Homework 3

CIS-675 DESIGN AND ANALYSIS OF ALGORITHMS

PROF. IMANI PALMER

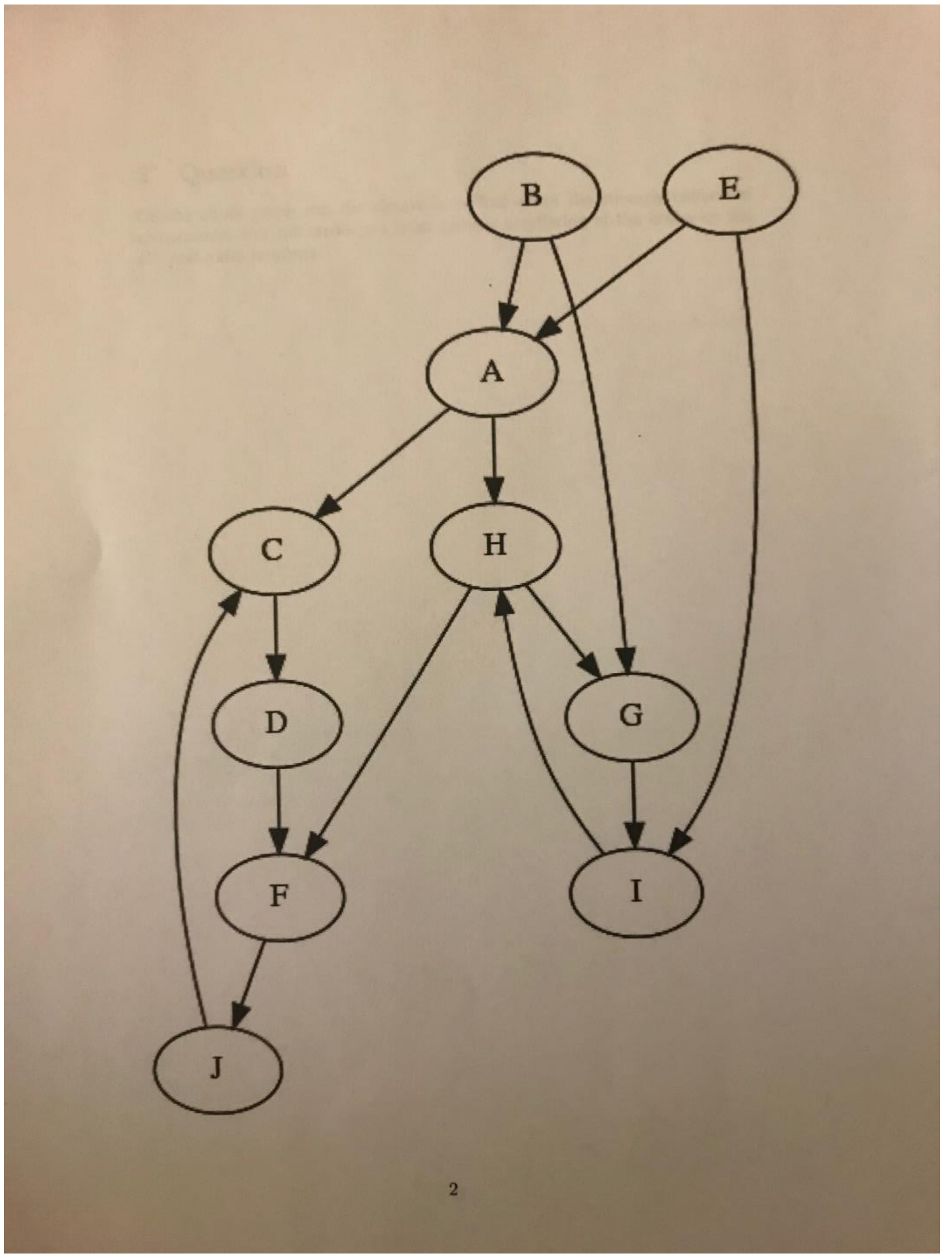
5/3/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

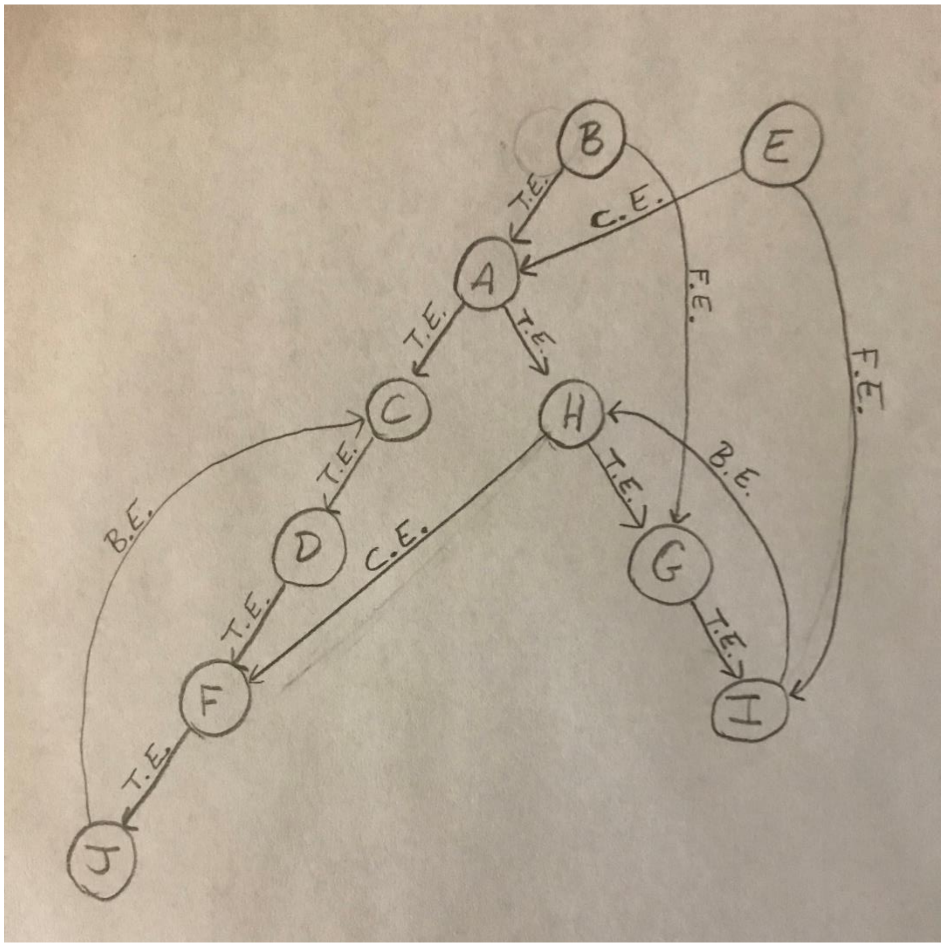
Question 1:

Run a Depth First Search on the following graph (with restarting)

- Show the post/pre visit numbers
- Draw the DFS tree with tree edges, forward edges, back edges and cross edges clearly labeled.



The graph’s DFS tree is below with tree edges, cross edges, back edges, and forward edges labeled T.E., C.E., B.E. and F.E. respectively.



Starting at Vertex B and traveling in alphabetical order when possible:

Vertex	Pre-order	Post-order
B	1	18
A	2	17
C	3	10
D	4	9
F	5	8
J	6	7
H	11	16
G	12	15
I	13	14

Question 2:

On the above graph run the algorithm to find all the strongly connected components. For full credit, you must provide an ordering of the nodes by the G^R post-visit numbers.

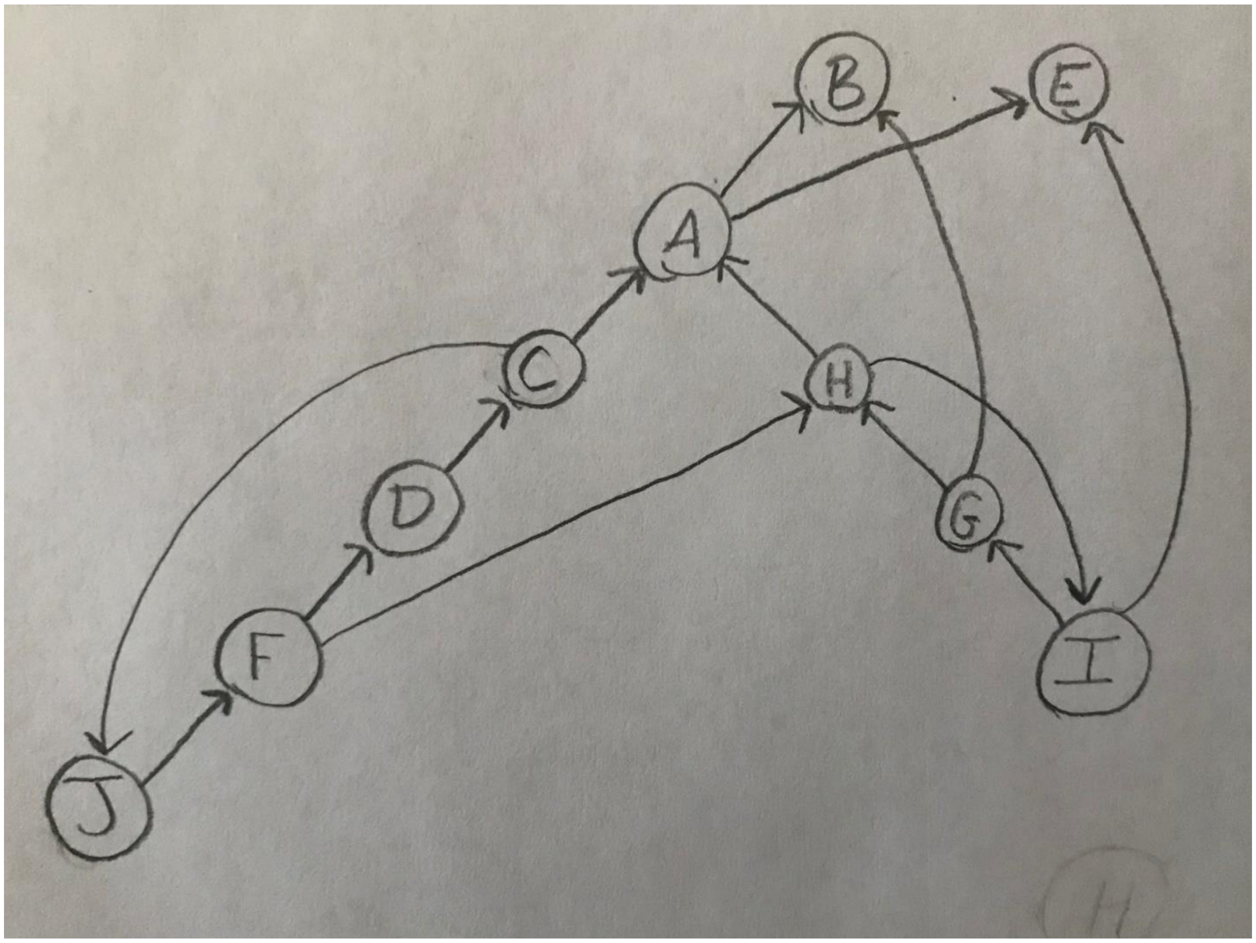
From Question 1, the DFS search yields the following pre and post-visit numbers starting at vertex B:

Vertex	Pre-visit	Post-visit
B	1	18
A	2	17
C	3	10
D	4	9
F	5	8
J	6	7
H	11	16
G	12	15
I	13	14
E	19	20

The ordering of the post-visit numbers yields the following order on the stack.

Stack
B
A
H
G
I
C
D
F
J

Now create G^R, where G^R is a directed graph with all edge directions reversed.



Pop the elements off the stack and perform DFS on them in G^R.

Vertex	Pre-visit	Post-visit
B	1	2
A	3	4
H	5	10
I	6	9
G	7	8
C	11	18
J	12	17
F	13	16
D	14	15

It is clear from the results of the algorithm that the following are the five strongly connected components in the graph:

- E
- B
- A
- {H,I,G}
- {C,J,F,D}

Question 3:

A bipartite graph is a graph G = (V,E) whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ *and* $V_1 \cap V_2$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$ then there is no edge between u and v).

a) Give a linear-time algorithm to determine whether an undirected graph is bipartite.

The below algorithm uses a breadth first search and has a runtime $O(E)$ where E is the number of edges in the graph.

Given the constraints of a bipartite graph, an effective strategy would be to execute a breadth first search on G starting at any vertex, V. Assign V with a color "red" to signify that it is part of the set RED. Assign each of its neighbors a color "blue" to signify that they are part of the set BLUE. Execute a BFS on each vertex in the graph and assign its neighbor to the opposite set as the vertex. If it is found that a vertex and its neighbor are assigned to the same set, the graph is not bipartite.

Input: Graph G(G, s), where graph G = (V,E)

Output: Boolean value (True/False) representing if the graph is bipartite or not.

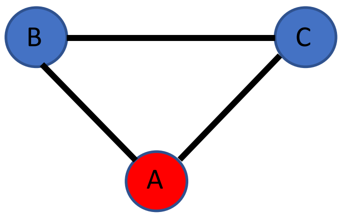
If s is a vertex in G, then u is an adjacent vertex to s also in G.

```
$boolean\ IsBipartite(G, s)\{$
    $for\ all\ u \in V:$
        $color(u) = unassigned;$
    $color(s) = red;$   $//\ initialize\ s\ with\ a\ color$
    $Q = [s];\ //\ queue\ containing\ just\ s$
    $while\ Q\ is\ not\ empty\ \{$
        $u = eject(Q);$
        $for\ all\ edges\ (u, v) \in E\ \{$
            $if\ (color(v) == unassigned)\{$
                $if\ (color(u) == Red)\ \{color(v) = Blue;\}$
                $else\ \{color(v) = Red;\}$
                $inject(Q, v);\ continue;$
                $\}$
            $if\big(color(v) == color(u)\big)\ \{return\ false;\}$
        $\}$
    }
    return true; // if no colors match then the graph is bipartite
}
```

Below is the implementation of the algorithm written in C:

```c
#include <stdio.h>
#define N 3

// ------------------------------ STRUCTS ------------------------------------//

typedef struct listNode {
   int value;
   struct listNode* next;
} Node;

typedef struct list{
   Node* head;
} List;

typedef struct queue_list{
   List list;
} Queue_List;

// ----------------------------- QUEUE LIST FUNCTIONS ---------------------------
-----//
void enqueue_list(List** head, int value){
   Node* new_node = NULL;
   Node* last_node = NULL;
   new_node = (Node*)(malloc(sizeof(Node)));
   new_node->value = value;
   new_node->next = NULL;
   // if list is empty insert new node at the head.
   if(*head == NULL){//if address for head node is NULL
      *head = new_node;
      return;
   }
   // start at the head looking for the last node, then insert new node after it.
   last_node = *head;
   while(last_node->next!=NULL){
      last_node = last_node->next;
   }
   last_node->next = new_node;
}
```

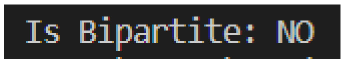
```
int dequeue_list(List** head){
   if(*head == NULL){
      //printf("Can not dequeue. Queue list is already empty\n");
      return -5; // special value meaning the list is empty
   }
   Node* temp = NULL;
   temp = *head;
   int value = temp->value;
   *head = temp->next;
   free(temp);
   return(value);
}

//------------------------- is_Bipartite algorithm ----------------------------//
// Return 1 for TRUE. 0 for FALSE.
int is_Bipartite(int* array[N][N]){
   int i;

   // initialize Queue
   Queue_List queue_list1;
   queue_list1.list.head = NULL;
   Queue_List** head_address_queue_list = &queue_list1.list.head;
   int color_array[N]; // 0 = RED. 1 = BLUE. -1 = UNASSIGNED.

   for(i = 0; i < N; i++){
      color_array[i] = -1; // assign values of -
1 to each node meaning "UNASSIGNED".
   }

   color_array[0] = 0; // assign index 0 with RED.
   enqueue_list(head_address_queue_list, 0); // enqueue first vertex.

   while(*head_address_queue_list != NULL){ // while queue is not empty
      int index = dequeue_list(head_address_queue_list);
      // all vertices connected to this vertex
      for(i = index + 1; i < N; i++){ // because it is undirected, only need to s
earch half of matrix
      // also do not need to compare array[i][i] because all diagonal elements ar
e zero.
         if(array[index][i] == 1){ // there is a connection
            if(color_array[i] == -1){ // color is UNASSIGNED, so assign a color
               if(color_array[index] == 0){color_array[i] = 1;} // assign opposit
e color
               else{color_matrix[i] = 0;}
               enqueue_list(head_address_queue_list, i); // enqueue first vertex.
               continue;
            }
     // If color matches, return 0 (FALSE) meaning the graph is not bipartite
            if(color_array[i] == color_array[index]){
               int free_the_memory = dequeue_list(head_address_queue_list);
               while(free_the_memory != -
5){ // special value meaning the list is completely empty
                  free_the_memory = dequeue_list(head_address_queue_list);
               }
               return 0; // FALSE (not bipartite)
            }
         }
      }
   }
   return 1; // return TRUE meaning the graph is bipartite.
}

int main(){
   int array[N][N] = {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}};  // adjacency matrix
   int is_bipartite = is_Bipartite(array);
   printf("Is Bipartite: ");
   if(is_bipartite == 1){
      printf("YES\n");
   }
   else{
      printf("NO\n");
   }
}
```



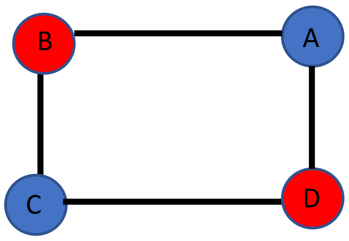
The graph above has the following adjacency matrix:

	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

The output of the algorithm for this graph is the following:



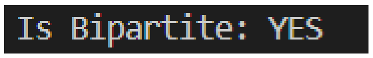
```
Is Bipartite: NO
```



The graph above has the following adjacency matrix:

	A	**B**	**C**	**D**
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

The output of the algorithm for this graph is the following:



```
Is Bipartite: YES
```

b) There are many ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors. Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.

The statement is true.

Definition: A bipartite graph is a graph G = (V,E) whose vertices can be partitioned into two sets ($V = V_1 \cup V_2 \; and \; V_1 \cap V_2$) such that there are no edges between vertices in the same set (for instance, if $u, v \; \in V_1$ then there is no edge between u and v).

In an undirected graph G(V,E), let $u_1 \in V_1$ where $V_1 \in V$. To create a cycle from u_1, the cycle must end at the point of origin, u_1. Let the origin u_1 be painted red. Let every vertex whose distance from u_1 is an odd number be painted blue and every vertex whose distance from u_1 is an even number be painted red. The colors will signify the two sets “Blue” and “Red”. If there is a cycle of odd length, then the origin, u_1, will eventually be painted blue when the cycle completes. Likewise, if there exists a cycle of even length, then the origin, u_1, will remain blue. Bipartite graphs do not allow edges between vertices of the same set. Therefore, if all cycles in G are of even length (no cycles of odd length) then G is a bipartite graph.

Question 4:

Often there are multiple shortest paths between two nodes of a graph. Give a linear-time algorithm for the following task.

Input: Undirected graph G = (V,E) with unit edge lengths; nodes $u, v \in V$.

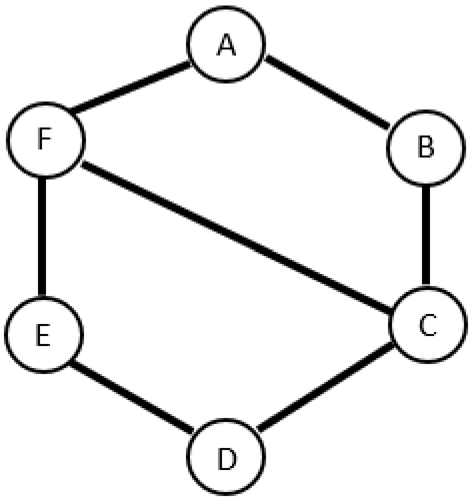
Output: The number of distinct shortest paths from u to v.

The following algorithm will run a breadth first search starting at u. If there is a path from u to v, the $minimum\_distance$ variable will be set to that distance if it is the shortest path encountered at that time, and the $number\_of\_shortest\_paths$ variable will be set to one. If there is another shortest path (equal length to $minimum\_distance$ variable) then the $number\_of\_shortest\_paths$ variable is incremented. When the DFS is complete, the $number\_of\_shortest\_paths$ variable is returned.

The following algorithm was implemented in C:

```c
#include <stdio.h>
#include <stdlib.h>
#define N 6

// ------------------------------ STRUCTS ------------------------------------//

typedef struct listNode {
   int value;
   struct listNode* next;
} Node;

typedef struct list{
   Node* head;
} List;

typedef struct queue_list{
   List list;
} Queue_List;

// ----------------------------- QUEUE LIST FUNCTIONS ---------------------------
-----//
void enqueue_list(List** head, int value){
   Node* new_node = NULL;
   Node* last_node = NULL;
   new_node = (Node*)(malloc(sizeof(Node)));
   new_node->value = value;
   new_node->next = NULL;
   // if list is empty insert new node at the head.
   if(*head == NULL){//if address for head node is NULL
      *head = new_node;
      return;
   }
   // start at the head looking for the last node, then insert new node after it.
   last_node = *head;
   while(last_node->next!=NULL){
      last_node = last_node->next;
   }
   last_node->next = new_node;
}

int dequeue_list(List** head){
   if(*head == NULL){
      //printf("Can not dequeue. Queue list is already empty\n");
      return -5; // special value meaning the list is empty
   }
   Node* temp = NULL;
   temp = *head;
   int value = temp->value;
   *head = temp->next;
   free(temp);
   return(value);
}

//------------------ find_Number_Of_Shortest_Paths algorithm ------------------//
// Return 1 for TRUE. 0 for FALSE.
int find_Number_Of_Shortest_Paths(int* array[N][N], int vertex_1, int vertex_2){
   int i, minimum_distance, number_of_shortest_paths;

   // initialize Queue
   Queue_List queue_list1;
   queue_list1.list.head = NULL;
   Queue_List** head_address_queue_list = &queue_list1.list.head;

   int distance_array[N];
   int has_been_queued[N];

   for(i = 0; i < N; i++){
      has_been_queued[i] = 0;
      distance_array[i] = 1000000000; // assign 1,000,000,000 (one billion) to re
present infinity
   }

   minimum_distance = 1000000000; // initialize minimum distance with infinity
   number_of_shortest_paths = 0;
   distance_array[vertex_1] = 0; // initialize vertex_1 with distance 0
   enqueue_list(head_address_queue_list, vertex_1); // enqueue vertex_1.
   has_been_queued[vertex_1] = 1;

   while(*head_address_queue_list != NULL){ // while queue is not empty
      int index = dequeue_list(head_address_queue_list);
      for(i = 0; i < N; i++){
         if(array[index][i] == 1){ // there is a connection
            if(has_been_queued[i] == 1){continue;} // do not queue a vertex that
has already been queued
            if(i == vertex_2){ // we have reached destination vertex.
               distance_array[i] = distance_array[index] + 1;
               if(distance_array[i] < minimum_distance){
                  minimum_distance = distance_array[i];
                  number_of_shortest_paths = 1; // reset shortest paths counter t
o 1
                  continue;
               }
               if(distance_array[i] == minimum_distance){ // if equal to current
min, increment shortest paths counter.
                  number_of_shortest_paths += 1;
                  continue;
               }
               continue; // if there is a path but it's longer than current minim
um, ignore it.
            }

            if(distance_array[i] == 1000000000){
               distance_array[i] = distance_array[index] + 1;
               enqueue_list(head_address_queue_list, i);
               has_been_queued[i] = 1;
               continue;
            }
         }
      }
   }

   return(number_of_shortest_paths);
}

int main(){

   int array[N][N] = {{0, 1, 0, 0, 0, 1},
                      {1, 0, 1, 0, 0, 0},
                      {0, 1, 0, 1, 0, 1},
                      {0, 0, 1, 0, 1, 0},
                      {0, 0, 0, 1, 0, 1},
                      {1, 0, 1, 0, 1, 0}};  // adjacency matrix

   int vertex_1 = 1;
   int vertex_5 = 5;

   int number_of_shortest_paths = find_Number_Of_Shortest_Paths(array, vertex_1,
vertex_5);
   printf("Number of shortest paths: %d\n", number_of_shortest_paths);
}
```

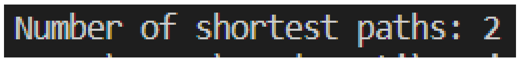
Consider the following undirected graph with unit edge lengths containing 6 vertices and 7 edges.



There are 2 shortest paths of length 2 between Vertex F and Vertex B: FAB and FCB. The adjacency matrix for this graph is below:

	A	B	C	D	E	F
A	0	1	0	0	0	1
B	1	0	1	0	0	0
C	0	1	0	1	0	1
D	0	0	1	0	1	0
E	0	0	0	1	0	1
F	1	0	1	0	1	0

The output of the algorithm to find the number of shortest paths from Vertex F to Vertex B is below.



```
Number of shortest paths: 2
```

Question 5:

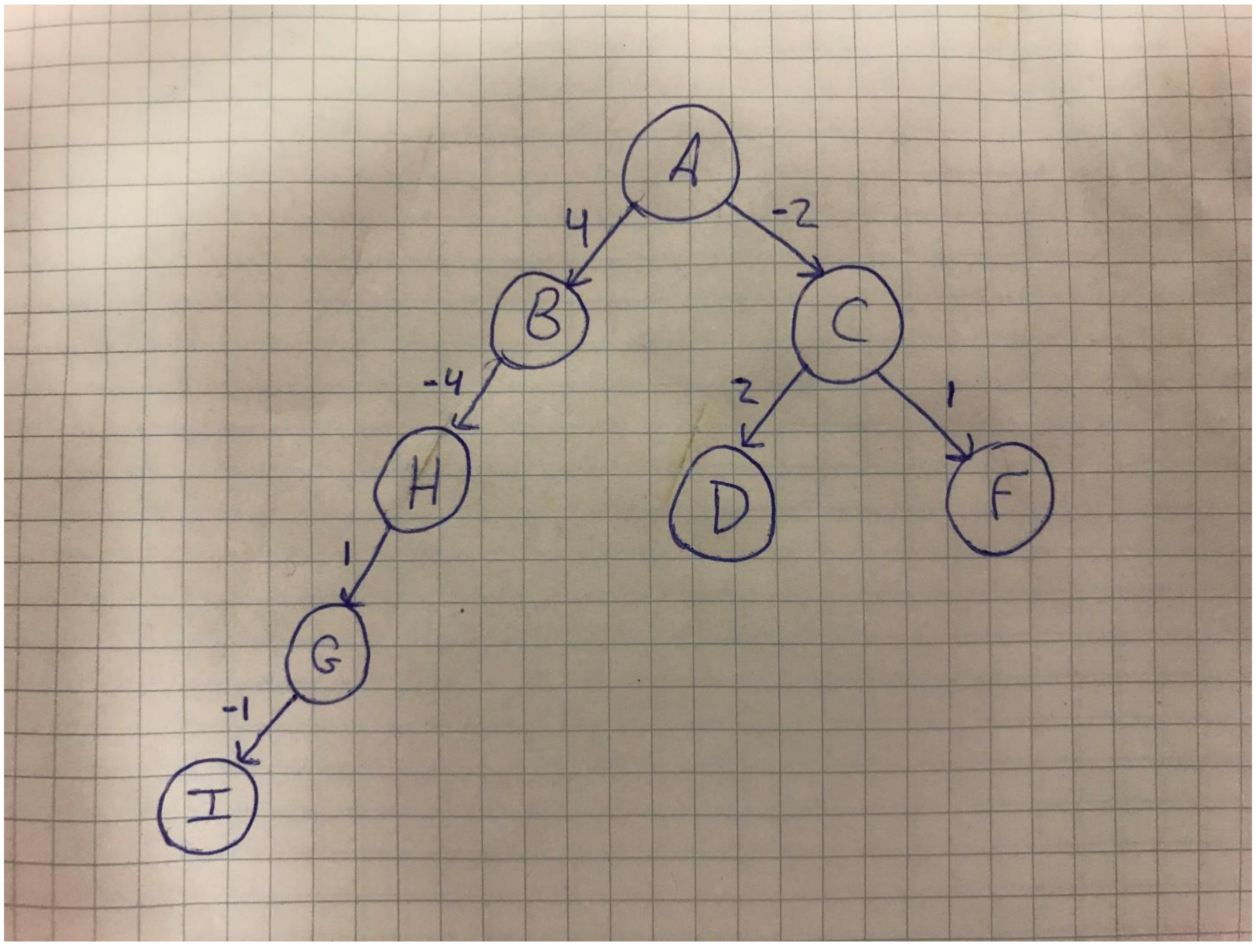
Use the Bellman-Ford algorithm starting at node A on the below graph.

a) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

The algorithm will run for 3 iterations because there is no improvement in the distances between vertices from the second iteration to the third. Before the first iteration, the starting vertex A is initialized with distance 0, and all other vertices are initialized with distance infinity.

Iteration	A	B	C	F	G	H	I	D
1	0	4	-2	-1	1	0	1	0
2	0	4	-2	-1	1	0	0	0
3	0	4	-2	-1	1	0	0	0

b) Show the final shortest-path tree.


A
B
C
H
D
F
G
I
4
-2
-4
2
1
1
-1