Abdullah Tahir
21L-5419
Section: BCS-5D

Assignment # 4

Graphs

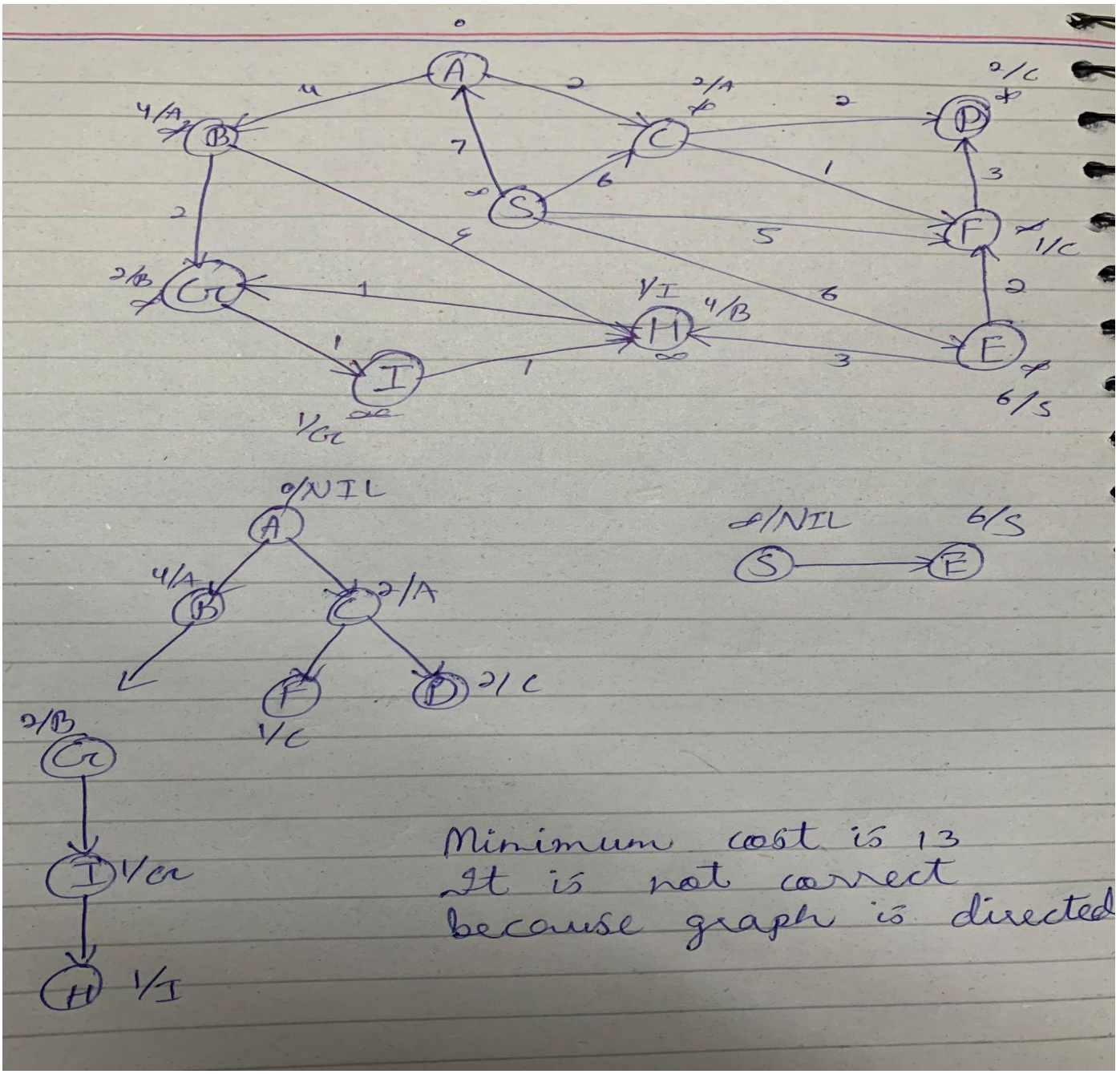
Problem 1. Dry Runs

(1) Draw the MST of the graph given below, take node A as the source in Graph and break the tie in numerical order.

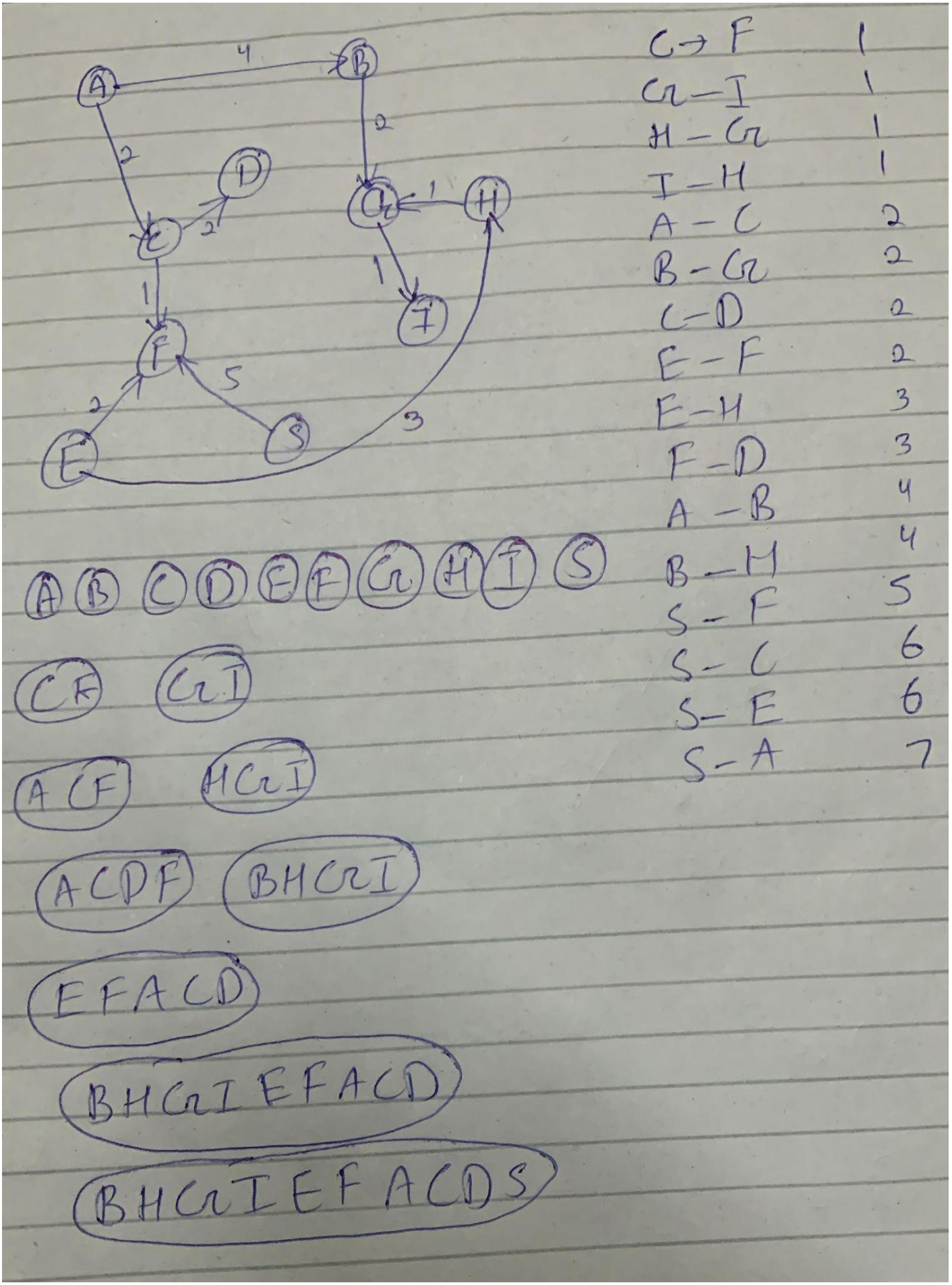
a. Prim's Algorithm **b. Kruskal's Algorithm**

Solution:

(a) Prim's Algorithm

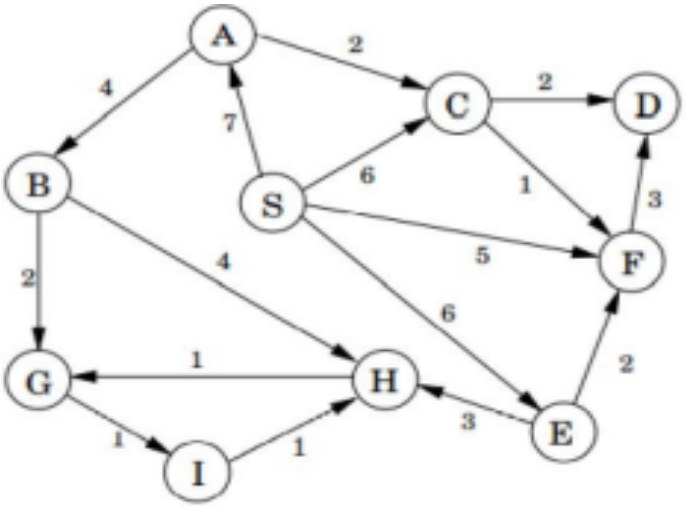


(b) Kruskal's Algorithm: MST using Kruskal's Algorithm (with tie-breaking in numerical order):

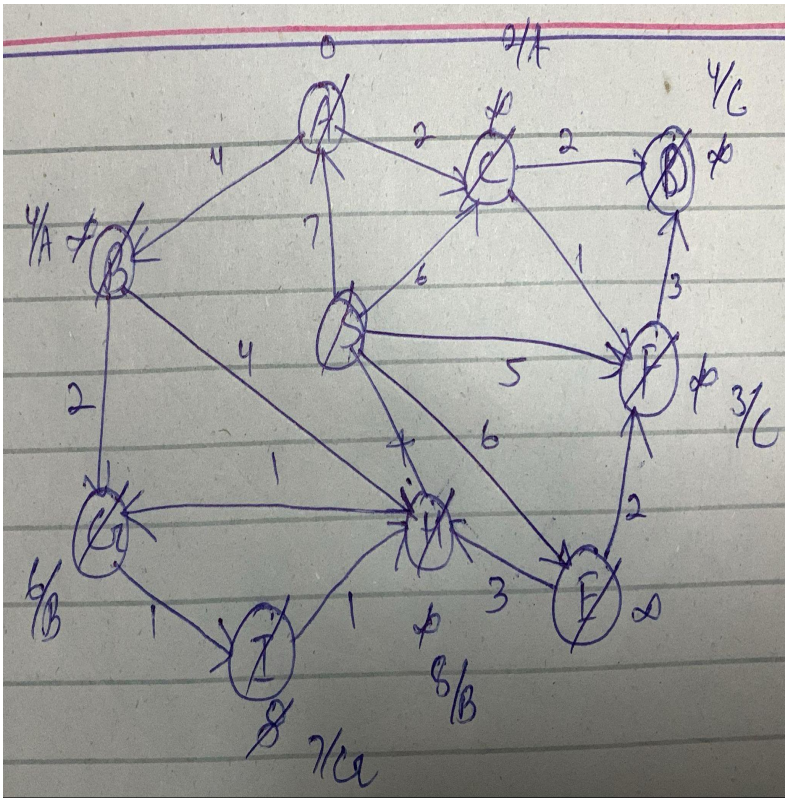


(2) Dry run Dijkstra's Algorithm on Graph given below. Show the node extracted from the min heap in each iteration of the algorithm; also, show the distance of extracted node from the source and its parent node. If two or more nodes in the heap have the same distance value which is minimum, extract the one which is alphabetically first. Take node A as the source in Graph and break the tie in numerical order.

For example, in Graph I below, the first output should be: [A, 0, nil], which means node A has distance 0 from the source and its parent is nil. Similarly, the second output is [B, 1, A], and so on.

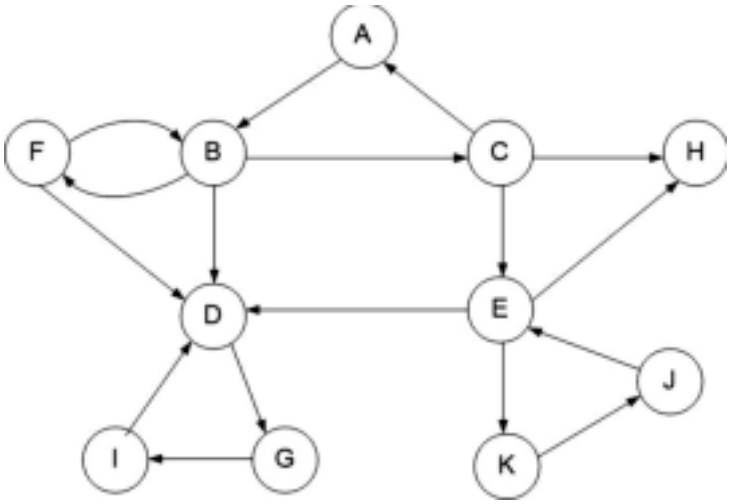


Solution:

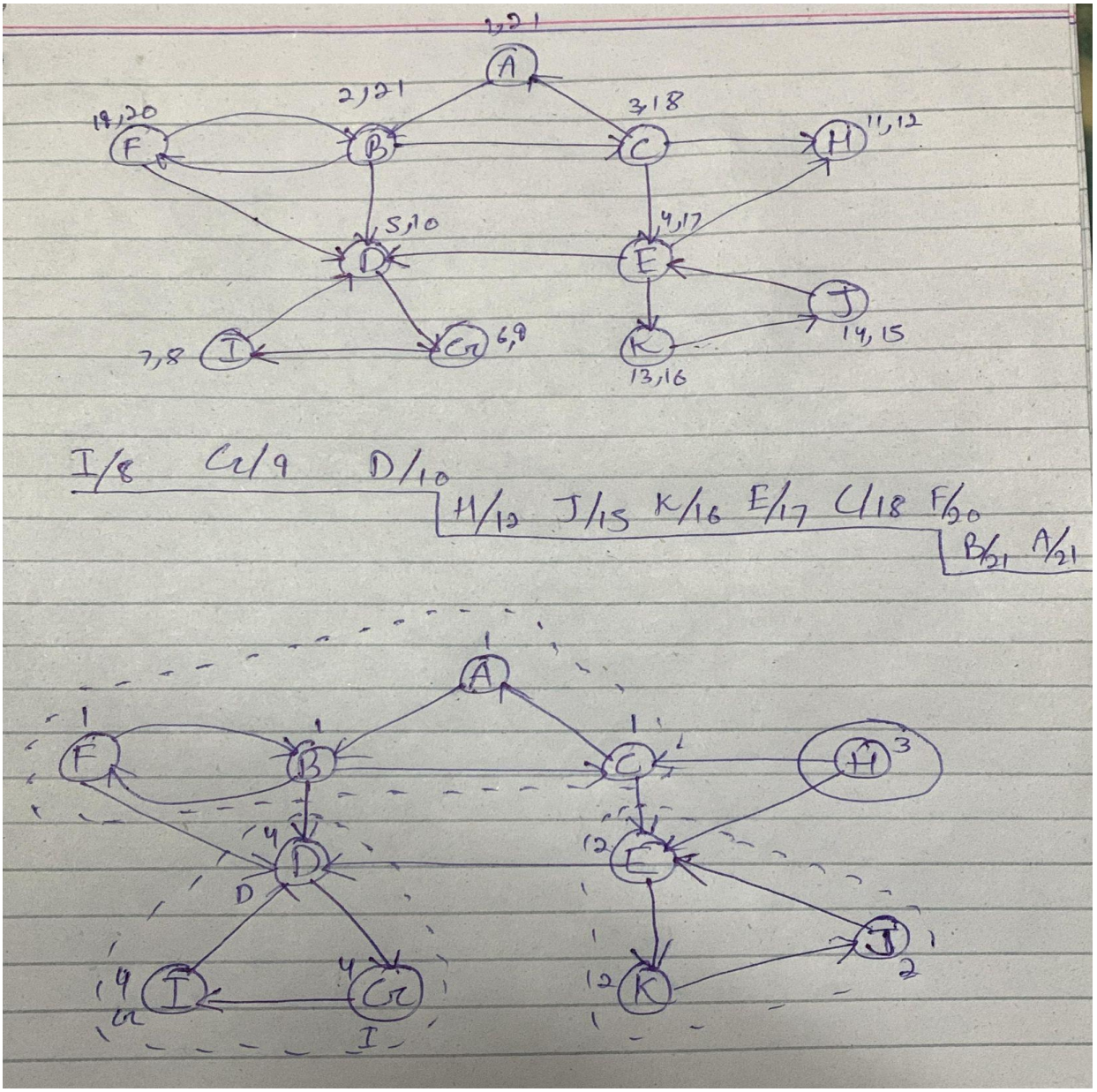


Extracted Node	Extracted Heap
A	[A, 0, nil]
C	[C, 2, A]
F	[F, 3, C]
B	[B, 4, A]
D	[D, 4, C]
G	[G, 6, B]
I	[I, 7, G]
H	[H, 8, B]
E	[E, ∞, Nil]
S	[S, ∞, Nil]

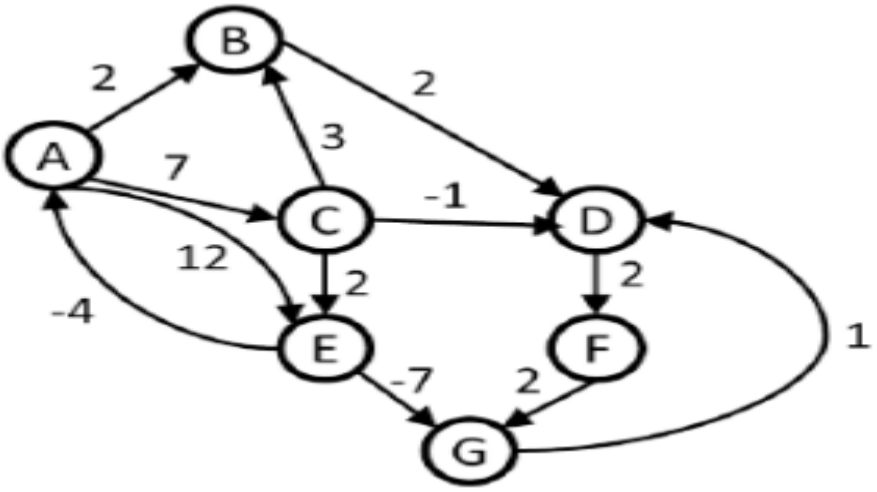
(3) Determine the strongly connected components of the graph using the algorithm discussed in class. Show all workings. Show finish times of both iterations of DFS.



Solution:



(4) Consider the following directed, weighted graph:



a. Even though the graph has negative weight edges. Execute Dijkstra's algorithm to calculate shortest paths from A to every other vertex

Solution:

A	B	C	D	E	F	G
AB	2	7	∞	12	∞	∞
ABC	2	7	4	12	∞	∞
ABCD	2	7	4	12	6	∞
ABCDE	2	7	4	12	6	8
ABCDEF	2	7		9	6	8
ABCDEFG				9		

b. Dijkstra's algorithm found the wrong path to some of the vertices. For just the vertices where the wrong path was computed, indicate both the path that was computed and the correct path.

Solution:

Shortest distance fromA -> E = 9 through C
When -4 is taken
A -> E = 9 - 4 -> 5 A - C -> E -> A + E
This cycle ends up in a never ending cycle.
A -> E computes wrong
and also the path
A -> G computes wrong
In our question
A -> G = 8 through A -> B -> B -> F -> G
One more shortest path is found
A -> G = 2 through A -> C -> E -> G

c. What single edge could be removed from the graph such that Dijkstra's algorithm would happen to compute correct answers for all vertices in the remaining graph?

Solution:

Removing the **E -> A** edge with a weight of **-4** from Dijkstra's algorithm might work correctly for all vertices. It will remove the negative weight edge, potentially leading to correct path calculations for all vertices.

d. Execute the Bellman ford Algorithm to find the correct shortest paths.

Solution:

Relaxing each vertex v - 1 times to find shortest path

*	A	B	C	D	E	F	G
	0	∞	∞	∞	∞	∞	∞
A	0	2	7	∞	12	∞	∞
B	0	2	7	4	12	∞	∞
C	0	2	7	4	9	∞	∞
D	0	2	7	4	9	6	∞
E	0	2	7	4	9	6	2
F	0	2	7	4	9	6	2
G	0	2	7	3	9	6	2

*	A	B	C	D	E	F	G
	0	2	7	3	9	6	2
A	0	2	7	3	9	6	2
B	0	2	7	3	9	6	2
C	0	2	7	3	9	6	2
D	0	2	7	3	9	5	2
E	0	2	7	3	9	5	2
F	0	2	7	3	9	5	2
G	0	2	7	3	9	5	2

*	A	B	C	D	E	F	G
	0	2	7	3	9	5	2
A	0	2	7	3	9	5	2
B	0	2	7	3	9	5	2
C	0	2	7	3	9	5	2
D	0	2	7	3	9	5	2
E	0	2	7	3	9	5	2
F	0	2	7	3	9	5	2
G	0	2	7	3	9	5	2

No change in 3rd relaxation so it is the shortest path from A to all other vertices.

Problem 2. Design

(1) DFS: Modify the pseudocode for depth first search so that it prints out every edge in the directed graph G, together with its type. Show what modification, if any, you need to make if G is undirected.

Solution:

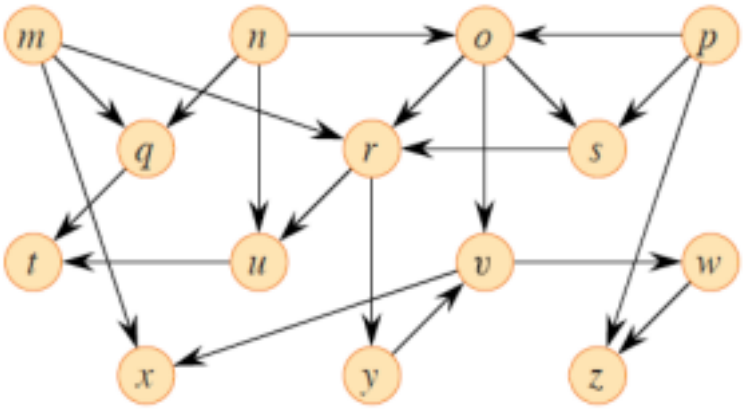
```
DFS(G):
  for each vertex v in G:
    v.visited = false

  for each vertex v in G:
    if not v.visited:
      DFS-Visit(G, v)

DFS-Visit(G, v):
  v.visited = true
  for each neighbor w of v in G:
    if not w.visited:
      print("Edge: " + v + " -> " + w + " (Directed)")
      DFS-Visit(G, w)
    else:
      print("Edge: " + v + " -> " + w + " (Backward)")
```

Changes:

```
DFS-Visit(G, v):
  v.visited = true
  for each neighbor w of v in G:
    if not w.visited:
      print("Edge: " + v + " -> " + w + " (Tree/Forward)")
      DFS-Visit(G, w)
    else:
      if w is a descendant of v in the DFS tree:
        print("Edge: " + v + " -> " + w + " (Backward)")
      else if v is a descendant of w in the DFS tree:
        print("Edge: " + v + " -> " + w + " (Forward)")
      else:
        print("Edge: " + v + " -> " + w + " (Cross)")
```

(2) Topological Sort: Give a linear-time algorithm that, given a directed acyclic graph G = (V, E) and two vertices a and b, returns the number of simple paths from a to b in G. For example, the directed acyclic graph below contains exactly four simple paths from vertex p to vertex v:

p->o->v,
p->o->r->y->v
p->o->s->r->y->v
p->s->r->y->v
Your algorithm needs only to count the simple paths, not list them.



Solution:

Code:

```
class Graph
{
   struct Node
   {
      int data;
      Node* next;
   };
         int MAX_VERTICES = 100;
   int V;
   Node* array[MAX_VERTICES]; // Array of adjacency lists

public:
   Graph(int V)
   {
      this->V = V;
      for (int i = 0; i < V; ++i)
      {
         array[i] = nullptr;
      }
   }

   // Add an edge to the graph
   void addEdge(int src, int dest)
   {
      Node* newNode = new Node();
      newNode->data = dest;
      newNode->next = array[src];
      array[src] = newNode;
   }

   // Recursive function to count paths from 'src' to 'dest'
   int countPathsUtil(int src, int dest, bool visited[])
   {
      if (src == dest) // If source is same as destination
      {
         return 1;
      }
```

```
        visited[src] = true;
        int count = 0;

        Node* curr = array[src];
        while (curr != nullptr)
        {
            if (!visited[curr->data])
            {
                count += countPathsUtil(curr->data, dest, visited);
            }
            curr = curr->next;
        }

        visited[src] = false; // Mark current node as unvisited
        return count;
    }

    // Function to count simple paths from 'src' to 'dest'
    int countSimplePaths(int src, int dest)
    {
        bool visited[MAX_VERTICES] = { false }; // Initialize all vertices as not visited
        return countPathsUtil(src, dest, visited);
    }
};
```

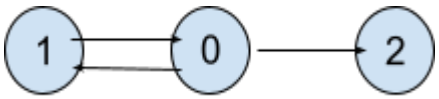
Explanation:

This **Graph** class defines a graph structure using adjacency lists, where **addEdge** adds directed edges between vertices. The **countSimplePaths** method utilizes a recursive **countPathsUtil** function to find the number of simple paths between **src** and **dest** vertices in the graph. It tracks visited nodes and accumulates the count while traversing through the graph, adhering to directed edges and marking visited nodes to ensure path uniqueness in a directed acyclic graph (DAG).

(3) Strongly connected component: Professor Bacon rewrites the algorithm for strongly connected components to use the original (instead of the transpose) graph in the second depth first search and scan the vertices in order of increasing finish times. Does this modified algorithm always produce correct results?

Solution:

Consider:



⇒ **DFS Finish Time Order:** DFS starting from 0 will result in the finish time order 8, 1, 2, 0

⇒ **Second DFS Execution:** The second execution of DFS shows all vertices as one strongly connected component, since both (0) and (2) are reachable from (1)

⇒ In this case, the DFS approach fails as there should be two components: {1,0} and {2} **so simpler algorithm does NOT always produce correct results.**

(4) Dijkstra's algorithm: Sometimes it is desirable not only to find the shortest path in a graph (assuming only positive weights here), but also among the shortest paths, the one that has the minimum number of edges. Modify Dijkstra's algorithm so that it makes sure that for each vertex t, the shortest path we get from s (source) to t, is the one with the minimum cost but also, among all the minimum cost paths, the one with the minimum number of edges. Take the pseudo-code from class and add your additional code to it. Also explain your logic in a few lines of English.

Solution:

DijkstraShortestPathWithFewestEdges(graph, source):

1. Initialize distances array with INF for all vertices except source (distance[source] = 0)
2. Initialize edges array with INF for all vertices except source (edges[source] = 0)
3. Initialize Min Heap with source vertex and its distance (source, 0)

4. while Min Heap is not empty:
   u = ExtractMin(Min Heap) // Extract vertex with minimum distance value
   for each neighbor v of u:
      if distance[u] + weight(u-v) < distance[v]:
         distance[v] = distance[u] + weight(u-v) // Update distance value
         edges[v] = edges[u] + 1 // Update number of edges in shortest path
         Update Min Heap with new distance value of v

   return distances, edges

This algorithm starts by initializing arrays for distances and edges, setting the source vertex's distance to 0 and edges to 0. Utilizing a min heap, it explores vertices iteratively, updating their distances and the number of edges in the shortest path if a shorter path is found. This continues until the heap is empty, ultimately returning the arrays containing the shortest distances and the fewest edges from the source to each vertex in the graph.

(5) Given a directed graph where every edge has weight as either 1 or 2, give an algorithm to find the shortest path from a given source vertex 's' to a given destination vertex 't'. Expected time complexity is O(V+E).

Solution:

```
class Graph
{
        struct Node
        {
                int dest;
                int weight;
                Node* next;
        };
        int MAX_VERTICES = 100;
        int INF = 1000000;
        Node* array[MAX_VERTICES]; // Array of adjacency lists
        int V;
```

```
public:
    Graph(int V)
    {
        this->V = V;
        for (int i = 0; i < V; ++i)
            array[i] = nullptr;
    }

    // Add an edge to the graph
    void addEdge(int src, int dest, int weight)
    {
        Node* newNode = new Node();
        newNode->dest = dest;
        newNode->weight = weight;
        newNode->next = array[src];
        array[src] = newNode;
    }

    // Find shortest path from source to destination
    int shortestPath(int src, int dest)
    {
        int dist[MAX_VERTICES];
        for (int i = 0; i < V; ++i)
            dist[i] = INF;

        queue<int> q;
        dist[src] = 0;
        q.push(src);

        while (!q.empty())
        {
            int u = q.front();
            q.pop();

            Node* temp = array[u];
            while (temp != nullptr)
            {
                int v = temp->dest;
                int weight = temp->weight;

                if (dist[v] > dist[u] + weight)
                {
                    dist[v] = dist[u] + weight;
                    q.push(v);
                }

                temp = temp->next;
            }
        }
        return dist[dest];
    }
```

};