**Test 3**

Name: \_\_\_\_\_Jamal Joseph\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

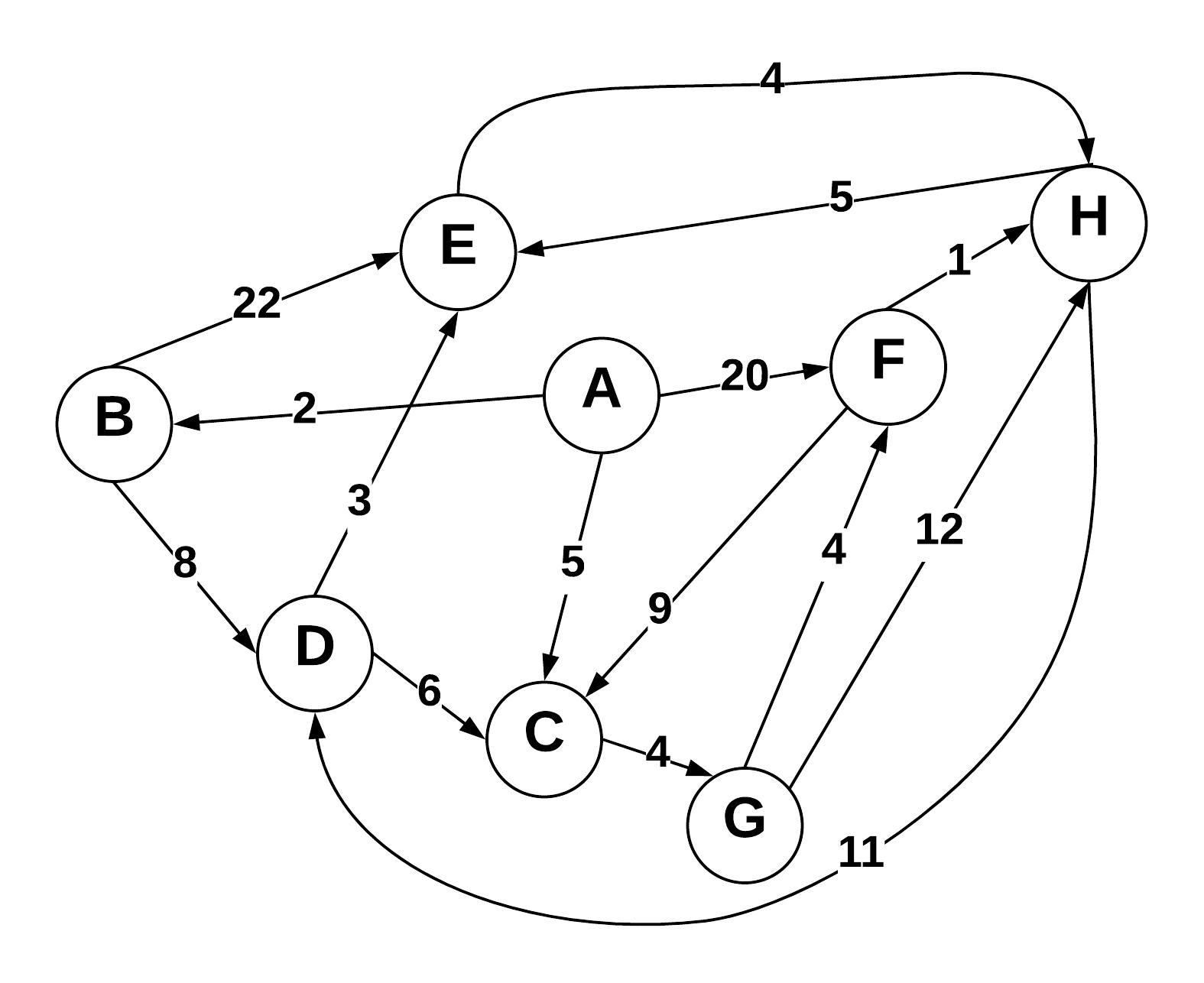
* Everything you turn in must be digitally created.
* No handwriting (except for signature below).
* You must work alone.
* Sharing of answers will result in a 0 on the exam, and possible F in the course.
* Send me your digitally created exam by Friday, May 4th by Midnight on a private slack message.
* Bring your printed signed copy by Monday Morning 10:00 am to my office.

|  |  |  |
| --- | --- | --- |
| Question | Possible | Score |
| 1 | 10 |  |
| 2 | 10 |  |
| 3 | 10 |  |
| 4 | 10 |  |
| 5 | 10 |  |
| 6 | 10 |  |
| 7 | 10 |  |
| 8 | 10 |  |
| 9 | 10 |  |
| 10 | Bonus |  |
| Total: |  |  |

By signing this, your saying “I worked alone and did not plagiarize”:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1) Dijkstra’s Algorithm**



Use Dijkstra’s algorithm to compute the shortest paths from vertex A to every other vertex. Show your work in the space provided below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same distance, use alphabetical order to determine which one is selected first. Also list the vertices in the order which Dijkstra's algorithm marks them as discovered.

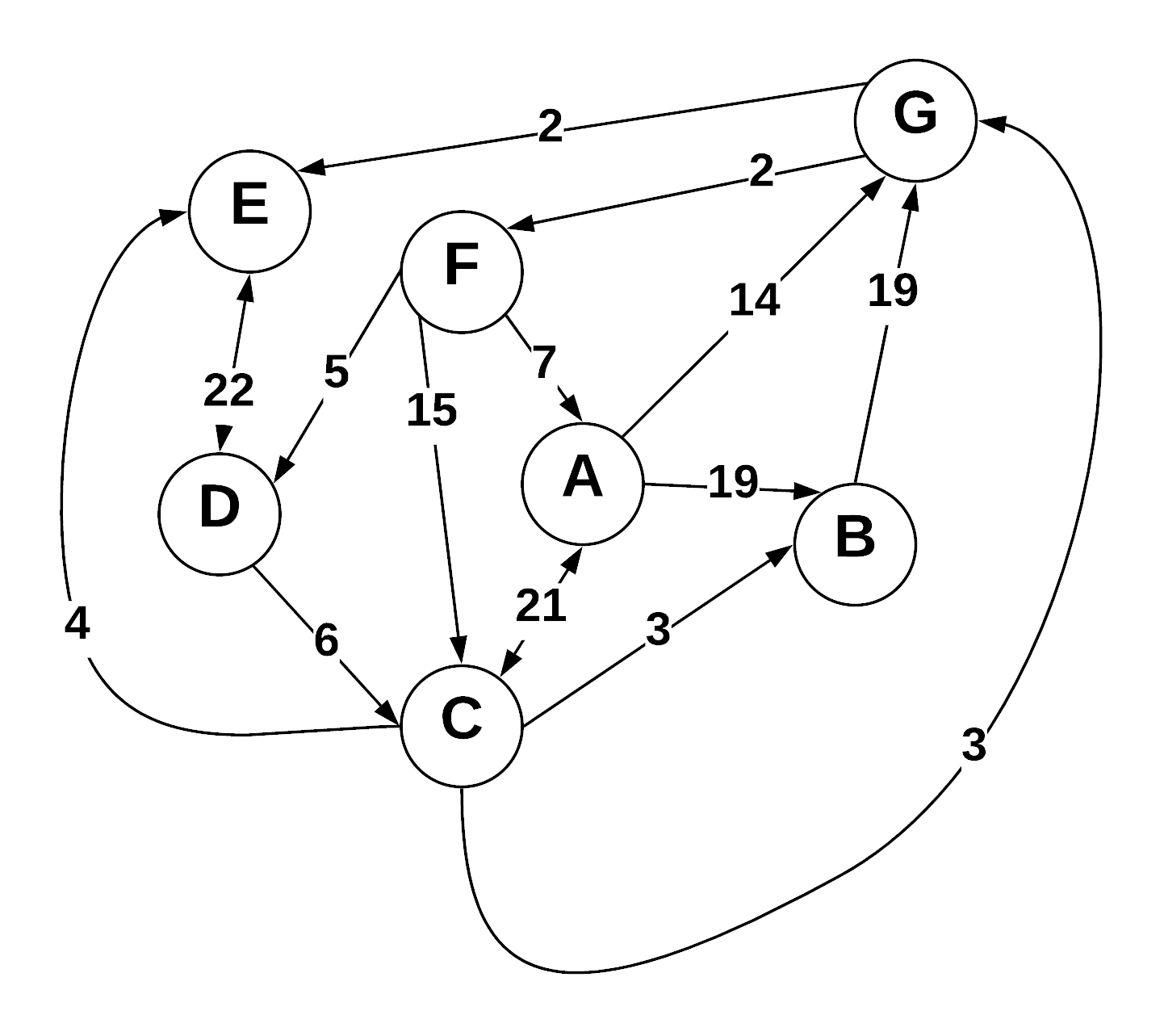
Vertices in Order of Discovery:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | G | D | E | F | H |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Known | Cost | Previous |
| A | T | 0 |  |
| B | T | 2 | A |
| C | T | ~~16~~, 5 | ~~D~~, A |
| D | T | 10 | B |
| E | T | 13 | D |
| F | T | ~~20~~, 13 | G |
| G | T | 9 | C |
| H | T | ~~21,17~~, 14 | ~~F,G,E~~,F |

I went by the order of smallest value and worked it as a directed graph as such I had to back track to a few times and update some vertex a few times in order to get the smallest value as seen with values like C and H.

**2) Prims Algorithm**



Step through Prim’s algorithm to calculate a minimum spanning tree starting from vertex *G.* Show your steps in the table below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same distance, use alphabetical order to determine which one is selected first. Also list the vertices in the order which Prims algorithm discovers them.

Vertices in Order of Discovery:

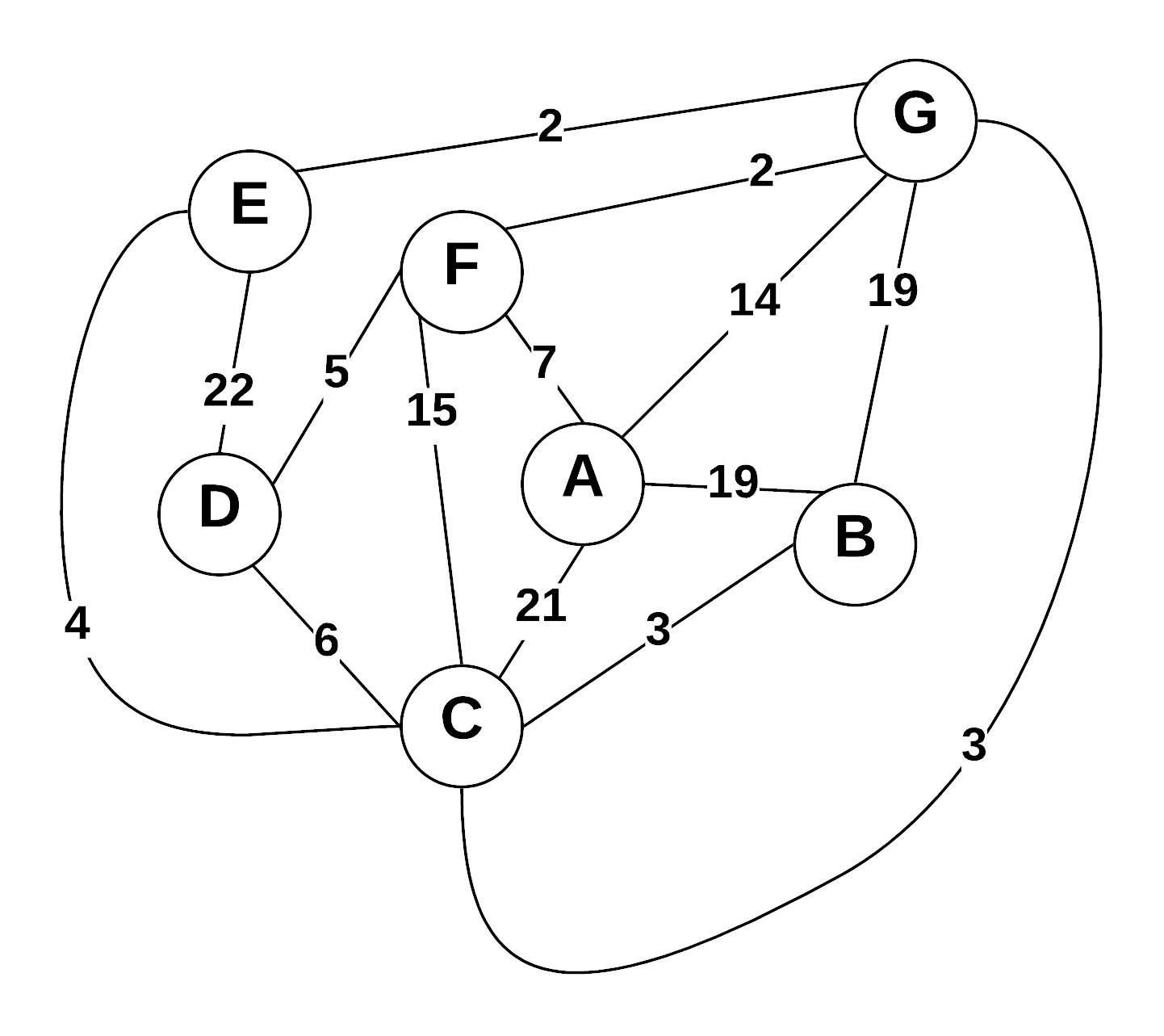
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| G | E | F | C | B | D | A |  |  |

* S = Vertices in spanning tree
* U = ! S (vertices not in S)
* Cut = edges going across cut listed alphabetically: (A B) , (C D) , etc.

|  |  |  |
| --- | --- | --- |
| *S (spanning tree)* | *U* | Cut (alphabetize) |
| G,E | A,B,C,D,F | {(E,G)} |
| G,E,F | A,B,C,D | {(E,G), (F,G)} |
| G,E,F,C | A,B,D | {(C,E),(E,G), (F,G)} |
| G,E,F,C,B | A,D | {(B,C),(C,E),(E,G),(F,G)} |
| G,E,F,C,B,D | A | {(B,C),(C,E),(D,f),(E,G),(F,G)} |
| G,E,F,C,B,D,A |  | {(B,C),(C,E),(A,F),(D,F),(E,G),(F,G)} |
|  |  |  |
|  |  |  |

Prims algorithm does not work on directed graphs so I worked this like it was an undirected graph.

**3) Kruskel’s Algorithm**



Use Kruskal’s algorithm to calculate a minimum spanning tree of the graph. Show your steps in the table below, including the disjoint sets at each iteration. If you can select two edges with the same weight, select the edge that would come alphabetically last (e.g., select E—F before B—C. Also, select A—F before A—B).

* Edge Added: put edges added to MST marked as (A B), (E G), etc.
* Edge Cost: weight of edge added
* Running cost is total weight of spanning tree at the point another edge is added.
* Disjoint sets starts as: (A) (B) (C) (D) (E) (F) (G) , and as edges are added => (A) (B C) (D) (E) (F) (G)

|  |  |  |  |
| --- | --- | --- | --- |
| Edge Added | Edge Cost | Running Cost | Disjoint Sets |
| (E,G) | 2 | 2 | (A) (B) (C) (D) (E,G) (F) (G) |
| (F,G) | 2 | 4 | (A) (B) (C) (D) (E,G) (F,G) (G) |
| (B,C) | 3 | 7 | (A) (B,C) (C) (D) (E,G) (F,G) (G) |
| (C,G) | 3 | 10 | (A) (B,C) (C,G) (D) (E,G) (F,G) (G) |
| (D,F) | 5 | 15 | (A) (B,C) (C,G) (D,F) (E,G) (F,G) (G) |
| (A,F) | 7 | 22 | (A,F) (B,C) (C,G) (D,F) (E,G) (F,G) (G) |
|  |  |  |  |
|  |  |  |  |

**4) Prims Vs Kruskels**

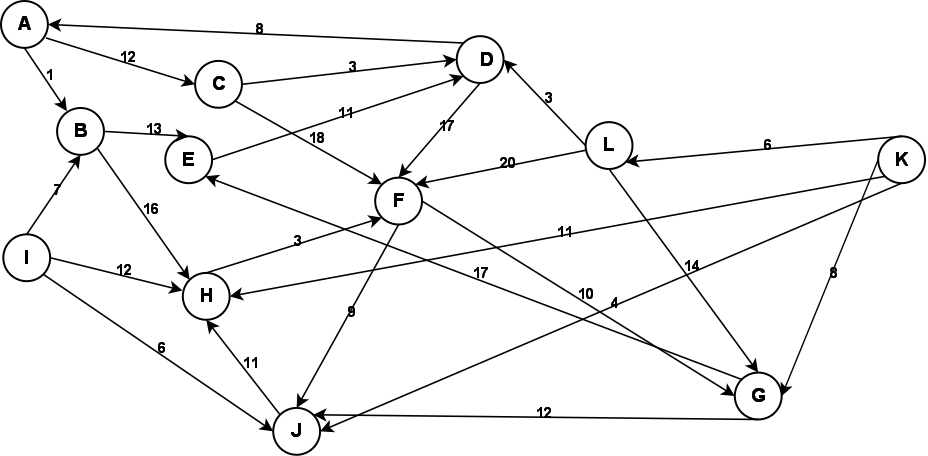
Explain why Prim’s algorithm is better for dense graphs, while Kruskal’s algorithm is better for sparse graphs. What data structures are used to represent each?

The complexity of both Prim’s and Kruskal’s change depending how dense or sparse the graph is, with how Prim’s algorithm works it takes advantage of dense graphs, Prim’s works faster and more efficiently when all the vertices and edges are close together which it why it has a time complexity of O(e +vlog(v)) while Kruskal’s has a time of O(e log (v)) where e is the number of edges and v is the number of vertices. Prims algorithm breaks down in terms of efficiency and time complexity when it has to deal with more sparse graphs. As a result its time complexity increases to O(n^2) while Kruskal’s is O( e log v) which runs better than the Prim’s Algortihm. Prim’s Algorithm is represented by a single tree while Kruskals is represented by a forest

**5) Greedy Algorithms**

1. Define “Greedy Algorithm”
2. Give an example of a greedy algorithm with explanation of its greediness and performance.
3. Can greedy algorithms produce “optimal” solutions? Short explanation.
4. A greedy algorithm is an algorithm that builds its solution in pieces by continually choosing the most optimal and beneficial piece, the pieces are picked based on which with give the most favorable and immediate benefit.
5. One example of a greedy algorithm is Kruskal’s Algorithm. With Kruskal’s the algorithm creates a minimum spanning tree for all the nodes in the graph. What makes Kruskal’s greedy is that it picks each node one by one and always goes for the smallest weight edge that also does not cause a cycle within the tree.
6. Greedy algorithms can produce optimal solutions. Greedy algorithms work on the principle of making the optimal choice and each step in a problem to create the most optimal solution. So when given any problem that would allow the greedy algorithm to sequentially go through and make optimal choices optimal solutions would be created.

**6) Graph Traversals**



Given the above graph, provide the output of a breadth first and a depth first search. Make choices based on smallest edge weight, then alphabetical to break ties. Start at node A for both.

**Depth First:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | E | D | F | J | H | G | C |  |  |  |

I, K, and L are not printed out in the depth first search because there is no case of any node pointing to these nodes to allow it to be added to the stack for it to be printed out when starting from A.

**Breadth First:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | E | H | D | F | G | J |  |  |  |

I, K and L could not be visited due to no path leading to them when starting with A

**7) Graph Storage / Manipulation**

Given that a weighted directed graph is represented as an adjacency matrix called *adjM*, write a method that reverses all the edges of the graph. That is, for every edge ( A , B ) in the original graph, there will be an edge  ( B , A ) in the reversed graph with the same weight. Your function should be called *reverse*.

Adjlist adjM // old adjacency list

Adjlist adjN // new adjacency list

Void reverse (adjM[], adjN[], int weight)

{

for (int i =0; i< numvertices< i++ )

{

for (int n : adjm[i])

adjN[n].add(i);

adjN[n].weight(i);

}

}

Assuming the old adjacency list has data in it already, what I did above was to loop through the old graph and take the edges then add them in reverse to the new graph. We go through all the vertices and edges with the for (int n: adjm[i]) line then add the edge and swap it as we do.

**8) Graph Traversal**

Write a method that returns whether a graph is a tree. Your method takes a graph *G=(V,E)* as the input and outputs a boolean value. Your method should be called *isTree*().

Graph G;

Bool::Graph isTree(int i; int parentnode)

{

bool visited;

visited[i] = true;

For (int r=adjM[i].begin(); r != adjM[i].end(); i++)

{

If(adjM[r] != visited[r];

{

If(isTree(r, i)

} return true;

}

else

return false;

}

I traversed through the tree and tried to see if I could find any cycles when starting from the initial vertex in order to see if the tree is a tree or not. I recursively call the function to make the process easier.

**9) Huffman Coding**

**(A)** What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers:

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21.

Show your answer as a tree. *Note:* assume that the ordering on the nodes is first by the frequency, and then by the alphabetic order of the node label, so that ab:2 precedes c:2; the node labels are alphabetized too, so that we have a node ab:2 but not ba:2.

54

H

33

G

20

12

F

7

E

D

4

2

C

A

B

|  |  |
| --- | --- |
| **letter** | **binary** |
| **A** | **1111100** |
| **B** | **1111101** |
| **C** | **111111** |
| **D** | **11110** |
| **E** | **1110** |
| **F** | **110** |
| **G** | **10** |
| **H** | **0** |

**(B)** Use the code from part (a) to decode the string 11111111111001111101. (As a check: the result should be the name of something that is often yellow.)

The answer is CAB

C A B

111111 1111100 1111101

**10) Bellman Ford (Optional)**

Using the graph from question 3, show a Bellman Ford solution.