Exercises: Dijkstra’s algorithm

Questions

1. In breadth first search, each vertex has a ‘visited’ field which is set to true before the vertex is put in the queue.   What happens if BFS instead sets the visited field to true when the vertex is removed from the queue? Does the algorithm still work?   Does it run just as fast?

2. Suppose we want to find the shortest distance from s to some particular vertex (rather than to all vertices reachable from s).   e.g. This is the problem that google maps solves. What would we do ?

3. What if we want to find the shortest path between every pair of vertices in the graph ?

4. What is the O( ) runtime of dijkstraEdges( ) from Assignment 2?   Briefly compare it to the O( ) runtime of dijkstraVertices( ).

5. Dijkstra’s algorithm assumes the edges have non-negative weights. (Where does this come up in the proof of correctness?)   But suppose we have a graph with some negative weights, and let edge e be such that cost(e) is the smallest (most negative).   Consider a new graph in which we add cost(e) to all the edge weights, thus making all weights in the new graph non-negative. Now the conditions hold for Dijkstra to find the shortest paths, so we could now run Dijkstra.   Is this a valid way to solve for shortest paths in the case that some edges have negative weights?

6. The problem with Dijkstra’s algorithm for graphs with negative weights is that we might commit to the shortest path to a vertex before we should.   For example, we might find a path to v which is shorter than the path to any other vertex w whose path we don’t know, but in fact there exists a shorter path to v via w such that cost(w,v) < 0.   One way to get solve our problem is to not commit ourselves too early.   Describe a modification of Dijkstra’s algorithm which uses a priority queue in the same way as was presented in class, but which doesn’t commit.   Hint: a vertex could be put into the priority queue multiple times, if its in-degree is greater than 1.   This is similar to the edge-based Dijkstra’s algorithm.

   Suppose we remove a vertex u from the priority queue during Dijkstra’s algorithm.   Then we check all edges of the form (u, v).   We have a current estimate of the shortest cost to v, and if the dist[u] + cost(u, v) < dist[v], then we set dist[v] to its new lower value, dist[u] + cost(u, v), and set parent[v] = u.   Continue until the queue is empty.

   While this algorithm seems to be correct, in fact there is one subtle problem with it.   Can you find it?

Answers

1. The algorithm still works. However, multiple copies of the vertex can be put the queue. The maximum number of copies is the in-degree of the vertex. The size of the queue grows as O(|E|) rather than O(|V|).

2. Run Dijkstra and stop when you reach the desired vertex. Note: I don't claim this is the best approach. It seems very inefficient to consider all possible paths to all possible vertices, when you know the vertex you are looking for. I assume that google maps does something smarter than the solution proposed here. However, the smarter part is probably to solve the problem on a subgraph, where vertices are pruned if they are too far away. For example, if I want to drive from Montreal to the White House in Washington DC, then I can use sort of "triangle inequality" and prune off all locations west of the Mississippi River, since the closest point of the Mississipi river is (I assume) farther from Montreal than all points in Washington DC.

3. One bad approach would be as follows:

   for each s in V { run Dijkstra to find shortest paths from s to all other vertices }.

   The runtime would be O(|V| |E| log |V|). This would be unacceptably inefficient since each pass through the for loop would start fresh. There is a better way to do it which we will see later in the course. The problem is called "all pairs shortest paths" and we will solve it using a general technique called dynamic programming.

4. The runtime is O(|E| log |E|) which is the same as the standard version of Dijkstra, since O(log |E|) is the same as O(log |V |) i..e. |E| is |V|^2 in the worst case. The big difference between the two algorithms is the space requirement, namely the priority queue size is O(|V|) in the vertex-based Dijkstra and O(|E|) in the edge-based Dijkstra.

5. No. For any path in the original graph, the distance of that same path P in the new graph will be greater by cost(e) multiplied by the number of edges in the path P, since we incremented each edges cost by cost(e). But for two paths with a different number of edges, the totals for the extra costs that we have added will be different. So there is no reason why you should expect to get the same solutions in the two cases. [Note that by "same solutions" here, I don't just mean the same distances of the shortest paths; I mean the paths themselves!] For example, take the graph with three edges cost(u,v) = -2, cost(v,w) = 2, cost(u, w)=1. The shortest path to w is (u, v, w). But adding 2 to the cost of each edge and then running Dijkstra would give the shortest path as (u, w).

6. The problem with the algorithm occurs when the graph contains a cycle with negative distance. In that case, the queue will never be empty. The reason is that it can always lower the cost of a path to any vertex in this negative cycle by increasing the number of times it goes around this cycle. (It turns out you can solve this problem by keeping a counter on the number of times that a vertex gets removed from the queue and stopping the algorithm if that number exceeds the number of vertices in the graph. This works because it means that at least one negative cycle has been found. But this makes the algorithm very slow.)