Brendan Bruce

ECE 573: HW4

Due Date: 4/30/2019

```
In [1]: from experiments import *
```

Question 1

Q1. Write a program that answers the following for an undirected graph: Is a graph acyclic? Run your program on the graph provided.

Discussion

Two approaches came to mind for finding cycles in a graph: Use a DFS or find if there are connected components.

In the DFS approach (the one I have implemented), we would perform a DFS and if a node that has been visited is visited again then it must be part of a cycle.

In the connected components approach we use the fact that if there are connected components then there must be cycles. However, the method I would have used to implement this would have been the double DFS approach so I figured, because we just want to know IF there are cycles and not WHAT are the cycles, then I would be better off using the simple DFS approach.

Results

```
In [2]: q1()
```

```
250 vertices, 1273 edges
Running modified DFS to look for cycles
Does graph contain cycles? True
```

Question 2

Q2. Implement and execute Prim's and Kruskal's algorithms on the graph linked below (the third field is the weight of an edge). Which performs better? Explain your answer.

Discussion

After implementing both algorithms to find the MST, we find that Kruskals completes much faster than Prims. Because it is the same input data, we must conclude that Kruskals outperforms Prims experimentally.

Doing some research on the topic, it appears that implementation of the algorithms dictate their performance. Specifically, Prim's will vary from O(|V|^2) to O(|E|+|V|log|V|) where V is num vertices and E is num edges depending on if it is made using an adjacency matrix or an adjacency list. In the implementation used for the below results, Prim's uses an adjacency matrix which explains the longer runtime.

Results

From running each algorithm on the linked graph we get the following runtimes:

Kruskals completes in 0.014340002616892611 seconds

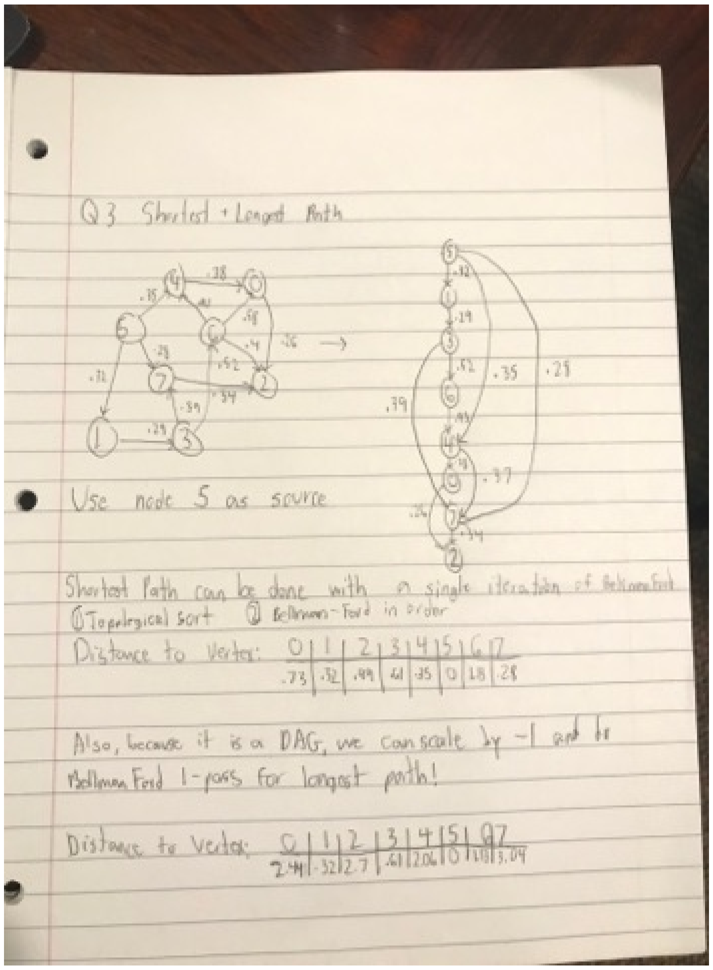
Prims completes in 0.31722370449593523 seconds

Question 3

Q3. For the edge-weighted directed acyclic graph given below, compute (i.e., manually trace) both the longest path and the shortest path.

8
13
5 4 0.35
4 7 0.37
5 7 0.28
5 1 0.32
4 0 0.38
0 2 0.26
3 7 0.39
1 3 0.29
7 2 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93

Manual Trace



Q3 Shortest + Longest Path

Use node 5 as source

Shortest Path can be done with a single iteration of BellmanFord
① Topological sort ② Bellman-Ford in order

Distance to Vertex:

0	1	2	3	4	5	6	7
.73	.32	.49	.61	.35	0	1.8	.28

Also, because it is a DAG, we can scale by −1 and do Bellman Ford 1-pass for longest path!

Distance to Vertex:

0	1	2	3	4	5	6	7
2.44	.32	2.7	.61	2.06	0	1.13	3.04

Discussion

To reduce the amount of work I needed to do I used two special properties and also used Bellman-Ford, the algorithm I had already traced for question 4.

The first property is that a DAG can be topologically sorted and that if we do Bellman Ford in topological order, then we will find the shortest path in a single iteration of the algorithm.

The second property is that if we make all the weights in the graph negative, then the shortest path will then be the longest path after negating all of the results.

With these two properties I was able to produce the trace in minimal calculations.

For topological sorting I plotted the graphs on an app that let me drag and drop nodes and I played around with it until they were in topological order.

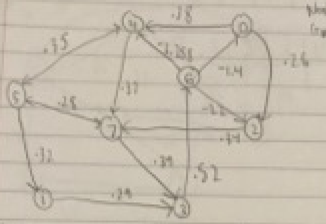
Question 4a

Q4. (a) For the digraph with negative weights, compute (i.e. manually trace) the progress of the Bellman-Ford Algorithm.

```
8
15
4 5 0.35
5 4 0.35
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32
0 4 0.38
0 2 0.26
7 3 0.39
1 3 0.29
2 7 0.34
6 2 -1.20
3 6 0.52
6 0 -1.40
6 4 -1.25
```

Manual Trace

Q4 a. Manual Bellman Ford Algorithm



~~Node, edge list in input~~
~~Graph itself didn't help much~~

On each iter. we look at all edges, if taking the edge can reduce path length we update. Repeat |V|-1 times because the longest shortest path can be is |V|-1 edges away.

- Using node 0 as the source

Iteration 0:

0	1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞	∞

1:

0	1	2	3	4	5	6	7

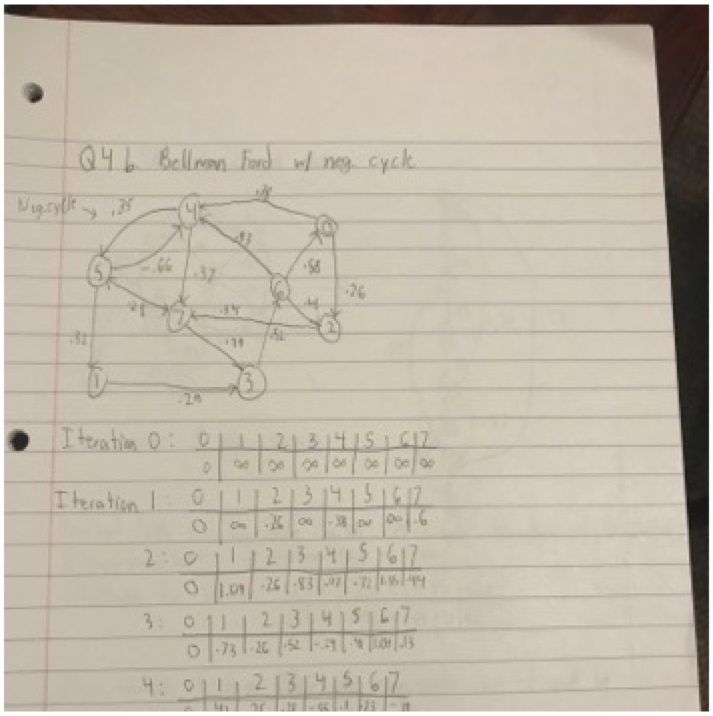
// updates starting at 0

Question 4b

Q4. (b) For the digraph with a negative cycle, compute (i.e. manually trace) the progress of the Bellman-Ford Algorithm.

```
8
15
4 5 0.35
5 4 -0.66
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32
0 4 0.38
0 2 0.26
7 3 0.39
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93
```

Manual Trace



Q4 b. Bellman Ford w/ neg. cycle

Iteration 0:

0	1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞	∞

Iteration 1:

0	1	2	3	4	5	6	7
0	∞	.26	∞	.38	∞	∞	.6

2:

0	1	2	3	4	5	6	7
0	1.09	.26	.83	1.07	.72	1.15	.44

3:

0	1	2	3	4	5	6	7
0	.73	.26	.52	.29	.4	1.01	.23

4:

0	1	2	3	4	5	6	7
[illegible]	.41	.26	.11	.55	.1	.23	.11

Question 5

Q5. Implement a DFS and BFS traversal for the data-set of the undirected road network of New York City. The graph contains 264346 vertices and 733846 edges. It is connected, contains parallel edges, but no self-loops. The edge weights are travel times and are strictly positive.

Discussion

I ended up working with two DFS implementations for this problem. The first implementation was recursive, this caused issues due to hitting the limit of the recursion stack and crashing my program. I did not realize this until trying to run on the NYC dataset, which was sufficiently big to break the program.

The BFS was an iterative approach that adds each neighbor to a queue and then visits the next unvisited item in the queue.

After learning that the DFS would break I modified it to mirror the BFS. We learned in class that the key difference in BFS vs. DFS is that BFS uses a queue and DFS uses a stack. Therefore, I copied pasted BFS and changed it to use a stack to get a DFS.

The DFS then became an iterative approach that adds each neighbor to a stack and then visits the next unvisited item in the stack.

Results

The traversals are very long so I did not include the print of each.

Question 6

Q6. Implement the shortest path using Djikstra's Algorithm for the graph in HW5 Q 4(b). Then run your implementation of Djikstra's on HW5 4(a). What happens? Explain.

Discussion

Dijkstra's algorithm is a greedy algorithm, it selects the shortest edge weight at every opportunity to do so. The issue arises when we have graphs with negative edge weights. The shortest path produced by Dijkstra is not guarenteed to account for these negative edges and therefore won't be shortest paths but rather short paths.

An analogy would be if you were offered to buy a 10 dollar discount for 5 dollars. Dijkstra would reject the discount while Bellman-Ford would try the discount.

Results

In our results we see that the outputs from Dijkstra differ from our manual trace of Bellman-Ford.

```
In [4]: q6()
```

```
Running Dijkstra on graph from Q4a (negative weights)
Vertex   Distance from Source
0                0.00
1                1.05
2                0.26
3                0.99
4                0.38
5                0.73
6                1.51
7                0.60
Running Dijkstra on graph from Q4b (negative cycle)
Vertex   Distance from Source
0                0.00
1                1.05
2                0.26
3                0.99
4                0.38
5                0.73
6                1.51
7                0.60
```