CS 161 W19: Recitation 6 Solutions

February 2019

Exercise 0

We have a network of n nodes and m directed links between them. Each of the n nodes send/forward packets to each other along the m links. The links are not reliable and may fail due to a variety of reasons. Each link between node i and node j is available with probability $p_{i,j}$. We assume that each link fails independently, so the probability that a path, P, is available is simply the product of the probabilities that all of the links in the path are available, $\pi(P) = (p_{1,2})(p_{2,3}) \cdots (p_{k-1,k})$.

We can represent this network with a directed graph G where the edge weights are the probabilities. Give an efficient algorithm for finding the path with the greatest probability of successfully transmitting a packet from node i to node j.

Solution 0

Construct a new graph, G', with the same set of vertices and edges as G. However, we modify the edge weight to be $w'(u,v) = -\log w(u,v) = -\log p_{u,v}$. All edge weights must be non-negative because $0 < p_{u,v} \leq 1$, so $\log p_{u,v} \leq 0$. Therefore we simply use Dijkstra's to find the shortest path from i to j in G', and this corresponds to the highest probability path in G. To see why this is the case, we note:

$$\sum_x w'(v_x, v_{x+1}) = -\sum_x \log p_{v_x, v_{x+1}} = -\log \prod_x p_{v_x, v_{x+1}}$$

To minimize this, we need to maximize the product in the final expression above, which is exactly the problem we wanted to solve.

Exercise 1

Dijkstra's algorithm will in general fail on graphs with negative-weight edges. In this problem, we will explore an interesting special case with negative edges for which we can still solve the shortest paths problem as quickly as Dijkstra's algorithm.

Recall that any directed graph $G = (V, E)$ can be broken up into its *strongly connected components* (SCCs)—in other words, a partition of V into disjoint vertex sets C_i such that within each C_i, any two vertices have a path to one another.

Suppose that in addition to the graph $G = (V, E)$, we are given an *ordered* set of its SCCs, $V = C_1 \cup C_2 \cup \cdots \cup C_k$, with the promise that:

- Within each component C_i, all edges have nonnegative weight.
- Edges between different components may have negative weight. However, they all obey the following rule: for any edge (u, v) such that $u \in C_i$ and $v \in C_j$ for $i \neq j$, we are promised that $i < j$. In other words, an edge out of one component can only point to a component "to the right".

It is helpful to draw some examples of graphs that obey this structure.

Devise an algorithm that, given G, an ordered set $\{C_1, C_2, \ldots, C_k\}$ of its SCCs obeying the above constraints, and a starting node s, finds the shortest path lengths from s to all other nodes $v \in V$. It should have running time $O(|V| \log |V| + |E|)$.

Solution 1

For each $u \in V$, define its *component number* $c(u)$ to be the i such that $u \in C_i$. We modify Dijkstra's algorithm as follows: in each iteration, instead of picking the not-sure node u with the smallest $d[u]$, we pick the not-sure node with the minimum $c(u)$; if there are multiple, then we pick the one with minimum distance estimate $d[u]$ (as Dijkstra's would). The rest of the algorithm remains unchanged.

Correctness

The proof of correctness will follow the same format as the proof for the original version of Dijkstra's algorithm from class. Specifically, the following claims will imply correctness:

0. Sub-paths of shortest paths are shortest paths.
1. For all u, $d[u] \geq d(s,u)$.
2. When u is marked sure, $d[u] = d(s,u)$.

Because $d[u]$ never increases, and the algorithm only terminates when all vertices are sure, claims 1 and 2 together imply that when the algorithm terminates $d[u] = d(s,u)$ for every vertex u.

Claims 0 and 1 hold for the same reasons seen in class. Namely, for claim 0: say you have a shortest-path P with sub-path Q from nodes u to v, where Q is not a shortest $u \to v$ path. That means there exists a shorter $u \to v$ path Q', but then if we replace Q with Q' in path P, we get a shorter path P', which is a contradiction. For claim 1: $d[u]$ is always equal to the length of some path from s to u (or ∞), so it can't be smaller than the length of the shortest path.

For claim 2, we will prove inductively that the t^{th} vertex marked sure has a correct estimate.

Base case. The 1^{st} vertex marked sure is s, and we have $d[s] = 0 = d(s,s)$ when s is marked sure. Even though we have negative edges, they only exist between SCCs, and by definition no path can leave and return to an SCC, so we can't use the negative edges to get $d(s,s) < 0$.

Inductive step. Assume the first $t-1$ vertices have correct estimates when marked sure, and let u be the t^{th} vertex to be marked sure. Suppose for the sake of contradiction that when u is marked sure, $d[u] > d(s,u)$. If there is no path from s to u, i.e., $d(s,u) = \infty$, then this is impossible. So consider a shortest $s \to u$ path P:

$$s \to \cdots \to z \to z' \to \cdots \to u$$

where $s, \ldots, z$ have correct estimates, and $z', \ldots, u$ have wrong overestimates. We claim z must be sure.

If $c(z) = c(u)$, then the sub-path of P from z to u cannot contain any negative edges. Using the fact that the estimate $d[z]$ is correct, claim 0, and the preceding observation, we have $d[z] = d(s,z) \leq d(s,u) < d[u]$. Since $d[z] < d[u]$, and within a component our algorithm always picks the not-sure node with the lowest estimate, this means z must already be sure.

If $c(z) \neq c(u)$ it must be the case that $c(z) < c(u)$ or u would not be reachable from z. Since our algorithm always picks a not-sure node with the minimum component index, z must already be sure.

However, we know that the sub-path of P from s to z' is a shortest path by claim 0. But since z is already sure, we've already updated all of the neighbors of z using the correct estimate $d[z] = d(s,z)$, meaning $d[z'] \leq d[z] + w(z,z') = d(s,z) + w(z,z') = d(s,z')$, contradicting that $d[z']$ is an over-estimate.

Runtime

Compared to Dijkstra's algorithm, the only modification is that we want our heap to compare tuples $(c(u), d[u])$ rather than just finding the minimum $d[u]$. This is an O(1) modification, so we obtain the same runtime as the original Dijkstra, i.e., $O(|V|\log|V| + |E|)$ when using a Fibonacci heap.