof. We must show that we always choose a node v such that the edge $\pi(v)$ is an SPT edge. This is clearly true for the first node chosen (that is always the node s, with $\pi(s) = \textbf{nil}$. Later, we choose a node v with smallest d-value outside T. Assume this is a bad choice, i.e., the edge $\pi(v)$ is not an SPT edge. Note that $\pi(v)$ connects v to the tree T. But then the shortest path from s to v must cross the cut T at some point, let's say with edge (x, y). But then $\mu(y) < \mu(v)$ (because y comes earlier on the path to v and all edge costs are positive) and $d(y) = \mu(y)$ (x is already in T, so we have relaxed all outgoing edges). Since $d(v) \geq \mu(v)$ we have a contradiction to our choice of v as the node with smallest d-value. Thus, Dijkstra's algorithm computes an SPT.

For the running time observe that we need n *deletemin* operations (one for each node) and m *decprio* operations (one for each edge). □

6.4 Car Navigation Systems

You all know the nice navigation computers you can nowadays find in many luxury cars. Besides showing your current location on a local map, one of their main features is that they compute shortest routes between two points A and B (and instruct you how to follow the route). Unfortunately, these car computers are extremeley slow, in particular their hard disks and CD-ROMS, and they have rather restricted internal memory. Thus, even the fastest shortest path algorithm, i.e., Dijkstra's algorithm, is not feasible in this environment.

Here are two heuristics that can drastically speed up the shortest path computation.

- Dijkstra's algorithm finds shortest paths by slowing growing a disc around the start place A, where shortest paths to all points within the disc are already computed. The algorithm stops if the destination B is contained in the disc. To speed up the computation we can simultaneously grow discs around A and B. If the two discs overlap, we have found the shortest route from A to B.

- Dijkstra's algorithm initially has all distances $d(v)$ set to ∞ (line (5)). Instead, we could initialize the distances with some upper bound on the real distance. Even if the bound is rather bad, the computation can be much faster than starting with the trivial upper bound of ∞.

But even with these heuristics, the shortest path computation may be unacceptably slow. Therefore, they actually use different algorithms, based on precomputed distances. Although it would be possible to preprocess the map and compute the shortest path between any two places, a CD-ROM could not

store all these paths (quadratic blow-up!). However, it is possible to do a preprocessing for the approximate search direction.

Assume, the map is divided into a certain number of regions (could be around 30). For any region R and any other place s outside R, we compute the shortest path from s to any place t in R. If s is sufficiently far from R, then all the shortest paths from s to t will start using the same edge out of s (the next highway in the direction of R, for example). We store this direction at node s. Thus, in each node we store a vector whose components are the edges to be taken first when going to the respective region. All the vectors together need only linear memory.

When searching for a shortest path from s to t, we try to follow the vector entries as long as possible. We will not be able to follow the direction pointers anymore when we reach the vicinity of the region containing t (because they will be undefined). However, if we simultaneously search backwards from t, both search paths will meet somewhere in the middle between s and t. In this case, we have found the shortest path from s to t in time proportional to the number of edges on the shortest path, which is clearly optimal.

6.5 APSP: The Floyd-Warshall Algorithm

In the *All Pairs Shortest Path Problem (APSP)* we are given a directed graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}$. The goal is to compute for any two nodes v and w the shortest path $sp(v, w)$ from v to w. A naive approach to solve the problem would be to solve n independent SSSP problems, one for each node. This would take time $O(n^2 m)$ in the general case of arbitrary edge costs (with the Bellman-Ford algorithm) or $O(n^2 \log n + nm)$ if all edge costs are positive (with Dijkstra's algorithm).

Using dynamic programming, we can derive a better algorithm (at least for the general case). This algorithm is called the *Floyd-Warshall algorithm*, and it was discovered in the early 60's of the last century, the stone age of computer science. Let $V = \{v_1, \ldots, v_n\}$ be an arbitrary numbering of the nodes. For $k = 0, \ldots, n$ let $d_k(v, w)$ denote the length of the shortest path from v to w with all intermediate vertices in $v_1, \ldots, v_k$. Then

$$d_k(i,j) = \begin{cases} c(v_i, v_j) & \text{if} \quad k = 0 \\ \min\{d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)\} & \quad\;\; k \geq 1 \end{cases}$$

because the shortest path using nodes $v_1, \ldots, v_k$ either does not use v_k, or it uses v_k but then we can split this path into two optimal subpaths from v_i to v_k and from v_k to v_j. If $k = 0$ then we can only use the direct edge (v_i, v_j), where we assume that non-existing edges have cost $+\infty$.

So this time we have a 3-dimensional table that we can fill for $k = 0, 1, 2, \ldots$, and for fixed k in any order of the i and j (we only need to look up values with index $k - 1$).

Theorem 6.5.1. *We can solve the APSP problem by dynamic programming in time $O(n^3)$ and space $O(n^2)$.*

Proof. Each table entry can be computed in constant time, and there are $\Theta(n^3)$ table entries. A closer look at the algorithm shows that we actually do not need to store the entire 3-dimensional table. Since we compute it layerwise (for $k = 0, 1, 2, \ldots$) and we only need values from the previous layer, it is sufficient to store two layers at any time (the current layer, and the previously computed layer). So we can reduce the space to $O(n^2)$. □

What happens if the graph contains negative cycles? If k is the highest index of a node on the cycle then in iteration k we will compute a negative value $d_k(i,i)$ for all nodes v_i on the cycle. So we actually still need a post-processing step where we identify all nodes v_i with negative value $d(i,i)$. And then we have to set $d(j,\ell) = -\infty$ for all nodes v_j that can reach v_i and all nodes v_ℓ that can be reached from v_i. But all these nodes can be identified by some graph traversal from v_i in time $O(n)$. Since we might find $O(n^2)$ pairs of nodes (v_j, v_ℓ), the total running time of the post-processing step is also $O(n^3)$.

6.6 APSP: Matrix Multiplication

Another way to solve the APSP is by matrix multiplication. The *distance matrix* M_G of a graph G with n nodes is an $n \times n$-matrix, where the entry $M_G[i,j]$ in row i and column j denotes the length of the edge from node i to node j, or $+\infty$ if the edge does not exists. We now consider a slightly unusual matrix multiplication. In the standard matrix multiplication algorithm, we replace all "+" by "min", and all "·" by "+".

Theorem 6.6.1. *We can solve the APSP problem in time $O(n^3 \log n)$ by computing M_G^n.*

Proof. One way to interpret the entries in M_G is that they represent the length of the shortest path using at most 1 edge between the two nodes. It is easy to see by induction that the entries in M_G^k represent the length of the shortest path using at most k edges between the two nodes. Since no simple path can use more than n edges, M_G^n contains the lengths of the shortest paths between any two nodes.

A single matrix multiplication takes time $O(n^3)$ (note that there are faster algorithms for multiplying matrices with real entries, but these sophisticated algorithms do not work for our modified matrix multiplication). We can compute M_G^n in time $O(n^3 \log n)$ by successive squaring of M_G, i.e., by computing M_G, M_G^2, M_G^4, M_G^8, etc. □

Note that the running time of this algorithm is worse than the running time of the Floyd-Warshall algorithm. However, matrix multiplication can be efficiently implemented on a parallel computer, in contrast to the more sequential Floyd-Warshall algorithm. Thus, on a parallel machine this algorithm can be faster than the other algorithm.