CS 477: HW 8

Due on December 12, 2023

Dr. Monica Nicolescu Section 1001

Christopher Howe

Problem 1

Problem Description

Answer the questions below regarding the following graph

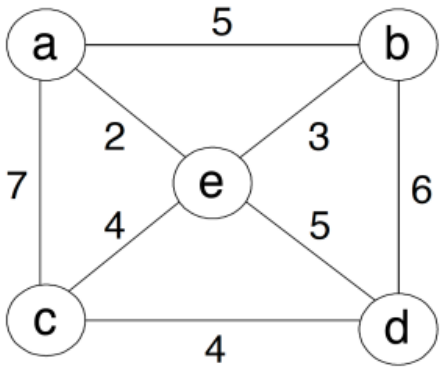

Figure 1: Minimum spanning tree graph

In what order are edges added to the Minimum Spanning Tree (MST) using Kruskal's Algorithm? List the edges by giving their endpoints.

Kruskals algorithm uses a disjoint set data structure to add edges to the MST. Kruskals algorithm relies on the greedy strategy that adding the edge with the lowest first produces the MST. First, sort all edges by their weight $[(a,e),(e,b),(e,c),(c,d),(e,d),(b,d),(a,c)]$. Next, create a disjoint set data struct with each vertex being its own set. $[[a],[b],[c],[d],[e]]$. Add from the sorted list. The first edge to be added is (a,e). Since these aren't in the same set, their sets are combined $[[a,e],[b],[c],[d]]$. The next edge to be added is (e,b). Since these aren't in the same set, their sets are combined $[[a,b,e],[c],[d]]$. The next edge to be added is (e,c). Since these aren't in the same set, their sets are combined $[[a,b,c,e],[d]]$. The next edge to be added is (c,d). Since these aren't in the same set, their sets are combined $[[a,b,c,d,e]]$. Since there is only one set, all the other edges can be disregarded.
After all of this the edges are added in the order $[(a,e),(e,b),(e,c),(c,d)]$**.**

In what order are edges added to the MST using Prim's Algorithm starting from vertex a? List the edges by giving their endpoints.

Prims algorithm uses a priority queue with all vertices not added to the tree. After each vertex is added to the MST, the keys for vertices adjacent to the added vertex must be updated. The algorithm uses the vertex with lower keys in the priority queue first, connecting vertices to the tree using their shortest path to the tree. Initially, the queue has all elements with infinite keys $[a:\infty,b:\infty,c:\infty,d:\infty,e:\infty,]$. A is chosen at random to be the "root" of the tree. When A is added to the tree, the values of c e and b in the queue are updated to their shortest path to A $[e:2,b:5,c:7,d:\infty]$. E is the chosen as the next vertex in tree. When E is added the tree, the values for b c and d are updated to $[b:3,c:4,d:5]$. B is the chosen as the next vertex in tree. When B is added the tree, the values for c and d are updated to $[c:4,d:5]$. C is the chosen as the next vertex in tree. When V is added the tree, the value for d are updated to $[d:4]$. D is added to the tree last since it is the last element in the queue.
After all of this the edges are added in the order $[(a,e),(e,b),(e,c),(c,d)]$**.**

Problem 2

Problem Description

A graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. We can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called 2-colorable. For example, graph (i) is bipartite while graph (ii) is not. **Write pseudocode for a DFS-based algorithm for checking whether a graph is bipartite.**

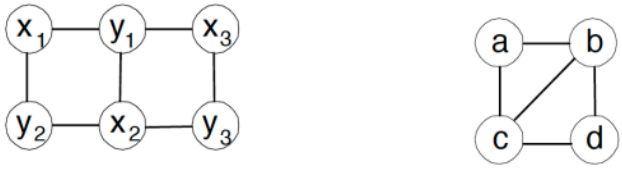

Figure 2: Shows a bipartite and a non bipartite graph

Solution

Pseudo Code

```
1:  procedure isBipartite(graph: G)
2:      is_bipartite ← True
3:      procedure bipartiteHelper(vertex, graph: G)
4:          neighborsState ← None
5:          for neighbor in graph.adj[vertex] do
6:              if graph.nodes[neighbor]['color'] ≠ 'white' then
7:                  if neighborsState ≠ graph.nodes[neighbor]['set'] and neighborsState ≠ None then
8:                      is_bipartite ← False
9:                  else
10:                     neighborsState ← graph.nodes[neighbor]['set']
11:                 end if
12:             end if
13:         end for
14:         if neighborsState ≠ None then
15:             graph.nodes[vertex]['set'] ← ¬neighborsState
16:         else
17:             graph.nodes[vertex]['set'] ← True
18:         end if
19:     end procedure
20:     depthFirstSearch(graph, bipartiteHelper)
21:     return is_bipartite
22: end procedure
```

Algorithm 1: isBipartite(graph)

Problem 2 continued on next page...

Python Implementation

```
import networkx as nx;
import matplotlib.pyplot as plt;
from depthFirstSearch import depthFirstSearch;

def isBipartite(graph: nx.Graph):
    is_bipartite = True

    def bipartiteHelper(vertex, graph: nx.Graph):
        nonlocal is_bipartite
        neighborsState = None
        for neighbor in graph.adj[vertex]:
            if graph.nodes[neighbor]['color'] != 'white': # check if neighbor has been explored
                if neighborsState != graph.nodes[neighbor]['set'] and neighborsState != None:
                    is_bipartite = False
                else:
                    neighborsState = graph.nodes[neighbor]['set']

        if neighborsState != None:
            graph.nodes[vertex]['set'] = not neighborsState
        else:
            graph.nodes[vertex]['set'] = True

    depthFirstSearch(graph, bipartiteHelper)
    return is_bipartite
```

Problem 3

Problem Description

In a directed graph, a vertex with no incoming edges is also called a source vertex. Describe how you would find a source, or determine that such a vertex does not exist, in a directed graph represented by its adjacency matrix. Indicate the time efficiency of your solution.

Solution

As a source vertex lacks incoming edges, every other vertex must also lack an edge leading to the source vertex. One potential algorithm involves representing the adjacency matrix as an array with indices corresponding to each vertex. To identify a source vertex, the algorithm iterates through all vertices, checking if there are connections in their respective "rows" at the index of the vertex being examined. This process is repeated for all vertices to determine if any of them are source vertices. Each vertex check requires $O(n)$ time to iterate through all vertices, resulting in an overall time complexity of $O(n^2)$ to identify a source vertex, if one exists.

Problem 4

Part A

Problem Description

Run the topological sort algorithm on the graph below and indicate the final ordering of vertices obtained. In the main DFS loop, choose white vertices in alphabetical order (thus, the initial source will be vertex a)

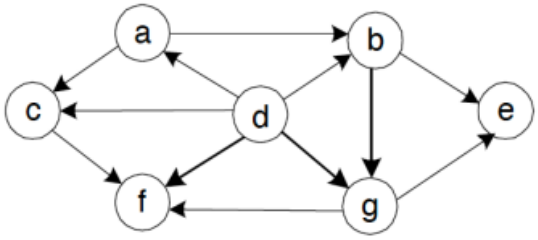

Figure 3: Problem 4 Part A

Solution

In order to preform the topological sort, first the depth first search algorithm must be ran. Elements are visited in alphabetical order. So, the source of the depth first search is a. The diagram below demonstrates the search.

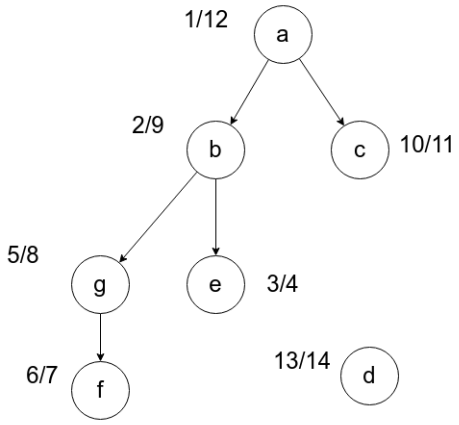

Figure 4: Timestamps and tree for DFS originating at A

After each element is "finished", add the elements to the front of a linked list. This produces a linked list that looks like this $[d, a, c, b, g, f, e]$. Notice that d has the highest finishing time, so it is displayed first, a has the second highest finishing time and is displayed second etc.

Part B

Problem Description

Run DAG-SHORTEST-PATHS on the directed graph from part a). Consider the following edge weights for the graph: ac = 2, ab = 3, be = -1, bg = -4, cf = 7, da = 1, db = 2, dc = 5, df = -3, dg = 4, ge = 1, gf = -4.

Solution

The first step of the DAG shortest path algorithm is to topologically sort the vertices of G. This has already been completed in part A. The next step is to initialize the single source problem. This gives every vertex an infinite estimated path length(d) and no shortest path tree parent. This excludes the source node a which gets a d value of 0. Now, the shortest path can be determined by relaxing every edge connecting every vertex. The vertexes are relaxed in topological order. These calls can be visualized in the tabled below. The DAG-shortest path algorithm produces the following shortest paths from a to each vertex: $[a:0, b:3, c:2, d:\infty, e:0, f:9, g:-1]$. There is also a diagram of the shortest path tree.

Relax Call	Conditional	new d[v]	Relax Call	Conditional	new d[v]
$(d, a, 1)$	$0 \geq \infty + 1 \implies F$	0	$(a, c, 2)$	$\infty \geq 0 + 2 \implies T$	2
$(d, b, 2)$	$\infty \geq \infty + 2 \implies F$	∞	$(c, f, 7)$	$\infty \geq 2 + 7 \implies T$	9
$(d, c, 5)$	$\infty \geq \infty + 5 \implies F$	∞	$(b, e, -1)$	$\infty \geq 3 - 1 \implies T$	2
$(d, f, -3)$	$\infty \geq \infty - 3 \implies T$	∞	$(b, g, -4)$	$\infty \geq 3 - 4 \implies T$	-1
$(d, g, 4)$	$\infty \geq \infty + 4 \implies F$	∞	$(g, e, 1)$	$2 \geq -1 + 1 \implies T$	0
$(a, b, 3)$	$\infty \geq 0 + 3 \implies T$	3			

Table 1: DAG Shortest relax calls (u values are based on topological order from graph from part A)

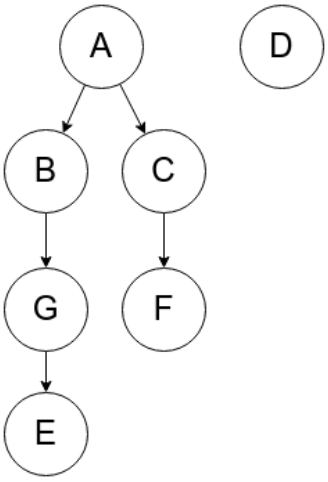

Figure 5: Shortest path tree derived from DAG-SHORTEST-PATH

Part C

Problem Description

Show how Dijkstra's algorithm runs on the graph below, with vertex a as the source.

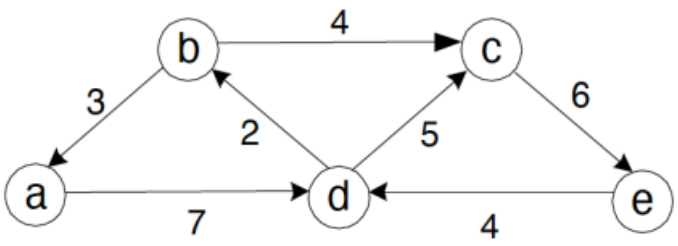

Figure 6: Problem 4 Part C

Solution

Dijkstras algorithm runs on the graph below by first adding all elements to a priority queue based on their estimated path distance. For each vertex in the set in initializes this estimate to ∞ except for the source a which it initializes to 0. Then, it adds all vertices to a priority queue with the key being their estimated path length. From here the algorithm relaxes all edges of each vertex in order of the priority queue. If any changes are made to d[v] during the relaxation, the value is updated in the priority queue as well. The table below demonstrates the relax calls made to run Dijkstras algorithm.

Relax Call	Conditional	new d[v]	Relax Call	Conditional	new d[v]
$(a, d, 7)$	$\infty \geq 0 + 7 \implies T$	7	$(b, c, 4)$	$12 \geq 9 + 4 \implies F$	12
$(d, b, 2)$	$\infty \geq 7 + 2 \implies T$	9	$(c, e, 6)$	$\infty \geq 12 + 6 \implies T$	18
$(d, c, 5)$	$\infty \geq 7 + 5 \implies T$	12	$(e, d, 4)$	$7 \geq 18 + 4 \implies F$	7
$(b, a, 3)$	$0 \geq 9 + 3 \implies F$	0			

Table 2: DAG Shortest relax calls (u values are based on topological order from graph from part A)

Problem 5

Indicate whether the following statements are TRUE or FALSE

Table 3: A True False Statements

Statement	**True?**
If e is a minimum-weight edge in a connected weighted graph, it must be among edges of at least one minimum spanning tree of the graph.	True
If e is a minimum-weight edge in a connected weighted graph, it must be among edges of each minimum spanning tree of the graph.	False
If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.	True
If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.	False

NOTE: The question specifies "**a** minimum weight edge" in statement b. There is an edge case where two nodes could have two minimum weight edges between them (ie verts A and B have two edges between them with length 1) It is not true that all MSTs would contain both edges. In fact, none of them would.