Brigham Michaelis
November 18th 2017
A00364835

Homework 6

Problem 1.

We say that a directed graph G is strongly connected if for any two vertices u and v, there exists a path from u to v and there also exists a path from v to u in G.
Given a directed graph G = (V, E), let n = |V | and m = |E|. Let s be a vertex of G.

(a) Design an O(m + n) time algorithm to determine whether the following is true: for each vertex v of G, there exists a path from s to v. (10 points)

We did this in class, in class we discussed using BFS with colors. The colors are red, white and blue. White means that the node was not visited. Blue means visited but it's adjacent vertices are not checked yet. Red means the node is visited and all of its adjacent nodes are checked. We use BFS with colors and just check for any white nodes at the end. If there are any white nodes that means that they were not visited and so we return false.

(b) Design an O(m + n) time algorithm to determine whether the following is true: for each vertex v of G, there exists a path from v to s. (10 points)

In the previous problem we did BFS with colors. In this problem we are going to first construct an "in-adjacency list" this is like an adjacency list but instead of showing all the nodes that can be reached from any given node, this list will, show all the nodes that are coming into the given node. Now that we have this "in-adjacency" list we simply perform BFS with the color scheme as mentioned above, on this "in-adjacency" list, and check for whites at the end. The same as we did previously. This will run in O(m+n) time because we are just doing BFS with a color scheme. The idea is how to construct the "in-adjacency" list.
This is easy, it is performed with the following psuedo-code:

```
For each vertex u;
     For each vertex v ∈ adj(u);
          add u.value to in-adj(v);
```

And that is all that is needed to construct the in-adjacency list, and it runs in O(m+n) time. So all together this algorithm runs in O(m+n) time and at the end you just check to see if any nodes are white. If they are then return false.

(c) Prove the following statement: G is strongly connected if and only if for each vertex v of G, there is a path in G from s to v and there is a path from v to s. (10 points)

Remark: According to the above statement, we can determine whether G is strongly connected in O(m + n) time by using your algorithms for the above two questions (a) and (b).

This can be proved by using both of the algorithms mentioned in problems 1a and 1b. Namely we would run both algorithms, then only if there are no white nodes at all in either case, then we can prove the graph is strongly connected. Otherwise if there are any white nodes at all then we can prove that the graph is not strongly connected.

Problem 2

Given an undirected graph G = (V, E), let n = |V | and m = |E|. Suppose s and t are two vertices in G. We have already known that we can find a shortest path from s to t by using the breadth-first-search (BFS) algorithm. However, there might be multiple different shortest paths from s to t (e.g., see Fig. 1 as an example).

Design an O(m + n) time algorithm to compute the number of different shortest paths from s to t (your algorithm does not need to list all the shortest paths, but only report their number). (Hint: Modify the BFS algorithm.)

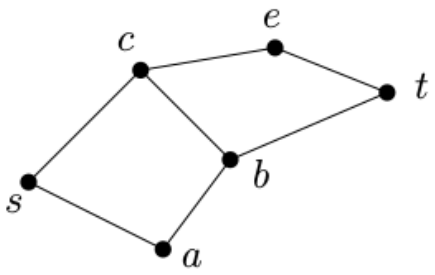

Figure 1: There are three different shortest paths from s to t: (s, a, b, t), (s, c, b, t), (s, c, e, t).

Do BFS as discussed in class…

```
BFS(G,s,t)
{
for each vertex v
        v.color = white;
        v.d = ∞;
        v.pre = null;
s.color = blue;
s.d = 0;
enqueue(Q,s)
while (Q is not empty)
        u = dequeue(q);
        for each v  ∈ adj(u)
                if v.color = white;
                        v.color = blue;
                        enqueue(Q,v);
                        v.d = u.d + 1;
                        v.pre = u;
                u.color = red;
}
```

and modify with the following:

```
// before checking for white:
if(v.color != white) // if we are looking at a node that has already been visited...
        if(v.d == u.d+1) // if distance is equivalent to the previous node's distance plus one...
                v.possible = v.possible + u.possible; // increment the nodes .possible value by itself and
                                                      //the previous node's .possible value.
```

// You would have to initialize all of the v.possibles first to zero. But when doing the final check make sure to use white checking. And initialize v.possible for s to 1

Also in the first if block where we are checking for white we need to set the v.possible to u.possible.

This simple modification to BFS will generate all the possible shortest paths there are in BFS runtime, aka O(m+n). The idea is to do BFS, but we are still going to consider things that were already visited.

Problem 3
Given a directed-acyclic-graph (DAG) G = (V, E), let n = |V | and m = |E|. Let s and t be two vertices of G. There might be multiple different paths (not necessarily shortest paths) from s to t (e.g., see Fig. 2 for an example).

Design an O(m + n) time algorithm to compute
the number of different paths in G from s to t. (20 points)

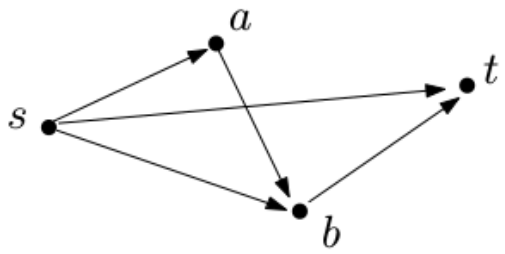

Figure 2: There are three different paths from s to t: s → t, s → b → t, and s → a → b → t.

The first and most important step is to do a topological ordering. And check for cycles.

The variable that we are interested in is node.possible, which indicates the number of paths from s to that node. For reference we are going to set s.possible = 1;
Now that that part is out of the way we continue with the algorithm.

We are going modifiy Single Source Shortest Path that was given in class.
We have a topological ordering of the nodes in the adjacency list, now we can do something like...

```
SSSPmod(G,s,t)
{
for each vertex v
        v.possible = 0;
s.possible = 1;
for each vertex u following the topological order and starting at s and going to t.
        for each vertex v in adj(u)
                v.possible = u.possible + v.possible;
}
```

This algorithm with run in O(m+n) time because it is a simple modification from Single Source Shortest Path which was given in class with a runtime of O(m+n), and when your are finished running you simply output t.possible.

We are only working from s to t in the graph, and the idea is that we are carrying the possibilities with us as we step through the topological ordering.

Problem 4

In class we have studied an algorithm for computing shortest paths in DAGs. Particularly, the length of a path is defined to be the total sum of the weights of all edges in the path. Sometimes we are not only interested in a shortest path but also care about the number of edges of the path. For example, consider a flight graph in which each vertex is an airport and each edge represents a flight segment from one airport to another, and the weight of every edge represents the price of that flight segment. If you want to fly from airport s to another one t, you are certainly interested in a shortest path from s to t to minimize the total price you have to pay. But you probably do not want a path with too many stops (i.e., too many edges) even if it is relatively cheap. For example, suppose you can only accept a path with at most two stops (i.e., three edges). Then, your goal is essentially to find a shortest path from s to t such that the path has at most three edges. This is the problem we are considering in this exercise.

Let G = (V, E) be a DAG (directed-acyclic-graph) of n vertices and m edges. Each edge *(u, v)* of E has a weight *w(u, v)* (which can be positive, zero, or negative). Let k be a positive integer, which is given in the input. **A path in G is called a k-link path if the path has no more than k edges.** Let s and t be two vertices of G. **A k-link shortest path from s to t is defined as a k-link path from s to t that has the minimum total sum of edge weights among all possible k-link s-to-t paths in G.** Refer to Fig. 3 for an example.

Design an O(k(m + n)) time algorithm to compute a k-link shortest path from s to t.

For simplicity, you only need to return the length of the path and do not need to report the actual path. (Hint: use dynamic programming or modify the shortest path algorithm on DAG we discussed in class.) (20 points)

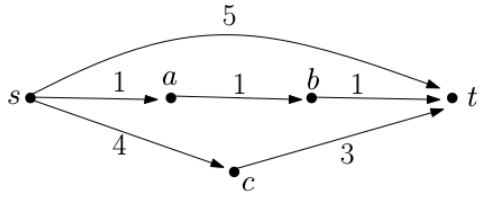

Figure 3: Illustrating a DAG: the number besides each edge is the weight of the edge. If we are looking for a 2-link (i.e., $k = 2$) shortest path from s to t, then it is the path consisting of the only edge (s, t), whose length is 5. Although the path $s \to a \to b \to t$ is shorter, it is not a 2-link path since it has three edges. Note that the path with the only edge (s, t) is a 2-link path because it has one edge, which is indeed **no more than** two edges.

This can be done by modifying BFS to check all possibilities as it spreads. In this fashion we start at s and spread outward, each node will keep track of its k value(the level at which it was found) and its weight. We will check and overwrite previous values only if they are smaller and we have not exceeded the k_value that was given as input to the program.

The psuedo code with comments is given below.

We include the Psuedo code with comments to describe the algortihm.

```
K-LinkPath(G,s,t,k_value)
{
  k = 1;              // We initialize our k to 1.
  for each vertex v   // Set the v.d = infinity and v.k to -1
        v.d = ∞;
        v.k = -1;
  s.d = 0;            // Start with s.d = 0, it costs us nothing to start here
  Q;                  // here we initialize the queue
  enqueue(Q, adj(s)); // Here we are placing the contents of the adjacency list of s onto the queue
  while(Q !Empty)
  {
        u = dequeue(Q); // dequeue an element, if u.k is greater than the current value of k we know
                        // that we have finished dequeuing all the items of the previous level.
        if(u.k > k)     // increment our k
               k += 1;
        if(k > k_value)
               terminate and return t.d; // if our k is too big we terminate and return t.d
        for each (vertex v ϵ adj(u))
               v.k = k; // set the k value for the vertex
               if(v.d > u.d + w(u,v))
               v.d = u.d + w(u,v); // if the distance is smaller than the previous value, store it and
                                   // update the solution space.
               store(v,v.d,v.k);   // store the solution, the idea is that this will store the smallest
                                   // distance for every node, t included. Then when we are finished we
                                   // will have the smallest distance for t which will be correctly
                                   // updated until our k-value is too large. At which point we break
                                   //out and return t.d
               for each (vertex r ϵ adj(v))
                      r.k = v.k + 1; // otherwise update the k values of the next pieces and enqueue
                      enqueue(Q,r);
  }
}
```

This algorithm will run in O(k(m+n)) time. Our algorithm is basically doing (m+n) work, k times. In other words we start at s and spread outwards in a BFS fashion. We keep track of what k-levels various nodes are found at and we also keep track of their minimum. And we are checking nodes that may have been previously picked up into a solution set and overwrite their values if appropriate.