CMSC351 Spring 2025 (§0101,§0201,§0301) Homework 9

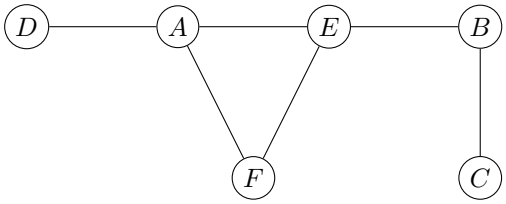
Due Thursday Apr 24, 2025 by 23:59 on Gradescope.

Directions:

- Homework must be done on printouts of these sheets and then scanned properly, or via latex, or by downloading, writing on the PDF, and uploading.
- Do not use your own blank paper!
- The reason for this is that gradescope will be following this template to locate the answers to the problems so if your answers are organized differently they will not be recognized.
- Tagging is automatic, do not manually tag.

---

1. Consider this graph:



Do not apply any particular pseudocode, rather:

- BFT: After the starting vertex visit all nodes distance 1 away, then all nodes distance 2 away, and so on.
- DFT: After the starting vertex go as deep as possible and back up only when you have to and only as far as is necessary.
- In both cases when a decision needs to be made use alphabetic priority.

Suppose BFT and DFT each start at some vertex x (unknown) which is considered the first vertex visited.

Question	Answer	
For which x would D be the second vertex they both visit?	A	[5 pts]
For which x would B be the third vertex they both visit?	NONE	[5 pts]
For which x would A be the fourth vertex they both visit?	B,C	[5 pts]

Put scratch work below. Scratch work is not graded but note that you should know how to explain your answer because you may be expected to do this on an exam.

2. Consider this implementation of BFT with a `print` statement added:

```
function bft(G,s)
    QUEUE = [s]
    VISITED = list of FALSE of length V
    VISITED[s] = TRUE
    VORDER = [s]
    while QUEUE is not empty:
        x = QUEUE.dequeue
        for all y adjacent to x:
            print("HELLO!")
            if VISITED[y] == FALSE:
                QUEUE.enqueue(y)
                VISITED[y] = TRUE
                VORDER.append(y)
            end if
        end for
    end while
    return(VORDER)
end function
```

(a) Looking at all graphs G which are simple, connected, unweighted, and undirected with V vertices. In terms of V, what is the maximum number of times that `HELLO!` could be printed ... [8 pts]

... if G is stored as an adjacency matrix?	$V(V-1)$
... if G is stored as an adjacency list?	$V(V-1)$

(b) Suppose G is a tree with V vertices. In terms of V, if G is stored as an adjacency list, exactly how many times will `HELLO!` be printed? [4 pts]

How many times?	$2(V-1)$

(c) Looking at all graphs G which are simple, connected, unweighted, and undirected with V vertices. In terms of V, what is the maximum number of elements that `QUEUE` could possibly contain during the code execution? [4 pts]

Maximum number?	$V-1$

(d) Briefly explain your answer to (c). [4 pts]

Solution:

It will start with s and then if s is connected to every other vertex then next it will have $V-1$ elements. That is the longest it will get.

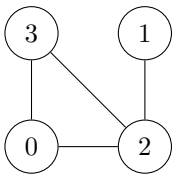
3. Consider this pseudocode for DFT:

```
VORDER = [];STACK = [s]
VISITED = list of length V full of FALSE
while STACK is not empty:
    x = STACK.pop()
    if VISITED[x] == FALSE :
        VISITED[x] = TRUE
        VORDER.append(x)
        for all nodes y adjacent to x:
            if VISITED[y] == FALSE: STACK.push(y)
        end for
    end if
end while
```

Suppose G is a graph with V vertices with every vertex adjacent to every other vertex. How many times will the `while` loop run? Simplify. [5 pts]

How many times?	$1 + \frac{V(V-1)}{2}$

4. Suppose we try to do DFT using a stack and such that when we pop a vertex we mark it as visited and then push each adjacent vertex onto the stack provided it has not been visited and isn't already on the stack. Moreover adjacent vertices are pushed in increasing numerical order.



(a) Starting with vertex 2 In the following table fill in the stack (comma separate) and `VORDER` or `DORDER` in Ting's slides after each iteration. You won't need to fill in all of the boxes. [10 pts]

Iter #	STACK	VORDER			
Before	2				
1	0,1,3	2			
2	0,1	2	3		
3	0	2	3	1	
4		2	3	1	0

(b) Is this a DFT? Explain in a sentence or two. [5 pts]

Solution:

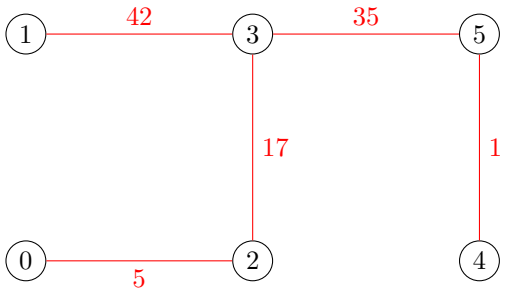
No, because after visiting 2 and then 3 we should go to 0, not 1.

5. Suppose we run Dijkstra's Algorithm on a graph with 6 vertices. We get the following final `PRED` and `DIST`. [10 pts]

Index	0	1	2	3	4	5
`DIST`	5	59	0	17	53	52
`PRED`	2	3	N	2	5	3

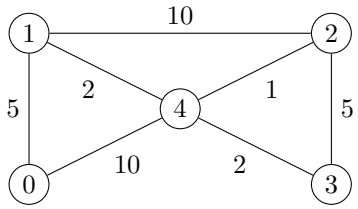
Use this to fill in the edges and weights for the corresponding shortest path tree below.

Note: You are not completing the graph, just the tree!



Put scratch work below. Scratch work is not graded but may be used for regrade partial credit.

6. Consider the following graph: [20 pts]



We apply Dijkstra's algorithm starting at verrtex 0. For each iteration fill in the set S, the distance list, and the predecessor list. Use N for NONE/NULL.

Iter #	S	D					P				
1	$\{0\}$	0	5	∞	∞	10	N	0	N	N	0
2	$\{0,1\}$	0	5	15	∞	7	N	0	1	N	1
3	$\{0,1,4\}$	0	5	8	9	7	N	0	4	4	1
4	$\{0,1,4,2\}$	0	5	8	9	7	N	0	4	4	1
5	$\{0,1,4,2,3\}$	0	5	8	9	7	N	0	4	4	1

Scratch Work; Not Graded:

7. Suppose G is a simple, connected, undirected, (positive) weighted graph with V vertices and suppose D is its $V \times V$ distance matrix, meaning $D[i][j]$ is the length of a shortest path between vertex i and vertex j. [15 pts]

   Suppose u, v are specific vertices in the graph which are connected by an edge. You have developed two optimized algorithms specific to this graph:

   - `dijkstra-u(x)` which returns the shortest distance from u to x, for any x.
   - `dijkstra-v(x)` which returns the shortest distance from v to x, for any x.

   Neither of these returns anything except the shortest distance - no paths, predecessors, etc. But because you are brilliant each of these runs in $\Theta(1)$ time.

   Suppose the weight of the edge joining u and v changes to another positive number. Describe a $\Theta\left(V^2\right)$ algorithm which will update D accordingly, if necessary. Make sure your description also makes it clear why your algorithm does what it needs to do.

   Solution:

   The only way a distance in D could change is the path using that updated edge is shorter than the path originally being used.
   Thus for each pair of vertices i, j we calculate:

   $$\texttt{dijkstra-u(i)} + \text{weight}(u, v) + \texttt{dijstra-v(j)}$$

   and:

   $$\texttt{dijkstra-v(i)} + \text{weight}(v, u) + \texttt{dijstra-u(j)}$$

   If either value is less than $D[i][j]$ then we update it accordingly.
   Since each calculation takes $\Theta(1)$ time and there are V^2 pairs to check we know that the algorithm is $\Theta(V^2)$.