CSE 413 Homework 7 Solutions

1)

The general idea is to follow the standard approach for a shortest path, but to consider vertices instead of edges. A heap to find the next smallest path edge based on edges can then be avoided.

Start with the source vertex s. Initialize an array A of size |V| with the current shortest path lengths from the source to all vertices, and a flag to indicate if each path is a final path. Initialize the path lengths to all vertices except for s to ∞ and the path length to s to zero. The path to each vertex is not final, including the vertex s.

Perform the following operation. Scan the array A and select the vertex v with minimum path length that is not final and is less than ∞. Mark the path to that vertex as final. Consider all the neighbors of v which are reachable in one step from v (in other words, the neighbors with an out edge from v). For each of these neighbors of v, consider the length of the path to v, plus the weight of the edge from v to each of its neighbors. Scan the shortest path array, and for each of these neighbors update the length if the new length is smaller. Continue doing this until all vertices are final. If the actual shortest path is desired, also store the last vertex along the path to each vertex in the array. This allows the path of each to be traced back.

There are |V| iterations as each iteration finalizes the path to a single vertex, and continues until the path to all vertices are final, or there are no more selectable vertices (as in a disconnected graph). Each iteration takes a scan in O(|V|) time of the vertex array, a maximum of another scan of the neighbors of the new node, and a final update of the vertex array. Thus, each iteration takes O(|V|) time. This gives $O(|V|^2)$ time.

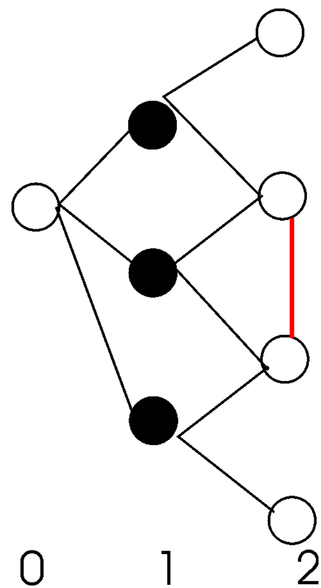
Pseudocode

```
        double costs[|V|];
        boolean final[|V|];
        Vertex lastVertices[|V|];
        int total_final =0;

        for(int i=0; i<|V|;i++)
        {
                costs[i] = ∞;
                final[i] = false;
                lastVertices[i] = Null;
        }

        costs[Start] = 0;

while(total_final<|V|)
{
        double min_cost = ∞;
```

```
		double min_index = -1;
		for(int i=0; i<|V|;i++)
		{
			if(costs[i]<min_cost && cost[i]< ∞ && !final[i])
			{
				min_cost = costs[i];
				min_index = i;
			}
		}

		if(min_index==-1)//no selectable vertices
		{
			break;
		}

		final[min_index] = true;
		total_final++;

		for(each edge e in adjacency_list[min_index])
		{
			if(costs[min_index] + e.weight < costs[e.destination_vertex])
			{
				costs[e.destination_vertex] = costs[min_index]+e.weight;
				lastVertices[e.destination_vertex] = min_index;
			}
		}
	}

	ShortestPath = {};
	Node current = path_destination;
	while(current!=start && current!=null)
	{
		shortestPath = current + ShortestPath;
		current = lastVertices[current];
	}
	ShortestPath = source + ShortestPath;
	return ShortestPath, costs[path_destination]
```

2)

Conceptually this algorithm is like running a breadth first search from a starting node s. Each node is on a level of the breadth first search corresponding to the distance of that node from the starting node s. The node s is on level 0, as a path of length 0 can reach only 0. All nodes adjacent to s are on level 1, and so forth. The longer the path required to reach a node, the higher the level of that node. See picture below for an illustration of this. Start coloring the first node one color, and then alternate the color on each level. So, all nodes with an even level are color 0, and all nodes with an odd level are color 1 in a fully colored graph. There is a problem if a node must be colored one color, and it is already colored the alternate color. An edge from a node will never connect back to a node on a lower level of the BFS, or the node it connects to would already have been discovered at an earlier level of the BFS. So, the only potential conflict is if two nodes on the same level are connected, like with the red edge in the illustration. If this occurs, there is a node that is adjacent to a node of both colors, and the graph is not colorable in two colors.



Specifically, initialize the color of all nodes as null. For simplicity, let the two colors be color 0 and color 1. Select an arbitrary node s in the graph. Color s color 0. Place any node that is adjacent to s (has an edge from s to that node) into a Queue Q_1. Every node in Q_1 is removed and colored as color 1. If any node v in Q_1 is already colored as color 0, then the graph is not colorable. This is because the node v is adjacent to both a color 0 and color 1 node, because it would only be colored as color 0 by being adjacent to a color 1 node. As each node is removed from Q_1, add each of its neighbors to a second Queue, Q_2. After Q_1 has been emptied, each node in Q_2 is removed and colored as color 0, again checking for conflicts with nodes that have already been colored, as described above. Any node adjacent to a node in Q_2 is placed in Q_1, to be colored the appropriate color. Note that it is possible to place a colored node in the queue, and this is necessary to check for conflicts. If the node is colored the appropriate color, no damage is done – if it is colored the alternate color, this is the conflict that results in an uncolorable

graph. This is repeated, alternating queues and colors, until no nodes remain to be added, or the graph is found to be uncolorable.

This algorithm is implemented with the same idea of BFS in 7.13 of Manber. Specifically, instead of just one queue, two queues will be used. In this way, one level of the BFS is handled, then the next level and so forth. A level of nodes are added to one queue, and then these nodes are colored as color c. Whenever a node v is colored, check all the neighbors of v and make sure a neighbor is not also colored c. If so, the graph is not colorable because v and its neighbor are both colored c. If there are no conflicts, add the neighbors of v to the other queue, as these neighbors are on the next level. Alternate colors and repeat until all nodes are colored, or a conflict is found.

This is a modification of breadth first search, and the same complexity argument applies, giving O(|V| + |E|) time.

Pseudocode – based on 7.13, page 199 of Manber

```
Queue q_array[2];//q_array[0] = red nodes, q_array[1] = blue nodes
which_q = 0;
mark v;
put v in q_array[which_q];

while(q_array[which_q] is not empty)
{
        remove the first vertex w from q_array[which_q];
        w.color = which_q;
        for(all edges (w,x))
        {
                //some vertex adjacent to w is already colored
                if(x.color != null && x.color==w.color)
                        return "Graph is not colorable";
                if(x.marked == false)
                {
                        mark x;
                        q_array[NOT which_q].add(x);
                }
        }
        if(q_array[which_q] is empty)
                which_q = NOT which_q;
}

return "Graph is Colorable";
```

3)

a) yes
b) yes
c) no

This algorithm looks for a longest weight path in a DAG. To do this, create a source vertex and a sink vertex. Connect the source vertex with out going edges to each vertex in the graph, with each edge having a cost of 0. Connect the sink vertex with incoming edges from each vertex in the graph, again with cost 0. Find the longest path from the source to the sink. To do this, modify the acyclic shortest path algorithm, found on page 203, chapter 7.15 of Manber to find the longest path instead of the shortest path.

A graph can have negative edge weights, the algorithm will work the same regardless, because it is acyclic and cannot have a negative cycle. However, if the graph is cyclic this becomes an instance of the longest path problem, which is an NP-complete problem and so the algorithm does not generalize to the cyclic case.

This algorithm requires an O(|V|) time insertion of a sink and source vertex to a directed acyclic graph, because both the sink and source node are connected to O(|V|) edges. Then, a minor modification is made to the acyclic shortest path algorithm, so that it will find the longest path instead of the shortest path. This algorithm then runs in the same time as the acyclic shortest path algorithm, running in O(|V|+|E|) time. This gives a total of O(|V|+|E|) time for this algorithm.

Pseudocode

```
Acyclic_Longest_Paths();//This is the same as 7.15 in manber, but find the
                        //longest path instead of shortest, in other words, change
                   //w.SP+length(w,z)<z.SP to w.SP+cost(w,z)>z.SP and change
                   //length to weights since it is weights instead of # of edges

old_vertex_count = |V|;
Vertex Source, Sink;
Insert Source, Sink into end of adjacency list.
for(int i=0; i<old_vertex_count;i++)
{
        adjacency[Source].insert((source, i, 0));
        adjacency[Sink].insert((i,sink,0));
}
//keep track of pointers to get the path if desired, and trim off source
//and sink before returning the path
return Acyclic_longest_paths(G,source, old_vertex_count+2);
```

4)

To find the minimum bottleneck spanning tree, the MCST algorithm from the textbook can be used. This is presented in chapter 7.20 on page 211 of Manber. This runs in O((|V|+|E|)log |V|) time.

The standard MCST algorithm can be used for the bottleneck spanning tree problem because the MCST algorithm finds the minimum weight edge at each step, and only adds it to the tree when it is the best option. If the minimum weight edge that helps solve the problem is added at each step, then the largest edge in the graph was the minimal edge at the time it was added. Therefore, the MCST will also produce the minimum bottleneck weight spanning tree.

However, it is possible to find the minimum bottleneck spanning tree in a faster time. To do this, find the median weight edge of the edges and remove all the edges larger than this edge. If the graph is still connected, then the problem is solved recursively on the graph without the removed edges. However, if the graph is not connected, then there are connected components left. Shrink each connected component down to a single vertex. Edges that are internal to the connected component point back to this vertex and are not useful, and edges between different components become edges in the new graph and the problem is solved recursively on the new graph. This procedure can then be repeated until the desired result is achieved. This can solve the problem in O(|E|+|V|log|V|) time because there are O(log |V|) recursions, with |V| vertices at each level. The number of edges is reduced by half in each iteration, and so the |E| term must be accounted for in the time required.

However, for this problem, using the MCST algorithm to find the minimum bottleneck weight spanning tree is sufficient for full credit.