**Test 3**

Name: \_\_\_\_\_Jakob Lopez\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

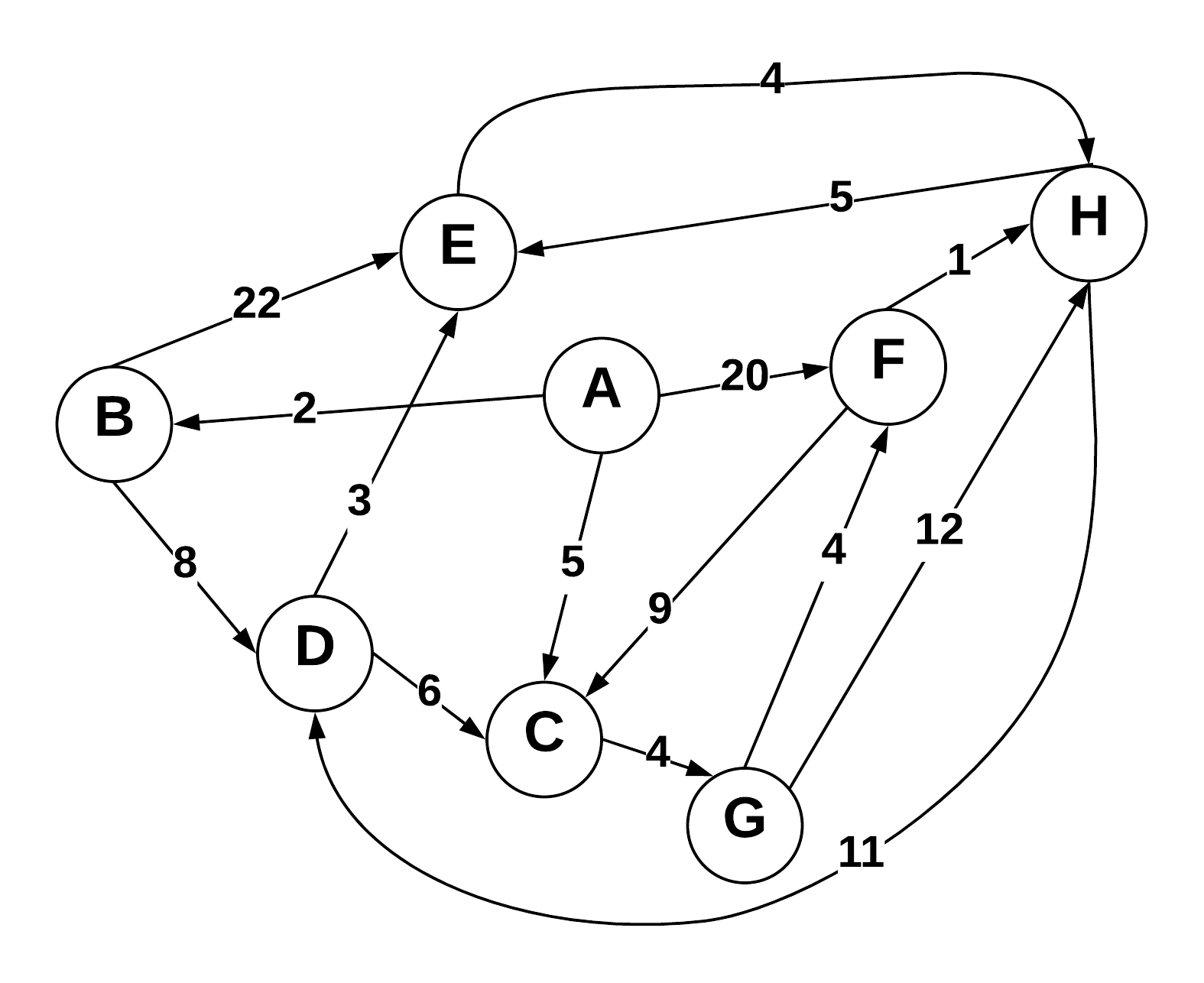
* Everything you turn in must be digitally created.
* No handwriting (except for signature below).
* You must work alone.
* Sharing of answers will result in a 0 on the exam, and possible F in the course.
* Send me your digitally created exam by Friday, May 4th by Midnight on a private slack message.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

\_\_\_\_\_\_\_\_\_Jakob Lopez\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1) Dijkstra’s Algorithm**



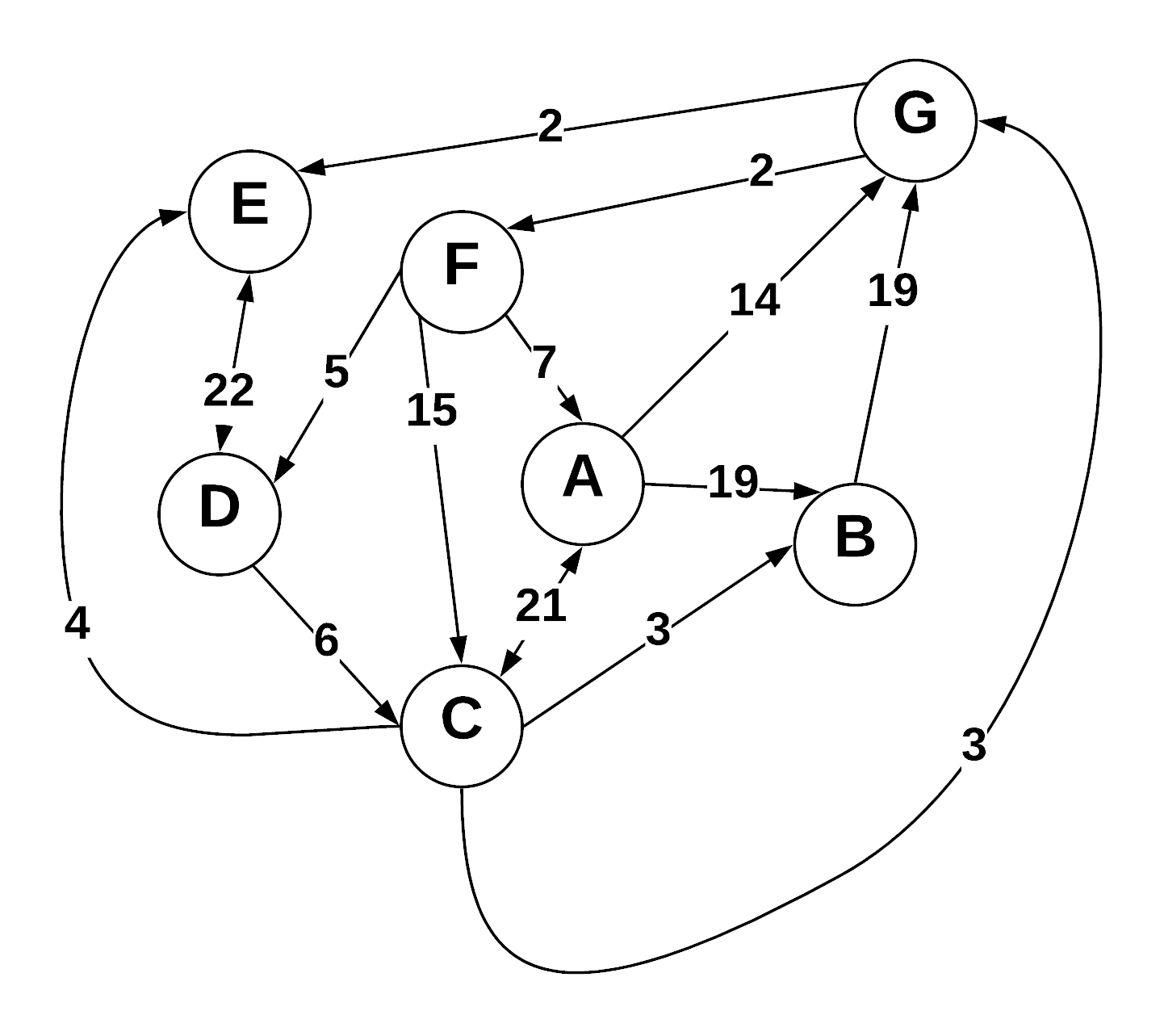
Use Dijkstra’s algorithm to compute the shortest paths from vertex A to every other vertex. Show your work in the space provided below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same 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 | A |
| B | FT | ~~∞~~2 | A |
| C | FFT | ~~∞~~5 | A |
| D | FFFFT | ∞~~∞~~10 | B |
| E | FFFFFT | ∞∞~~24~~  13 | ~~B~~ D |
| F | FFFFFFT | ∞~~∞20~~  13 | ~~A~~ G |
| G | FFFT | ∞∞~~∞~~9 | C |
| H | FFFFFFFT | ∞∞∞~~∞21~~ ~~17~~ 14 | ~~G~~ ~~E~~ F |

**2) Prims Algorithm**



Prim’s algorithm is supposed to be executed on an undirected graph, however it can be made to work on a directed graph. When drawing the cut, I only considered the out-going edges, since a vertex cannot go out of an in-going edge. All vertices contained at least 1 out-going edge, so all vertices were reachable

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.

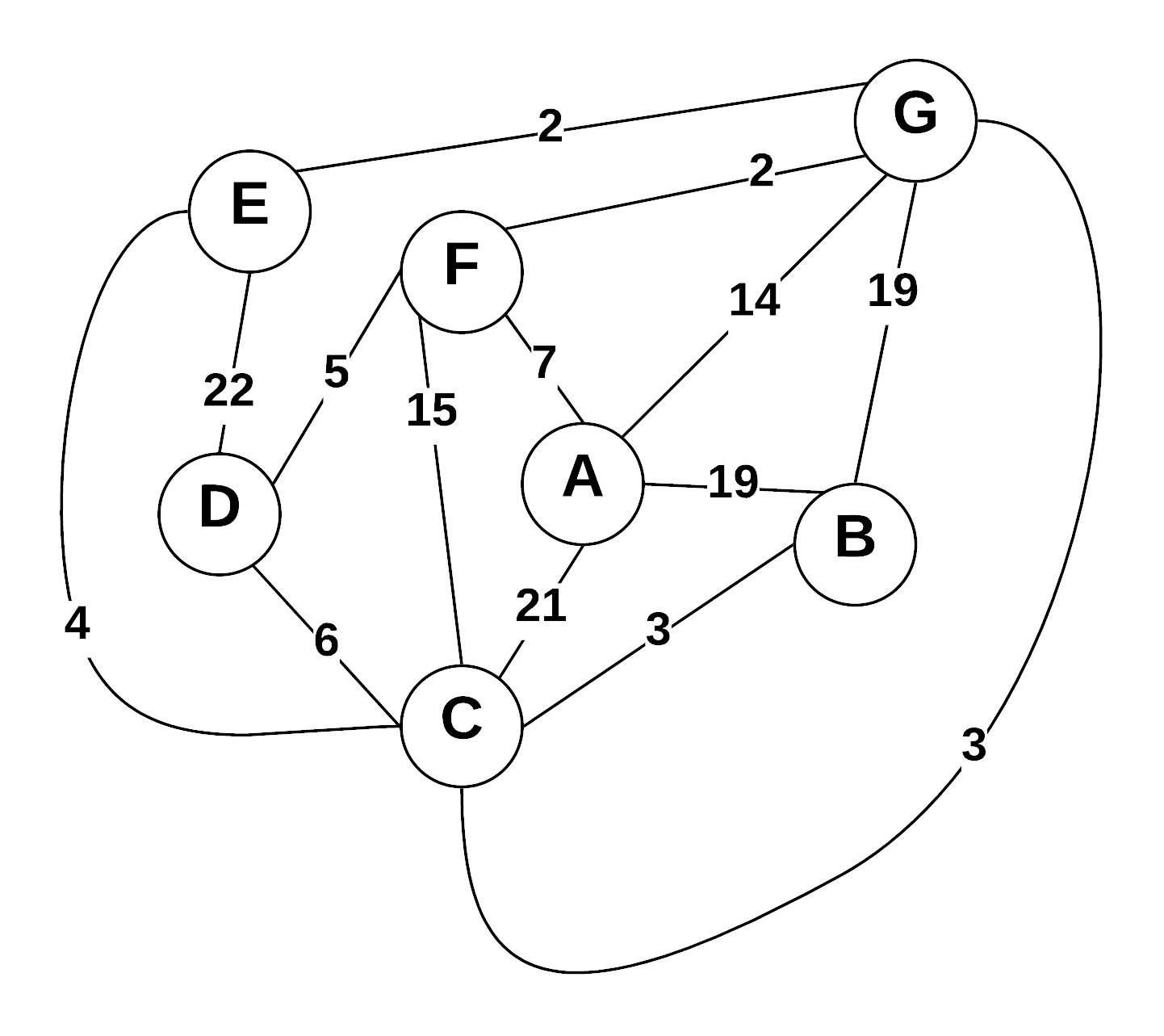
Vertices in Order of Discovery:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| G | E | F | D | C | B | 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) |
|  | A B C D E F G |  |
| G | A B C D E F | (E G) , (F G) |
| G E | A B C D F | (D E) , (F G) |
| G E F | A B C D | (A F) , (C F) , (D E) , (D F) |
| G E F D | A B C | (A F) , (C D) , (C F) |
| G E F D C | A B | (A C) , (A F) , (B C) |
| G E F D C B | A | (A C), (A F) |
| G E F D C B A |  |  |

**3) Kruskal’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 start 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 |
|  |  | 0 | (A) (B) (C) (D) (E) (F) (G) |
| (F G) | 2 | 2 | (A) (B) (C) (D) (E) (FG) |
| (E G) | 2 | 4 | (A) (B) (C) (D) (EG) (FG) |
| (C G) | 3 | 7 | (A) (B) (CG) (D) (EG) (FG) |
| (B C) | 3 | 10 | (A) (BC) (CG) (D) (EG) (FG) |
| (D F) | 5 | 15 | (A) (BC) (CG) (DF) (EG) (FG) |
| (A F) | 7 | 22 | (AF) (BC) (CG) (DF) (EG) (FG) |
|  |  |  |  |

**4) Prim’s Vs Kruskal’s**

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?

A graph is dense when it has lots of edges(E) between its vertices(V) and sparse when there are few edges.

Kruskal’s algorithm creates a minimum spanning tree by adding edges one at a time. The first step in Kruskal’s algorithm is to sort every edge by its weight in increasing order; this operation takes O(E\*logE) time complexity. In the second step, the smallest edge is picked and checked to see if it forms a cycle in the tree. If it does not, then the edge is added, otherwise the edge is discarded. Step two is repeated until there are (V -1) edges in the tree. This step takes O(E\*logV) time complexity. This makes the total time complexity to be O(E\*logE + E\*logV), but E can at most be V2 so that makes O(logE) and O(logV) the same. Now, the total time complexity is either O(E\*logE) or O(E\*logV). This algorithm is implemented using a disjoint set data structure, which is simple compared to some more complex options of Prim’s.

Prim’s algorithm creates a minimum spanning tree by adding vertices one at a time. The algorithm creates an adjacency matrix of nodes and an array of minimum distances from a vertex to a node. Since the array contains the minimum distances, all we ever need to do is check the array to find the next target, rather than iterate through all the edges every time. The search for all adjacent vertices takes O(V) worst case, and extracting the minimum value takes O(V). This makes total time complexity for one vertex to be O(V +V),which is equal to O(V). This must be done V number of times, thus making time complexity O(V \* V) or O(V2).This time complexity applies to Prim’s algorithm’s adjacency matrix representation; however, the algorithm can be improved by using different data structures. A binary heap implementation will run in time complexity O(E\*logV) and a Fibonacci heap which will yield time complexity O(E + V\*logV).

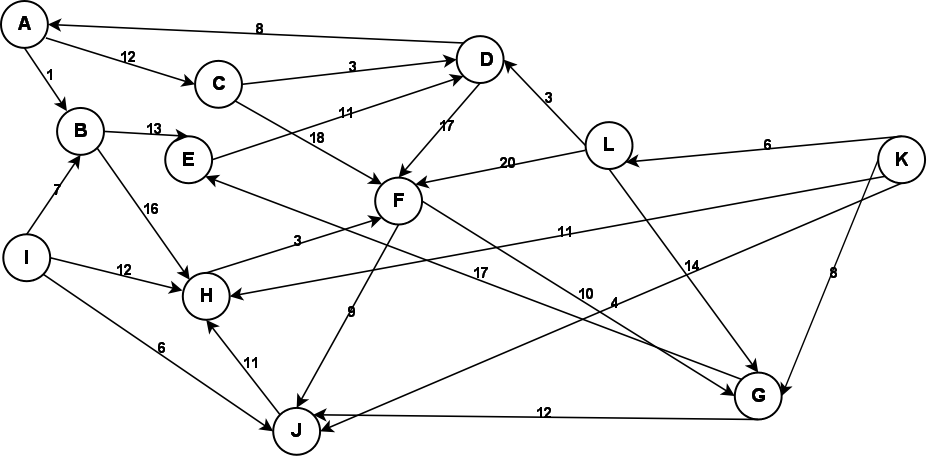
If we compare Kruskal’s and Prim’s time complexities, then we can see in which situations they are optimal. If the graph is dense(E = V2), then the time complexity of Kruskal’s algorithm is worse than Prim’s algorithm. If the time complexity of Prim’s is O(V2) and there are V2 edges, then Kruskal’s algorithm must be slower than Prim’s in a dense graph because it multiplies the result of Prim’s algorithm by logV. The edges need to be sorted in Kruskal’s algorithm and become expensive when they cannot be sorted in linear time or are not already sorted, thus making Prim’s more efficient in dense graphs. In a sparse graph(E = V), the time complexity for Kruskal’s algorithm is better than Prim’s algorithm. V2 results in a significantly bigger number than logV. Since E = V, Kruskal’s algorithm must perform faster because (V \* some number less than V)

is less than (V \* V).

**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 a technique that makes the best choice at a given time and builds a solution piece-by-piece. It makes micro decisions rather than macro decisions. The idea of a greedy algorithm is that a local best choice will result in a globally optimal solution.
5. Prim’s algorithm is a greedy algorithm. The algorithm only makes choices based on a subset of nodes while ignoring the rest of the graph. Because it does not look at the entirety of the graph at once, it must make the best choice one-by-one as the algorithm continues. Prim’s algorithm performs in O(V2) because we must loop through every vertex V number of times to form the minimum spanning tree.
6. Greedy algorithms can give optimal solutions, but only if the problem contains two traits: an optimal substructure and a greedy-choice property. An optimal substructure is an optimal solution that is made up of other optimal solutions to subproblems. A greedy-choice property is when a globally optimal solution can be reached by making the best choice one at a time. If the problem does not have these two characterize then greedy algorithms cannot be assured to give the most optimal solution.

**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 |  |  |  |

**Breadth First:**

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

I used the iterative solution for both the Breadth first and Depth first traversals. In-going degrees were ignored; only out-going edges were chosen to be traversed. Lower weighted edges were added to the data structure in such a way that they would be popped off before the other heavier adjacent vertices. Vertices I,K, and L are unreachable. Vertices I and K only have out-going edges, meaning there is no way to get to them. Vertex L can only be reached through K which is isolated.

**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* .

int\*\* reverse(int \*\*adjM, int n)

{

int \*\*a;

This function takes two parameters: a 2d array and the number of nodes in the graph. The old graph is transposed to a new one that gets returned

a = new int\* [n];

//initializes 2D array

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

{

a[i] = new int [n];

}

//transposes matrix

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

{

for(int j = 0; j < n; j++)

{

a[j][i] = adjM [i][j];

}

}

return a;

}

**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*().

bool Graph::isTree()

{

//stack of ints

This is a method for an assumed Graph class represented by an adjacency matrix. There are no parameters since this method is embedded in the class and executes on an already constructed graph object. Vertices are ints in this case. Stack library is assumed to be included. A depth first search is executed to make sure there are no cycles, and it marks vertices as visited through traversal of graph. If there were no cycles, then it checks if all vertices were visited. If not all vertices were visited then the graph must have disconnected vertices and wouldn’t be a tree, otherwise the method returns true.

stack<int> s;

//pushes first vertex on stack

s.push(0);

//sets all vertices to false

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

{

visited[i] = false;

}

while (!s.empty())

{

int v = s.top();

s.pop();

//If popped vertex is visited, then there is a //cycle

if (visited[v])

{

return false;

}

//Changes visit status to true

visited[v] = 1;

//Add to stack if:

// 1.edge exists between parent and current node.

// 2.current node is not already visited

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

{

//If no edge, value is assumed to be 0

if (adj[v][i] != 0 && !visited[i])

s.push(i);

}

}

//If there is vertex not visited, then graph

//is not tree. Can't have disconnected vertices

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

{

if (!visited[i])

return false;

}

return true;

}

**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.

abcdefgh:41

abcdefg:33

h:21

g:13

abcdef:20

f:8

abcde: 12

e:5

abcd:7

d:3

abc:4

ab:2

c:2

a:1

b:1

**(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.)

cab

**10) Bellman Ford (Optional)**

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

ITERATION 1

For this problem I assumed we start from vertex A. The left-side table represent the current vertex that is updating its adjacent vertices. Vertices were selected in alphabetical order.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
|  | 19 |  |  |  | 7 | 14 |
|  |  |  |  |  |  |  |
|  |  |  | 27 | 25 |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  | 9 |
|  |  | 12 | 12 | 11 |  |  |

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

Array after iteration 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 19 | 12 | 12 | 11 | 7 | 9 |

ITERATION 2

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

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

Array after iteration 2:

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

ITERATION 3

Nothing changed in iteration 3 so this is where we stop. Final distances to everything from vertex is A is in iteration 3 array.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 19 | 12 | 12 | 11 | 7 | 9 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Array after iteration 3:

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