Chris Bohlman

April 3, 2018

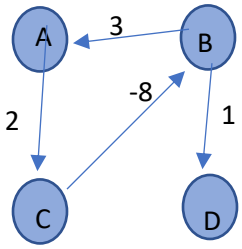
CSc 445

Algorithms Homework #4

1. I have download this information. I have adapted this information. This information is now a part of me.

2. Because the shortest path contains at most k vertices for all vertices, consider the idea of ending the algorithm after k iterations, meaning all vertices will have finished by kth iteration of the for loop. Therefore, once you iterate through the relaxation for all vertices and no values change in array of shortest path values, you can assume that the algorithm is done, and exit. This will be at the kth iteration; therefore, the algorithm runtime will be O(km).

3.



List of edges: (A, C), (C, B), (B, A), (B, D)

d[] = 0 ∞ ∞ ∞

First iteration:

Look at (A, C): d [] = 0 ∞ 2 ∞

Look at (C, B): d [] = 0 -6 2 ∞

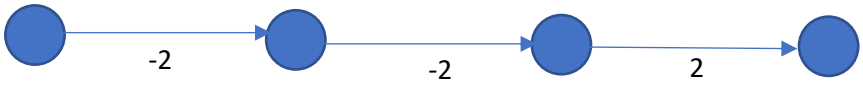
Look at (B, A): d [] = 0 -6 2 ∞

Look at (B, D): d [] = 0 -6 2 -5

Etc.

After iterating through all vertices, we enter for loop at bottom. This ends up triggering the condition for one of the vertices, which means that Bellman-Ford reports that there is a negative cycle in this graph (which there is).

4. Consider a linked list looking graph:



Imagine the first n -2 edges have a negative edge weight, and the last edge has a positive edge weight.

The two stipulations are that when the algorithm terminates, the shortest path at a node should be the correct value in d[v], and at a lower bound of n^2, d[v] should be greater than the shortest path. If the last node is considered node v, when the algorithm terminates, node v will have the shortest path, but before the algorithm terminates (but after Ω(n^2)), the value in d[v] will be larger than the current shortest path.

5. Running time of Dijkstra = Θ(V) * T EXTRACT_MIN + Θ(E) * T DECREASE_KEY
    a. Running Dijkstra with a Θ(n^2) running time:
    You would want to use an array for the priority queue Q, which holds vertices and sorts by vertex with closest d[v] value to current vertex, and hold all of the edges in an array as well:
    EXTRACT_MIN would take O(V) time to copy rest of array
    DECREASE_KEY would take O(1) time
    Together: Θ(V)*O(V) + Θ(E) * O(1) = Θ(n^2) time total
    b. Running Dijkstra with a Θ((n+m)log(n)) running time:
    You would want to use a binary heap for the priority queue Q, a min heap with vertex with smallest d value in relation to current vertex stored at the top of heap
    EXTRACT_MIN would take O(log(V)) time due to height of heap
    DECREASE_KEY would take O(log(V)) time due to height of heap
    Together: Θ(V)*O(log(V)) + Θ(E) * O(log(V)) = Θ((n+m)log(n)) time total
    a. Running bellman ford with O(nm) running time:
    Implicit DECREASE_KEY in relaxation step
    DECREASE KEY could take O(1) time if you stored vertices in an array, with the vertices being sorted by whichever vertex has a d value that is the smallest being closest to current vertex.
    Have to do Θ(E) DECREASE KEY operations, for Θ(V) vertices
    Therefore, running time would be O(EV) = O(nm).

6. Using depth first search means you can step through the topologically sorted graph in the order it's presented to you, relax the edges that are adjacent to the current vertex, and obtain shortest path from any s to any t. Since you're relaxing from the start vertex through every other edge and vertex in worst case time, runtime must be O(n+m) time.

7. In a topologically sorted graph, finding the longest path from a to d must mean you have to find the longest path from a to c, and from a to b before that. Therefore, consider a greedy algorithm that first calculates maximum path to successors of a, and then calculates max path to successors of the successor of b, and so on until you reach node d. The max path is stored for each node, and the relaxation step is reversed so larger values are changed. In essence, this would be the same as finding the shortest distance to a vertex in a FAG, but we're just relaxing longest path values instead of shortest

values. In worst case, this algorithm would go through all vertices and edges, so it would run in O(m+n) time.

8. Find O(m +n) alg for finding shortest path when given edges.

   From the list of edges, consider only the edges that are moving in a forward direction (x2 – x1 > 0). This will give a directed graph. With no cycles, since edges cannot move backwards to return to a previous edge. If there are no cycles and directed edges, then the graph turns out to be Directed Acyclic Graph. From there, we add a vertex with a one-way connection to all blue vertices (but new vertex has no incoming connections), and another vertex that all orange vertices connect to (but new vertex has no outgoing connections). As we proved in class, a topological sort of this graph, as well as finding the shortest path in between these two just added nodes, will allow us to find the shortest path in O(V + E) time = O (n + m) time.

9. When preprocessing, consider street addresses in quadrants. Find shortest path in each cardinal direction of each quadrant (i.e. the northernmost pointof the quadrant, the southernmost point on the quadrant, the , and chain them all together to get shortest path from quadrant to quadrant. Once you reach quadrant of beginning address, try to connect it with the end quadrant. Potentially, make each quadrant a zip code or something, so you can break the problem down into smaller sub problems (zip codes) first.

10. When operating Floyd-Warshall on a graph that has a negative path, sooner or later, you will get a negative on the diagonal. (For any 1,1; 2,2; etc.). This is indicative of a negative cycle involving that point because if there weren't a negative cycle, each vertex's shortest path to **itself** would have to be equal to zero. Therefore, a negative on any element on the diagonal of the n'th iteration in the outer for loop (the final table) indicates that the graph contains a negative cycle.

11. Shortest path of Floyd-Warshall:
    For n vertices, consider an n by n "next" array of pointers to vertices. Initialize this array by setting each element on the diagonal (every i, j vertices where i=j) to be equal to vertex v[i]. For every other edge, set next[i][j] = vertex v[j]. Now, in the for loop, if the relaxation step occurs, set next[i][j] = vertex at next[i][k].

    What this does is create an n x n array of vertices at the end, where if one was looking for shortest path from v[i] to v[j], one would just need to go to next[v[i]], and iterate through array until you reach v[j], at which point you would terminate the algorithm. If you reach null before v[j], there is not shortest path to v[j].