**Test 3**

Name: \_\_\_\_\_\_\_\_Jackson Tran\_\_\_\_\_\_\_\_\_\_\_\_\_

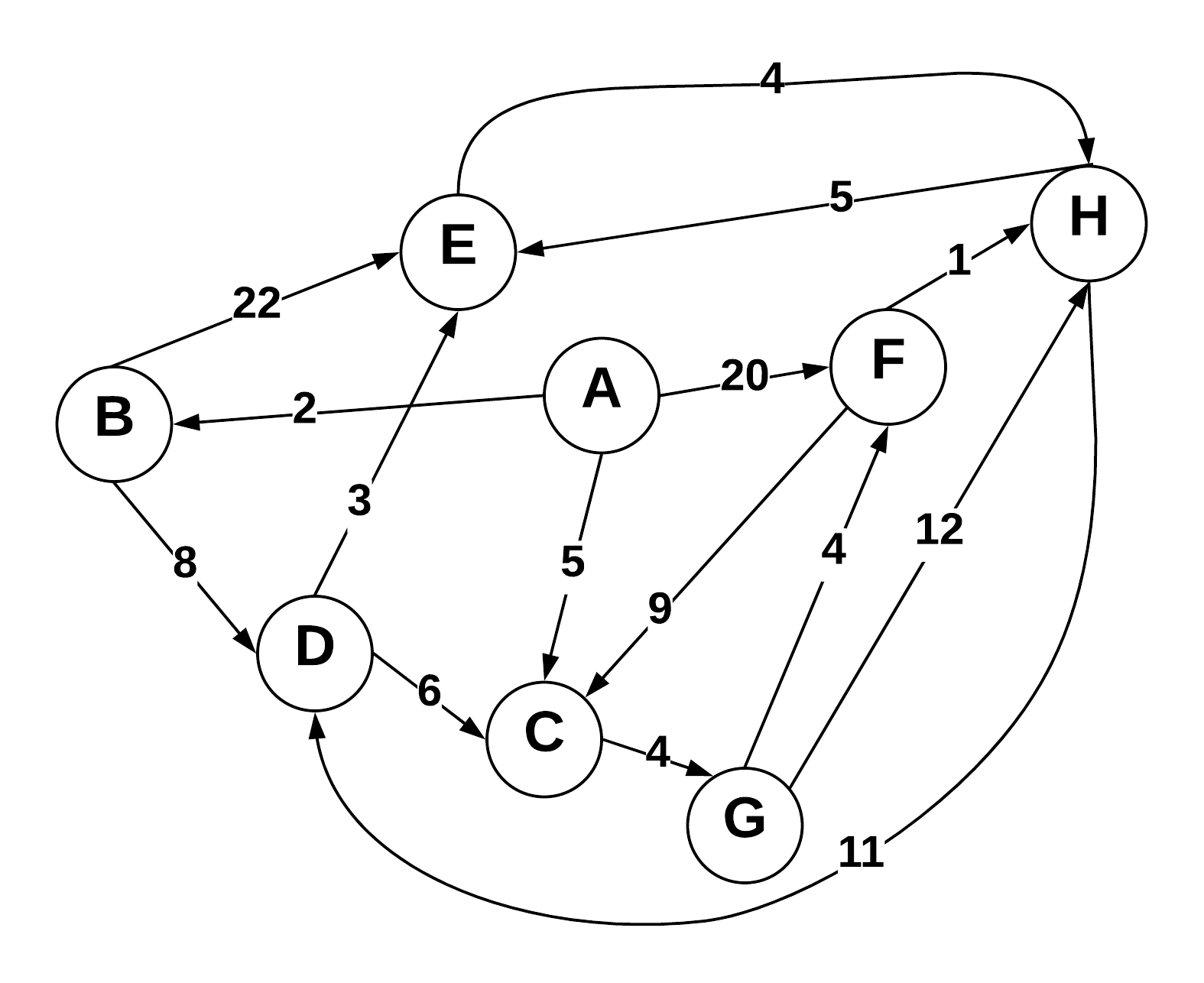
* Everything you turn in must be digitally created.
* No handwriting (except for signature below).
* You must work alone.
* Sharing of answers will result in a 0 on the exam, and possible F in the course.
* Send me your digitally created exam by Friday, May 4th by Midnight on a private slack message.
* 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.

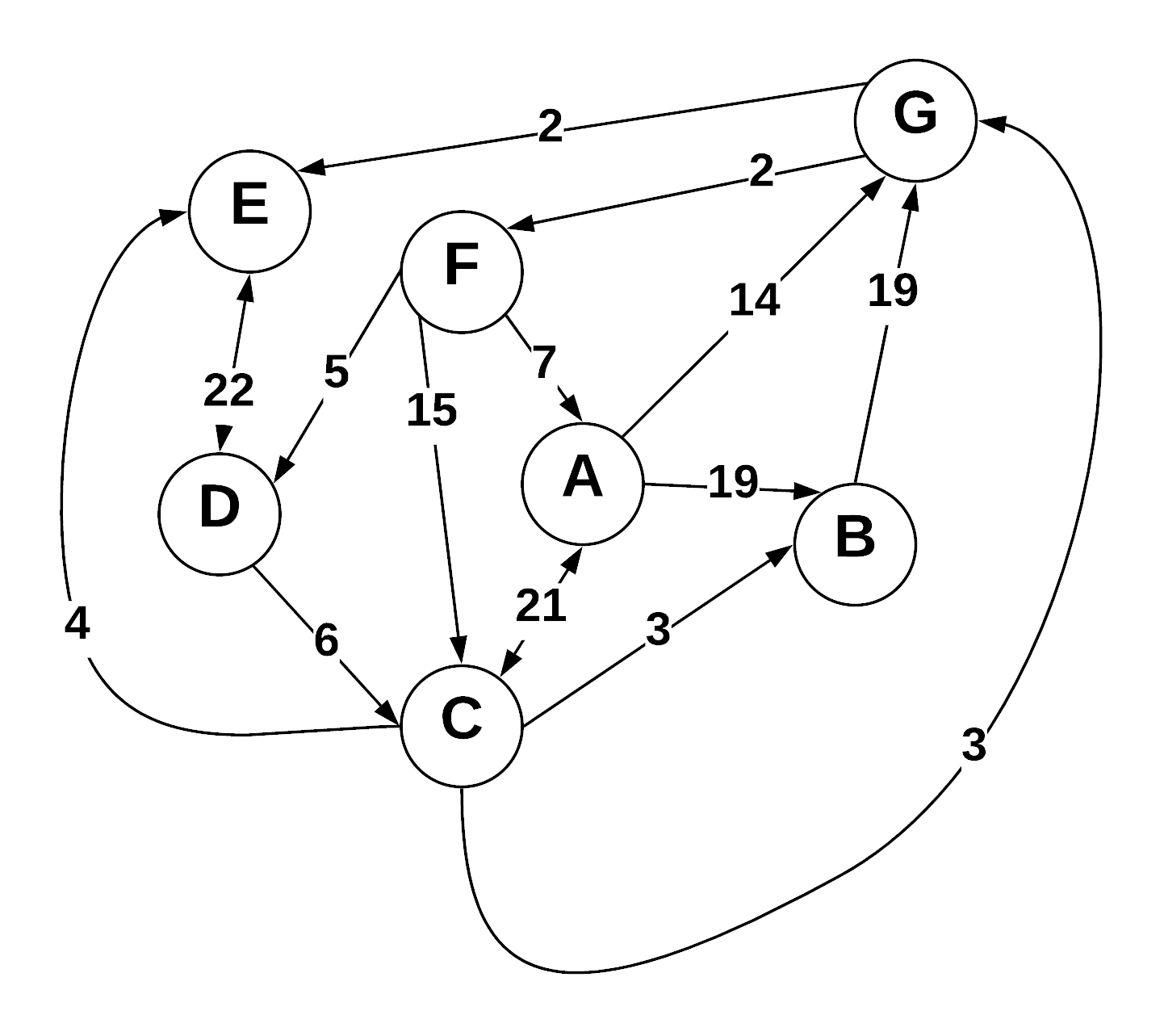
Vertices in Order of Discovery:

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

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Known | Cost | Previous |
| A | B, C, F | 2, 5, ~~20~~ | NULL |
| B | D, E | 10, ~~24~~ | A |
| C | G | 9 | A |
| D | C, E | ~~16~~, 13 | B |
| E | H | 17 | ~~B~~, D |
| F | C, H | ~~22~~, 14 | ~~A~~, G |
| G | F, H | 13, ~~21~~ | C |
| H | D, E | ~~25,~~ 18 | ~~G~~, ~~E~~, F |

Starting from A, the known, accessible vertices are B, C, and F. The cost for these are 2, 5, and 20, respectively. Mark these costs down and let the previous node of those nodes be A, then mark A as visited. The algorithm will discover B next as it has the lowest cost to take. At B, nodes D and E are accessible at the total cost of 10 for D and 24 for E. Mark those down, with B being the pervious node for those two. Mark B as visited. The next step is discovering C as the next lowest cost to take. It only has one outward edge going to G for a cumulative value of 9, so mark that down with C as the previous node and then mark C visited. Now at node G, there are two outward edges going to F and H. The cumulative value for going to F taking this path costs 13, which is less then the edge (A F), so mark out the previous value and set the new pervious node for F as G. The other edge from G goes to H for a cost of 12+9=21. Mark it down with G as the previous node and mark G visited. Go back to D with the cumulative cost of 10. It has an edge going to E and C. The path to take to get to from D to E costs 13, which is less than the previous 22. E’s new previous node is D, marking out B, and the cost is 13. The edge from D to C will cost 16, which is higher than taking the edge (A C) making it irrelevant. D is now visited. The path to take to E now costs the same as the path to take to F, but we will use alphabetical order to break the tie. Now at E, there is only one edge to H, costing at a total of 17. This beats the initial path value of 21, making H’s previous node E and cost now 17. E has been visited now. Now that we did E for the tie breaker, go back to F. The only edge to matter for F in this situation is the (F H) edge since the (F C) edge isn’t useful. The (F H) edge has a cumulative cost of 14, beating the previous path to H of cost 17. F is now H’s new previous node with the cost of 14. F is now visited and the last node to be discovered is H. The outward edges from H won’t be replacing the cost of any previous paths, so now H is visited and the Djikstra’s Algorithm is completed.

**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 | D | C | A | B |  |  |

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

Looking at the graph as is

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

Prim’s Algorithm is typically supposed to be utilized with undirected graphs, but we’ll step through the problem as it is, then look at it as if it were an undirected graph.

Looking as it is now, we start off with G. The vertices not in the spanning tree are A, B, C, D, E, and F. There haven’t been any cuts since we just started. We’ll be traversing to nodes only if we have access by the direction of the edges. For the first step, we have access to both E and F from g, both at the cost of 2. We will take the edge (G E) because of alphabetical order. E is the next node to be discovered and added to the spanning tree. The remaining nodes in U are A, B, C, D, and F. The first cut made is (G E). E’s only outward edge is to D with a cost of 22. However, we still have the (G F) edge to go to because it has the lower cost of 2. Discovering F and adding it to the spanning tree, the remaining nodes are A, B, C, and D. The second cut is edge (G F). F has edges allowing access to D and A at the cost of 5 and 7, respectively. So, we’ll take the edge (F D). D is then discovered and added to the spanning tree leaving A, B, and C left. Edge (F D) is added to the cut list as well. Now we have the edge (D C) at a cost of 6, which is the lowest, accessible edge we can take. C is added to the spanning tree and now only A and B are the remaining nodes not in the tree. Edge (D C) is added to the cut list. The edge (C B) has a lower cost of 3 compared to the edge (F A). B is discovered, added to the spanning tree, and the edge (C B) is added to the cut list. Finally, we will go through edge (F A) to add to the cut list. A is the final node to be discovered and added to the spanning tree. No more nodes are left unvisited.

Vertices in Order of Discovery:

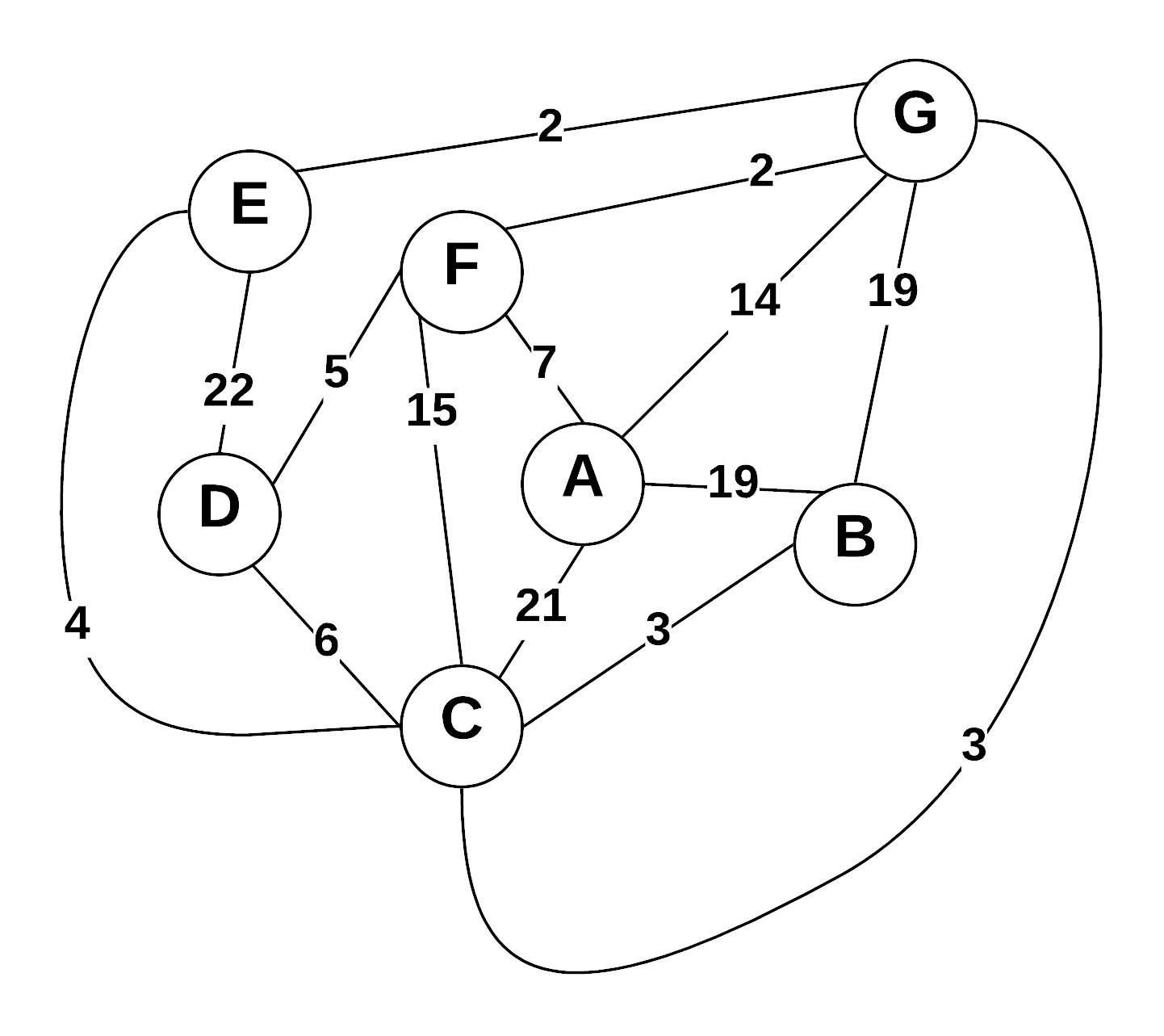
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| G | E | F | C | B | D | A |  |  |

Looking at the graph as if it were undirected

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

Now, let’s look at the graph as if it were undirected. The first two steps are the same as we were attempting to perform Prim’s algorithm on the same graph but directed. E will be added after the initial node G with edge (E G) being the first in the cut list. Edges are alphabetized since the graph is undirected, allowing us access both ways. Now F is added after E using the edge (F G), leaving the A, B, C, and D nodes not in the tree. This next step is where the results differ from looking at the graph as a directed graph. Since we’re allowed access both ways by looking at the graph undirectedly, we can go to C from G. This edge, (C G), only has a weight of 3, which is the lowest of our choices currently. We will add C to the spanning tree and add (C G) to the cut list. Next node to be accessed with the least amount of weight to it is B. We’ll access it through edge (B C). Add B to the spanning tree and that edge to the cut list. This leaves only A and D to be added to the spanning tree. With access to those nodes via F, we have edges (D F) and (A F) at the costs of 5 and 7, respectively. We’ll take (D F) to add to the cut list and add D to the spanning tree. Finally, we’ll take (A F) and add A as the last node in the spanning tree and (A F) as the last edge in the cut list.

**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 |
| (F G) | 2 | 2 | (A) (B) (C) (D) (E) (F G) |
| (E G) | 2 | 4 | (A) (B) (C) (D) (E G) (F G) |
| (C G) | 3 | 7 | (A) (B) (C G) (D) (E G) (F G) |
| (B C) | 3 | 10 | (A) (B C) (C G) (D) (E G) (F G) |
| (D F) | 5 | 15 | (A) (B C) (C G) (D F) (E G) (F G) |
| (A F) | 7 | 22 | (A F) (B C) (C G) (D F) (E G) (F G) |
|  |  |  |  |
|  |  |  |  |

Since this graph is undirected, we won’t have the same problems as the previous problem. Starting off, we will look at the edges with the lowest amount of weight. We have two edges, (F G) and (E G), with the cost of two. The instructions say to select the edge that comes alphabetically last, so we’ll choose (F G). Add two to the running cost and (F G) is part of the disjoint set. Now we can do the edge (E G). Add another 2 to the running cost and the edge is added to the disjoint set. Looking at out choices, we have another two edges equal to each other. Edge (C G) and edge (B C) both have a cost of 3, but because of the alphabetically last rule, we’ll choose edge (C G). Add this edge to the spanning tree and disjoint set and also add 3 to the running cost for a cumulative total of 7. Now we’ll add the edge (B C) and also add 3 to the running cost. Now looking at our choice of edges, the next lowest edge is (C E) with a weight of 4. However, we can’t choose this edge as it would create a cycle from (C G) to (C E) then (E G) and Kruskal’s Algorithm doesn’t allow cycles. So, our next best option would be edge (D F). We’ll add it along with it’s weight of 5 to the running cost. Finally, we’ll add edge (A F) and its weight of 7 to the running cost, leaving a final total running cost of 22.

**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 reason why Prim’s algorithm is better for dense graphs while Kruskal’s algorithm is better for sparse graphs is due to the number of edges in those graphs. A dense graph is known to contain a large amount of edges, while a sparse graph is the opposite with a small amount of edges.

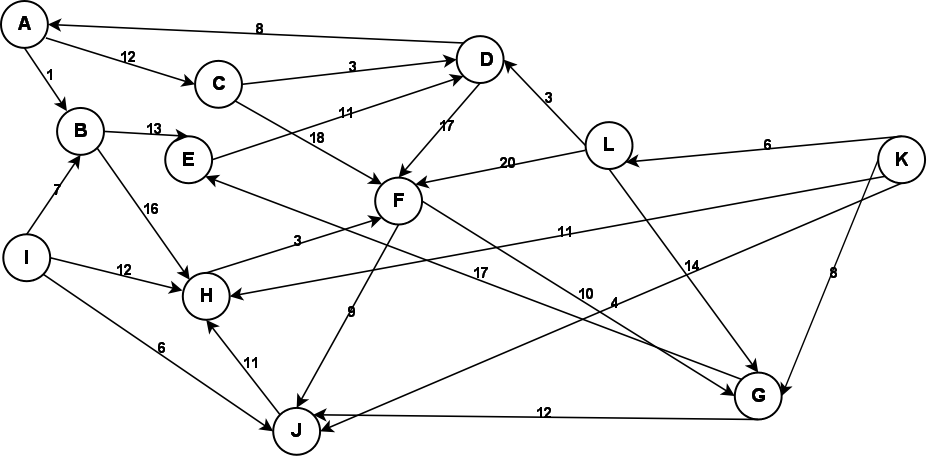
Kruskal’s algorithm:

**“Kruskal's algorithm** is a [greedy algorithm](http://en.wikipedia.org/wiki/Greedy_algorithm) used for finding a minimum spanning tree for a **connected**, **weighted**, and **undirected** graph. Meaning that the algorithm finds the set of edges that forms a tree including ***every vertex*** and has the ***lowest weight*** possible, without any [cycles](http://en.wikipedia.org/wiki/Cycle_(graph_theory)).”  
  
Prim’s algorithm:   
“**Prim's algorithm** is very similar to **Kruskal's Algorithm**. Both are very greedy and are used to find minimum spanning trees in weighted, connected, undirected graphs. The only real differences between the two is that Prim's algorithm can only add nodes adjacent to the current tree and any node can be added as the starting node.”  
  
In those definitions, Kruskal’s algorithm doesn’t allow cycles. A graph with a larger quantity of edges, or a dense graph, would have more possibilities of cycles being present. Thus, the Kruskal algorithm wouldn’t be as efficient to use as Prim’s algorithm since the Kruskal algorithm creates the minimum spanning tree without cycles. However, Prim’s algorithm may be harder to implement than Kruskal’s algorithm. Since a sparse graph isn’t as complex as a dense graph, you’d want to choose the simpler of the two algorithms, which would be Kruskal’s in this case. There wouldn’t be as many possibilities of cycles in sparse graphs, so the Kruskal algorithm would be better for the situation.  
  
Data structures representations:  
  
Kruskal’s Algorithm  
-Priority queue  
-Disjoint-set  
-Forest  
-Cluster  
  
Prim’s Algorithm  
-Adjacency list  
-Adjacency matrix  
-Heap

**5) Greedy Algorithms**

1. Define “Greedy Algorithm”  
   A Greedy Algorithm is an algorithm in which makes an optimal choice at each step. These algorithms are designed to maximize or minimize certain parameters, based on the situation.
2. Give an example of greedy algorithm with explanation of its greediness and performance.  
   Kruskal’s algorithm is an example of a greedy algorithm. It is utilized in creating a minimum spanning tree for a graph that’s connected, weighted, and undirected. It will seek to create a tree that incorporates all vertices while maintaining the lowest weight possible. It also doesn’t allow cycles. The complexity of this algorithms falls under O(ElogE), or O(ElogV), where E represents the number of edges and V represents the number of vertices in the graph.
3. Can greedy algorithms produce “optimal” solutions? Short explanation.  
   Depends on how you define optimal, but greedy algorithms typically do not produce optimal solutions. However, they do produce solutions that are in general reasonably satisfying solutions.

**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 |  |  |  |

Starting at A, we push A onto a stack, mark A as visited, and look at adjacent nodes. We look at edges (A B) and (A C) and take the former because it’s weight is smaller. We arrive at B, marking it as visited and push it onto the stack. Looking at adjacent nodes and respective weights of edges, we take the edge (B E). Pop E onto the stack. E only has one edge going to D, so take that edge. Pop D onto the stack and mark it visited. There are two edges going to A and F. We would take the edge to A since it has a lower weight, but A has already been visited, so we take the edge (D F). F is now visited and pushed onto the stack. Looking at the two available edges, we will move to J because of its lower weight. J is then marked as visited and pushed onto the stack. We only have one edge going to H, so we’ll make it visited and push it onto the stack. Now at H, there are no available edges to traverse. We will pop vertices off the stack until we can find an edge adjacent to those vertices that we can traverse. This will happen when looking at F on the stack. It will traverse the edge (F G), mark G as visited, and then we won’t be able to traverse from the point. Then we will keep popping off until we get back to A, take the edge (A C) and then C is marked as visited. These are all the vertices able to be traversed to using A as the starting node.

**Breadth First:**

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

Starting at A, we will look at each node adjacent to A. For each node, we will enqueue that node, order based on weight, and mark it as visited. After all nodes adjacent to the current node, which is A, are marked, we will look at the front of the queue to decide how to traverse. So, we will dequeue B and move to it, repeating the process. At B, the adjacent nodes to be queued are E and H. E will be queued first since it has the lower weight. Now that those nodes adjacent to B have been taken care of, the next node in the queue is C, so dequeue it. Adjacent to it are D with a weight of 3 and F with a weight of 18, so enqueue D then F. Now we dequeue E. It has an edge to D, but D has already been visited. Next up o dequeue is F. Node F has edges going to J and G with weights of 8 and 10, respectively. Enqueue J then G. These are all the accessible nodes, so the rest of the process is just to dequeue the queue.

**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* .  
  
void reverse(adjM matrix)

{

Int revMatrix[][]; //Matrix to hold transposed version of original matrix

for(int i = 0; i < matrix.rows; i++)//Loop through rows of the passed in matrix

for(int j = 0; j < matrix.cols; i++)//Loop through columns of passed in matrix

revMatrix[i][j] = matrix[j][i];

for(int i = 0; i < matrix.rows; i++)

for(int j = 0; j < matrix.cols; i++)

matrix[i][j] = revMatrix[i][j];

}

Create a matrix to hold the transposed matrix. Loop through the rows and columns of the passed in matrix. Set the reversed values in the created matrix. Loop through again, except now set the values in the passed in matrix to the values in the transposed matrix.

**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*().

(Return to this to define the functions)

bool detectCycle(vertex V)

{

}

bool allVisited(vertex V)

{

}

bool isTree(graph G)

{

If(detectCycle(vertex V) == false || allVisited(vertex V) == false)

return false;

else

return true;

}

A graph can’t be a tree if it has cycles or if the graph is not connected.

**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.

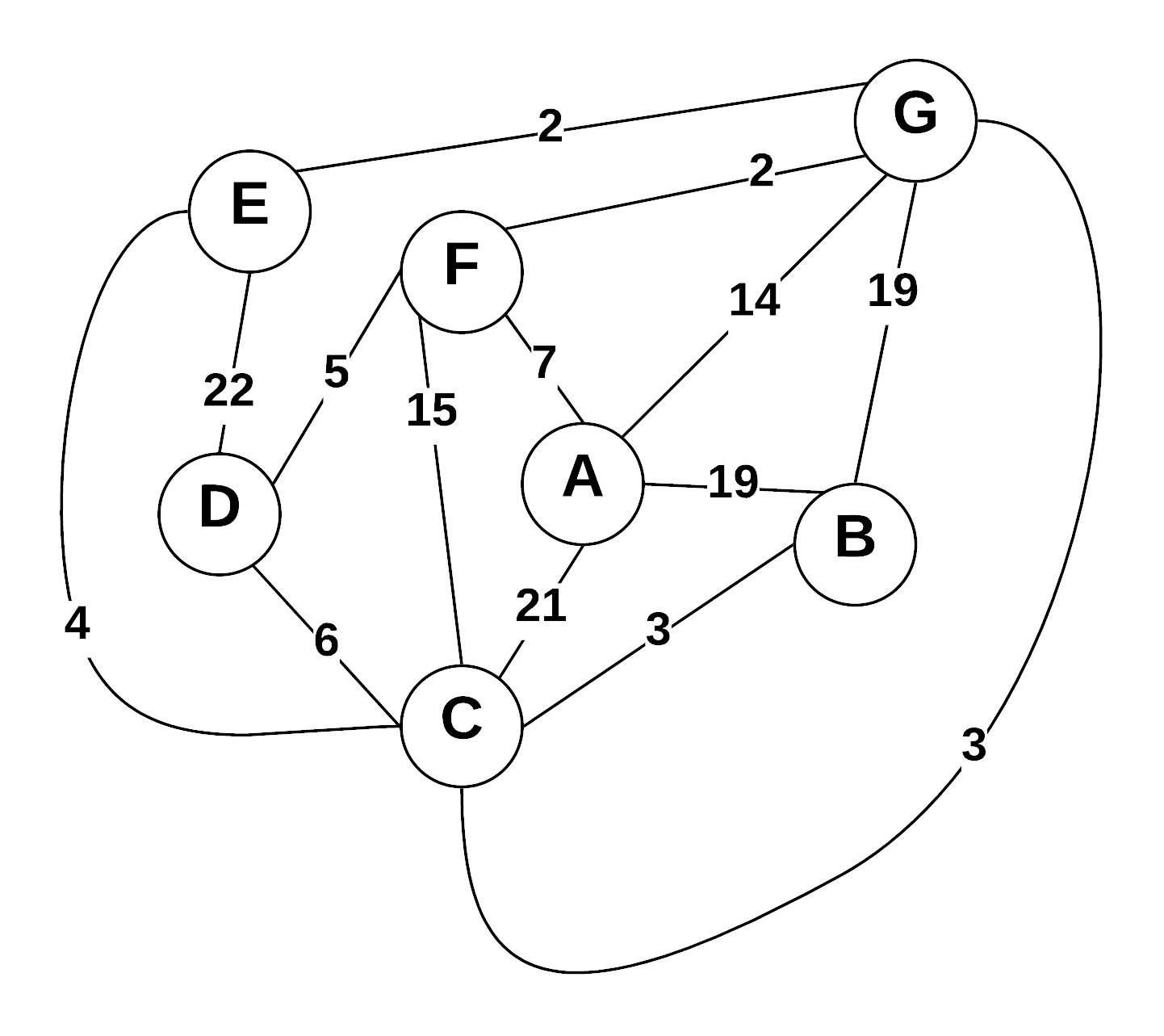
**(B)** Use the code from part (a) to decode the string 11111111111001111101. (As a check: the result should be the name of something that is often yellow.) 11111111111001111101

(A)

(B)  
(I don’t understand how to utilize the code, look up later)

**10) Bellman Ford (Optional)**

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



Iteration 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cost | 0 | 19 | ~~21~~, 12 | ~~27~~, 12 | ~~25~~, 11 | 7 | ~~14~~, 9 |
| Vertex | A | B | C | D | E | F | G |

Iteration 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cost | 0 | ~~19~~, 15 | 12 | 12 | 11 | 7 | 9 |
| Vertex | A | B | C | D | E | F | G |

Iteration 3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cost | 0 | 15 | 12 | 12 | 11 | 7 | 9 |
| Vertex | A | B | C | D | E | F | G |

Iteration 4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cost | 0 | 15 | 12 | 12 | 11 | 7 | 9 |
| Vertex | A | B | C | D | E | F | G |

Iteration 5

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cost | 0 | 15 | 12 | 12 | 11 | 7 | 9 |
| Vertex | A | B | C | D | E | F | G |

Iteration 6

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cost | 0 | 15 | 12 | 12 | 11 | 7 | 9 |
| Vertex | A | B | C | D | E | F | G |