**Test 3**

Name: Vincent Rodriguez

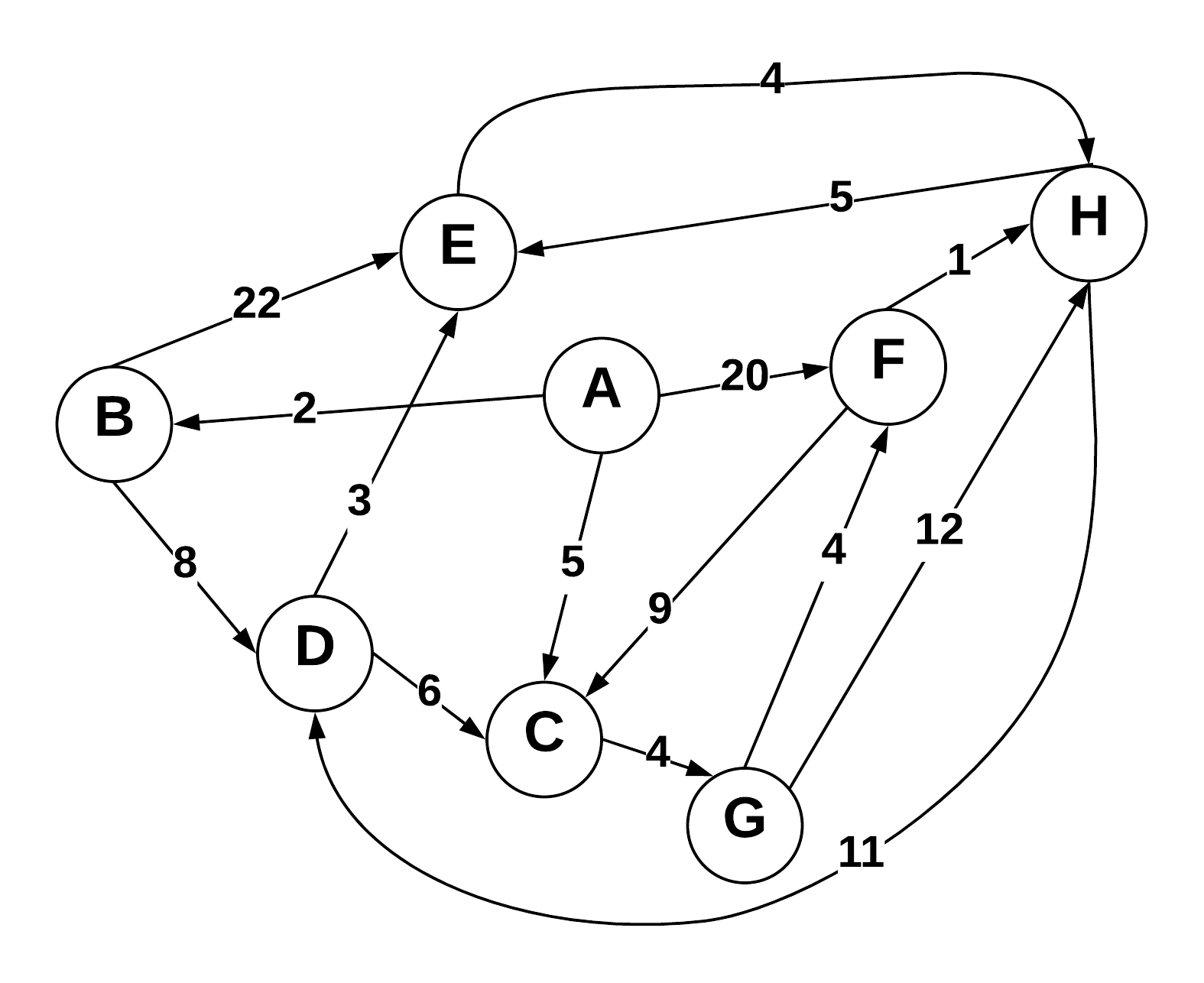
* Everything you turn in must be digitally created.
* No handwriting (except for signature below).
* You must work alone.
* Sharing of answers will result in a 0 on the exam, and possible F in the course.
* Send me your digitally created exam by Friday, May 4th by Midnight on a private slack message.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



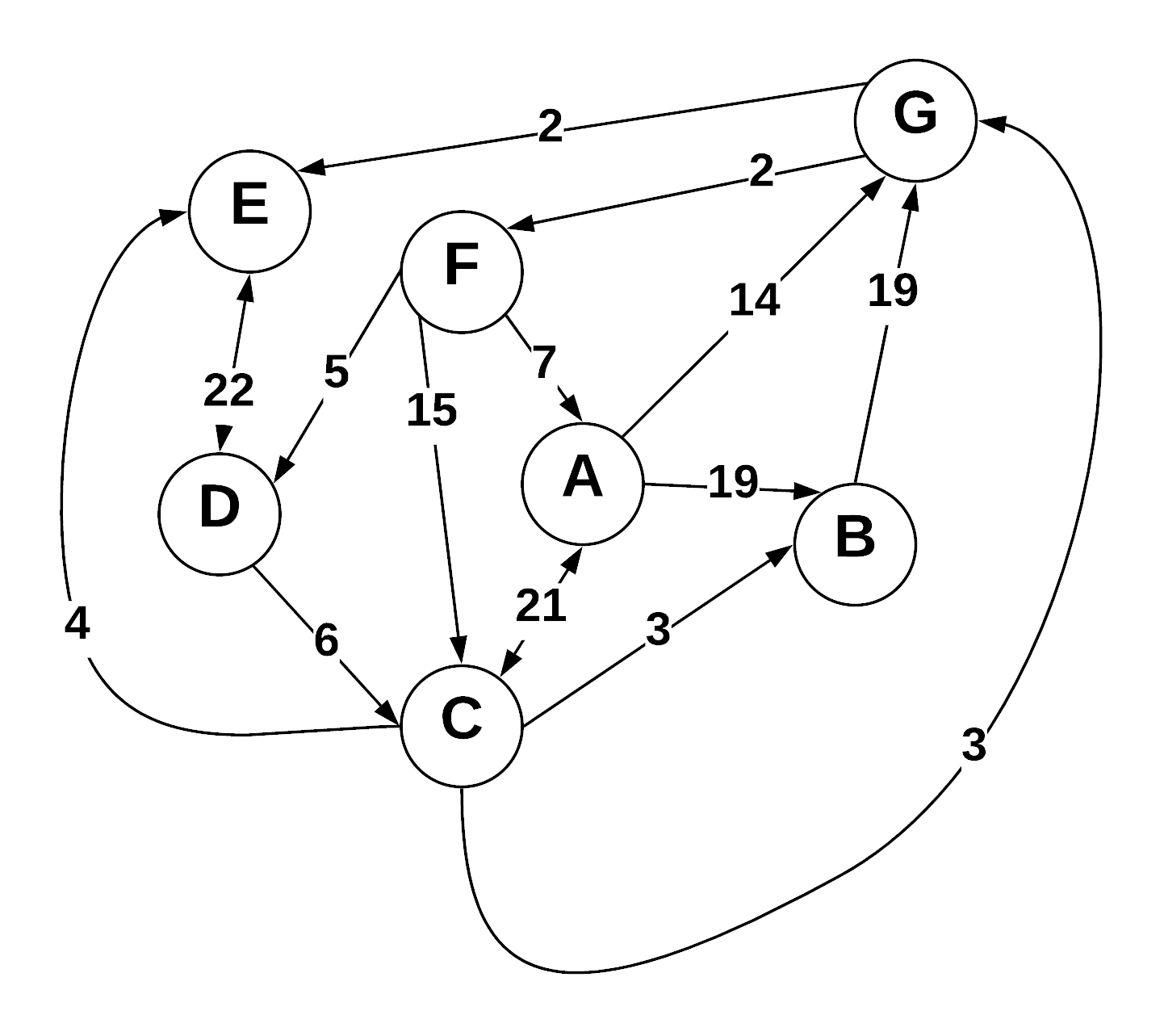
Use Dijkstra’s algorithm to compute the shortest paths from vertex A to every other vertex. Show your work in the space provided below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same 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 | 1 | 0 | — |
| B | 2 | 2 | A |
| C | 3 | 5 | A |
| D | 5 | 10 | B |
| E | 6 | 13 | D |
| F | 7 | 10 | G |
| G | 4 | 9 | C |
| H | 8 | 14 | F |

**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.

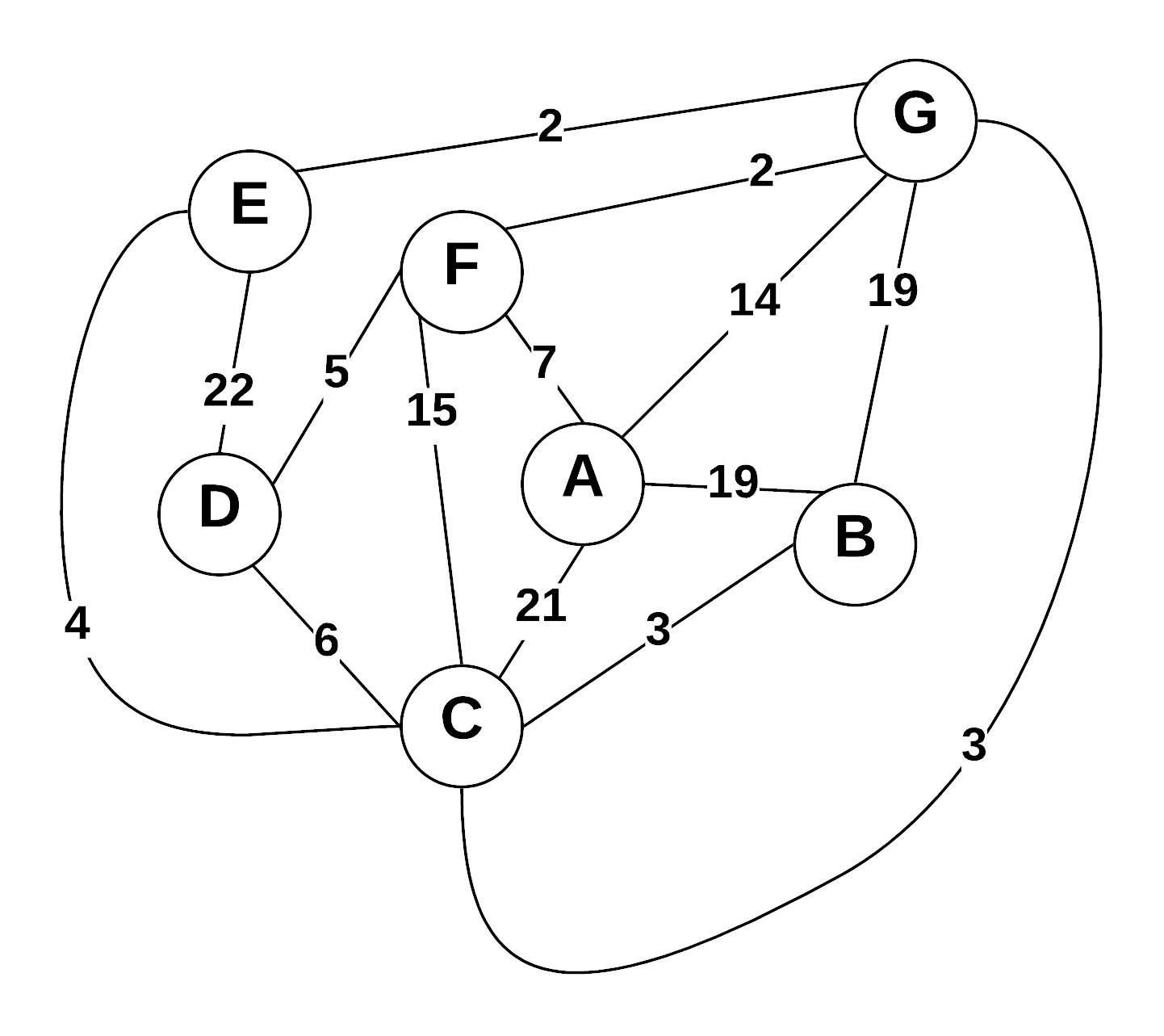
Vertices in Order of Discovery:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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) |
| {E,G} | {A,B,C,D,F} | (AG),(BG),(CG),(EG),(FG) |
| {E,F,G} | {A,B,C,D} | (AG),(BG),(CE),(CG),(DE),(FG) |
| {C,E,F,G} | {A,B,D} | (AF),(AG),(BG),(CE),(CF),(CG),(DE),(DF) |
| {B,C,E,F,G} | {A,D} | (AC),(AF),(AG),(BC),(BG),(CD),(DE),(DF) |
| {B,C,D,E,F,G} | {A} | (AB),(AC),(AF),(AG),(CD),(DE),(DF) |
| {A,B,C,D,E,F,G} | ᶱ | (AB),(AC),(AF),(AG) |
|  |  |  |
|  |  |  |

**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 |
| (FG) | 2 | 2 | (A)(B)(C)(D)(E)(FG) |
| (EG) | 2 | 4 | (A)(B)(C)(D)(EFG) |
| (CG) | 3 | 7 | (A)(B)(D)(CEFG) |
| (BC) | 3 | 10 | (A)(D)(BCEFG) |
| (FD) | 5 | 15 | (A)(BCDEFG) |
| (AF) | 7 | 22 | (ABCDEFG) |
|  |  |  |  |
|  |  |  |  |

**4) Prims 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?

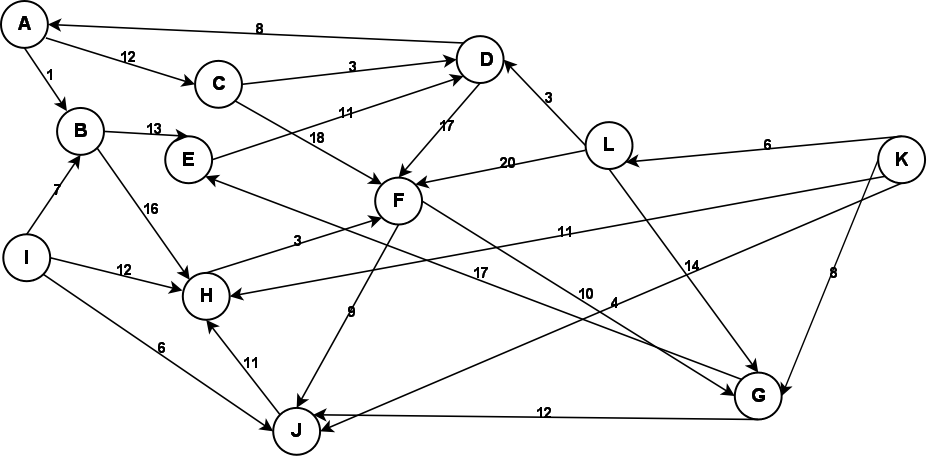
For Prim’s Algorithm, with many edges to choose from, it is more possible to connect all vertices that are not initially in the graph. Otherwise, less choices of edges available to connect the graph may not produce a minimum spanning tree. The data structure used to represent this algorithm is a priority queue.

For Kruskal’s Algorithm, which you select edges from all possible edges, the lesser the number of possible edges to choose from yields better lookup time. The data structure used to represent this algorithm is a union find, where all the edges are contained in a priority queue.

**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. Greedy Algorithm — A greedy algorithm is a technique that ensures the most efficient choice is made, at the specific point in which the code is being executed. However, in an overall/global sense, the choice is made which is most optimal for the time being and not for resulting optimal solution.
5. A choice greedy algorithm would be Dijkstra’s Algorithm. To explain its greediness, we can talk about the selection process. Dijkstra’s Algorithm executes code to choose the shortest path from an original vertex to the next closest path, i.e. the shortest edge value to the next vertex. Hence, for each new vertex chosen, the shortest path to each one is a minimal distance. However, this may not perform best, because the algorithm only considers new vertices already connected to the specific path. In a specific path created by choice connected vertices, the algorithm must choose which vertex will yield a minimal weight distance. So local performance from vertex to vertex will be optimal. However, such a path may not be the only minimal spanning tree result, nor may be the minimalist one either. What if another path considered the same weight, and the algorithm had to choose between same weights, what would happen? Simply put, it would have to be hardcoded as instruction to determine which path was to be taken. Hence, although the local efficiency is optimal, the global efficiency might be determined to be a less than minimal spanning tree and therefore not the most efficient.
6. As in the above example, it depends on the implementation of the greedy algorithms, whether all possible optimal outcomes on the local level are truly optimal on the global level. Most greedy algorithms will produce an optimal result, but only if the logic is truly thought through.

**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 | G | J | H | C |  |  |  |

**Breadth First:**

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

**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){

//variable m is assumed to be the number of edges in the adjacency matrix

int numEdges = m;

char \*edgeArray = new char[m];

**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 isTree ( ){

//Vertex[] is array that holds vertices in graph

//vertexPtr is an integer pointer that points to vertices

//vertices have parent, left child, right child

int \* vertexPtr = NULL;

//each vertex is primarily marked as unvisited

for (int x = 1; x <= G.numVertices; x++){

Vertex[x].isVisited = false;

}

//starts traversing graph at each vertex and checks if all vertices are reachable

for (int x = 1; x <= G.numVertices; x++){

//vertexCtr counts the number of vertices

int vertexCtr = 0;

int \* vertexPtr = Vertex[x];

Vertex[x].isVisited = true;

for (int y = 1; x <= G.numVertices – 1; x++){

if (vertexPtr->left && vertexPtr->left != vertexPtr->right){

vertexPtr = vertexPtr->left;

vertexCtr++;

} else { return false; }

vertexPtr = Vertex[x];

if (vertexPtr->right && vertexPtr->right != vertexPtr->right){

vertexPtr = vertexPtr->right;

vertexCtr++;

} else { return false; }

}

**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.)

A)

B) h: 0

g: 10

f: 110

e: 1110

d: 11110

c: 111111

b: 1111101

a: 1111100

String: 111111 1111100 1111101

Answer: C A B

X 3 d X

X 1 b X

X 1 a X

X 2 c X

X 5 e X

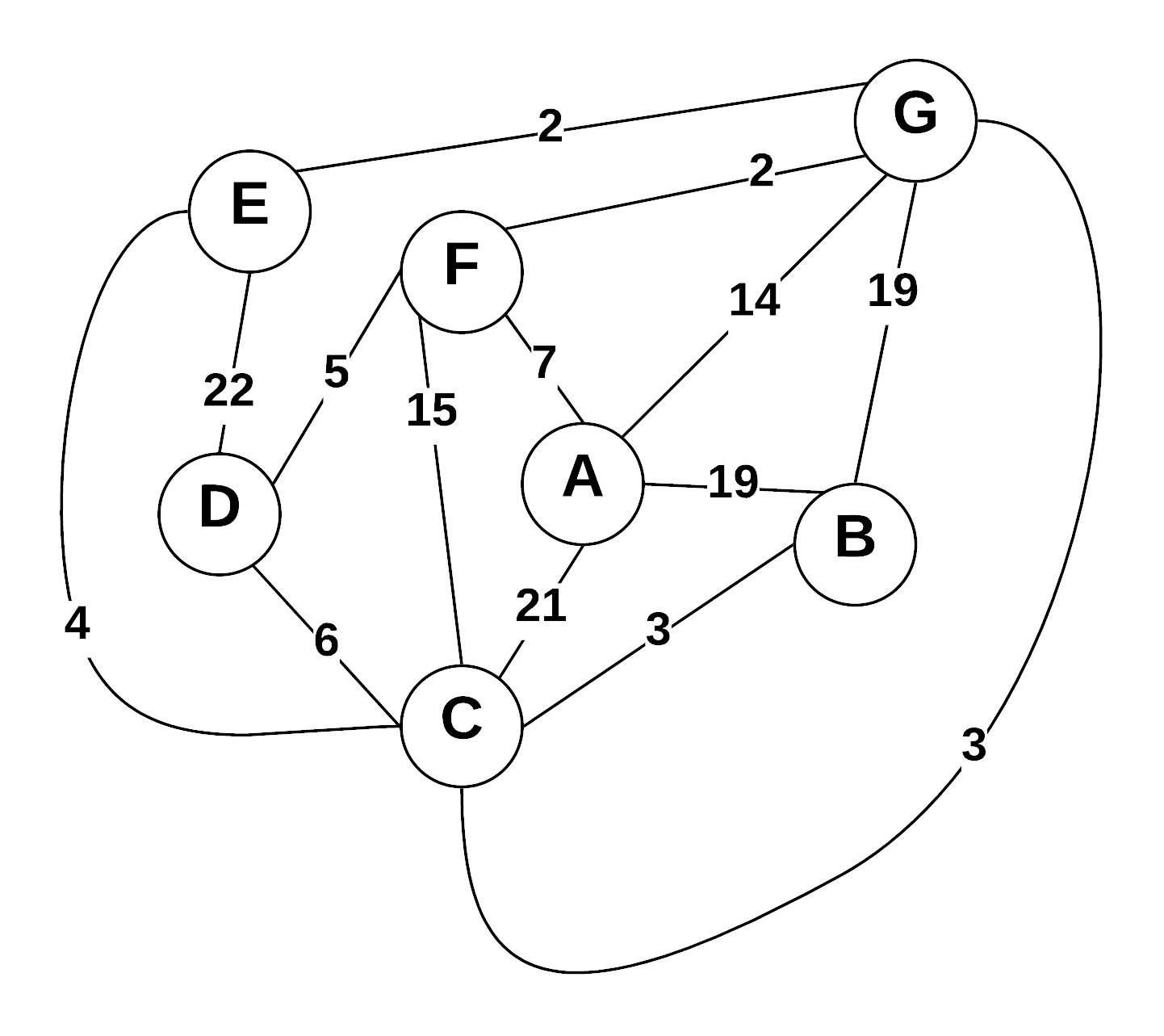
X 8 f X

X 13 g X

X 21 h X

**10) Bellman Ford (Optional)**

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



Vertex Distance to V

A 0

B 19

C 21

D 12

E 25

F 7

G 9