CS 6901 (Applied Algorithms) – Lecture 7

Antonina Kolokolova

October 4, 2016

1 Single-source shortest path: Dijkstra's algorithm

Recall the minimum spanning tree algorithm due to Prim-Jarnik:

MST-Prim-Jarnik(G,c,s)
Initialize $s.key = 0$, $s.pred = null$ and $T = \emptyset$
$\forall v \neq s, v.key = \infty; v.pred = null$
Insert all vertices into Q
while $Q \neq \emptyset$
    $u = ExtractMin(Q)$
    $T = T \cup \{(u.pred, u)\}$
    for **each** v adjacent to u
        if $v \in Q$ and $c(u, v) < v.key$ **then**
            $v.key = c(u, v); v.pred = u$
return T

The running time of this algorithm is $O(m \log n)$, as extracting/updating vertices on the priority queue takes time $O(\log n)$ with the standard implementation using heaps (there are ways to get this algorithm to run in $O(m + n \log n)$ using Fibonacci heaps, though).

To prove the correctness of this algorithm, we can use a similar notion of "promising" as before for the loop invariant: the partial spanning tree S_i can be extended to some optimal spanning tree S_{opt} by using edges that do not go between the vertices already in S_i. Then, proceed by induction.

1.1 Dijkstra's algorithm

Now, consider a quite different problem: single-source shortest path. There, given a starting vertex s, we would like to find the shortest paths from s to all other vertices in the graph, that is, paths with the smallest sum of the weights of their edges. We will call such path the shortest path. Equivalently, we will be looking for the distances from s to all other vertices: just the values of sum of weights on a shortest path to a given vertex from s. Also, in this case, we will consider directed graphs.

You already saw one algorithm that computes shortest paths: if there are no weights, BFS started from s gives the shortest paths (in terms of the number of edges) from s to any other vertex in the graph. However,

here weights complicate the matter, as the shortest path can be one with more edges, as long as they have smaller total weight. In the case when the weights are positive numbers (of arbitrary size, given in binary as a part of the input graph description), the classic algorithm to achieve this goal is Dijkstra's algorithm. And it looks very, very similar to Prim-Jarnik's algorithm. Below, the highlighted parts are the changes between Dijkstra's and Prim-Jarnik (in addition to removing the T: all shortest distances are stored as the keys).

ShortestPath-Dijkstra(G,c,s)
Initialize $s.key = 0$, $s.pred = null$
$\forall v \neq s, v.key = \infty; v.pred = null$
Insert all vertices into Q
while $Q \neq \emptyset$
    $u = ExtractMin(Q)$
    for **each** v adjacent to u
        if $v \in Q$ and `u.key+`$c(u,v) < v.key$ **then**
            $v.key =$ `u.key+`$c(u,v); v.pred = u$
return G

As you see, the only change is that the key now stores the shortest distance rather than the smallest edge going to that vertex from the explored part. This change comes up in two places: first in the comparison in the **if** statement, then in the subsequent update of $v.key$.

As Dijkstra's algorithm is a greedy algorithm, its correctness can be proved using the same "promising set" idea for the loop invariant. In this case, though, we will call a partial solution S_i (list of distances) promising if there is an optimal solution S_{opt} which has the same distances as S_i for all vertices already removed from the queue, and distances no greater than those in S_i for vertices still in the queue.

1.2 A^* algorithm

In practice, it is often the case that the size of the solution (in this case, a path) is much smaller than the size of the whole graph. For example, if you have a map of all of Newfoundland and trying to plan a route from MUN to George Street, you would like your algorithm not to consider locations in Gander or Corner Brook.

The A^* algorithm is an extension of Dijkstra's algorithm, which tries to achieve this goal. It extends Dijkstra by incorporating an estimate of the distance to the target into the decision which vertex to consider next. More specifically, rather than using as a key of a node v the distance $v.dist$ from s to v, it sets $v.key = v.dist + h(v)$, where $h(v)$ is the heuristic estimate of the distance to the target. In planning problems, $h(v)$ is often taken to be Euclidean distance from v to the target t ("as the crow flies" distance).

Does A^* algorithm always find an optimal path? That depends on the choice of heuristic. In order to guarantee that when A^* algorithm sees the node t it would have come to it via the shortest possible route, the heuristic has to be *admissible*, which means that $h(v)$ is never larger than the actual distance from v to t. In our implementation, we might need a slightly stronger statement: that for any two nodes v and u, $h(v) \leq h(u) + c(v,u)$. With this condition, you can guarantee that A^* will find the shortest path from s to t, by using a similar proof to the correctness of Dijkstra's or Kruskal's algorithm.