Anthony Redamonti

Syracuse university

Homework 3

CIS-675 Design and analysis of algorithms

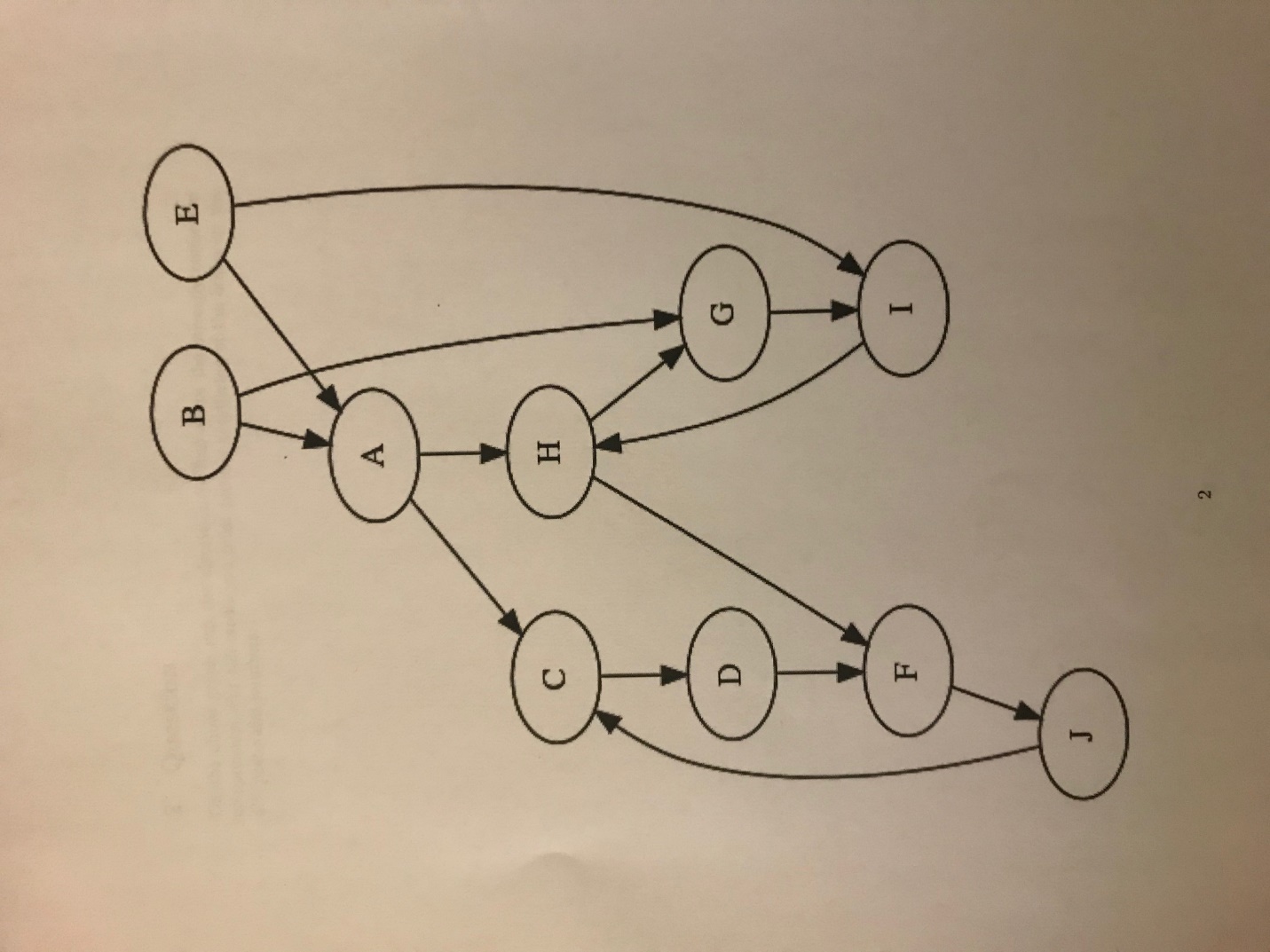
prof. Imani Palmer

5/3/2021

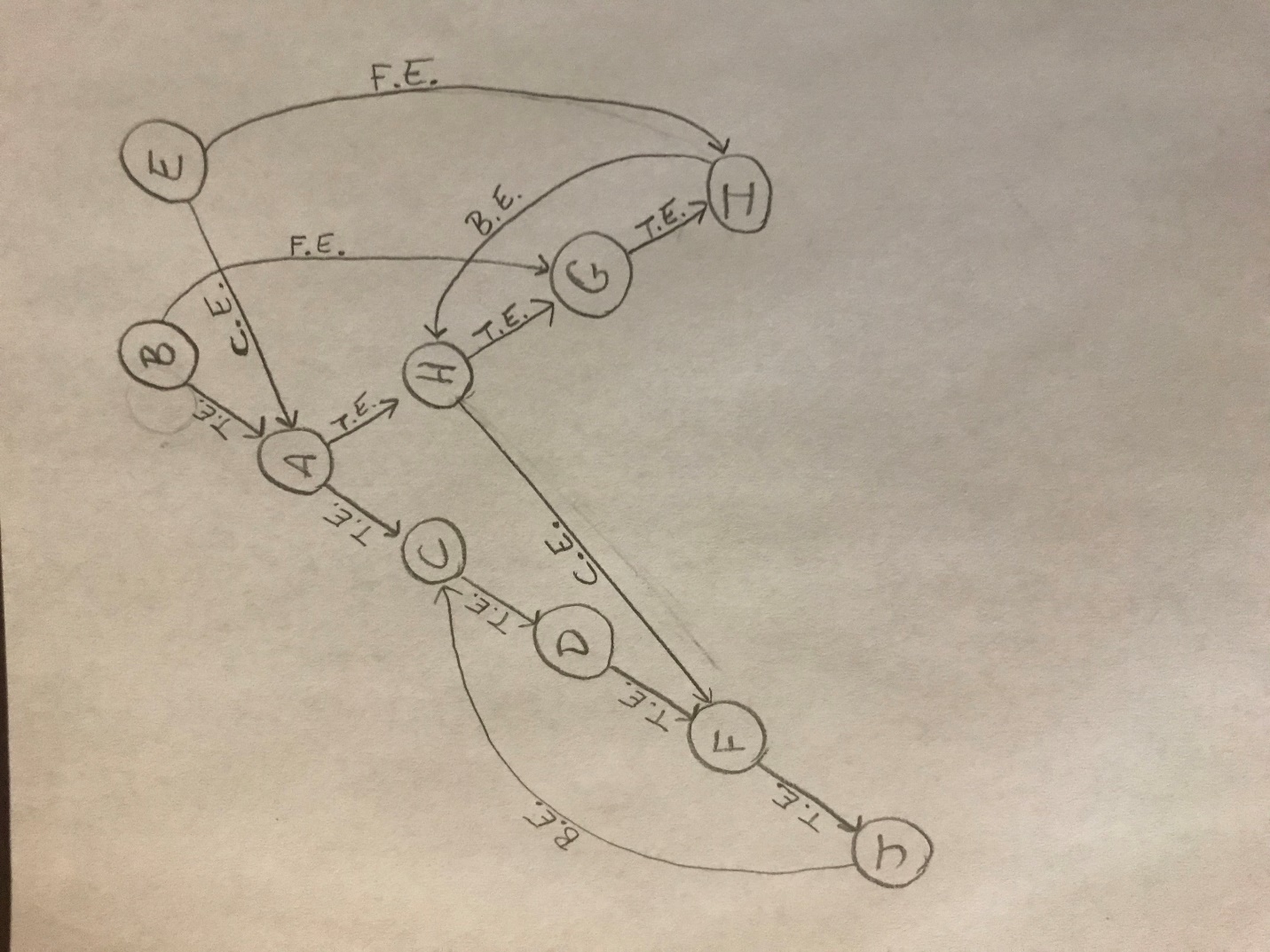
Question 1:

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 GR post-visit numbers.

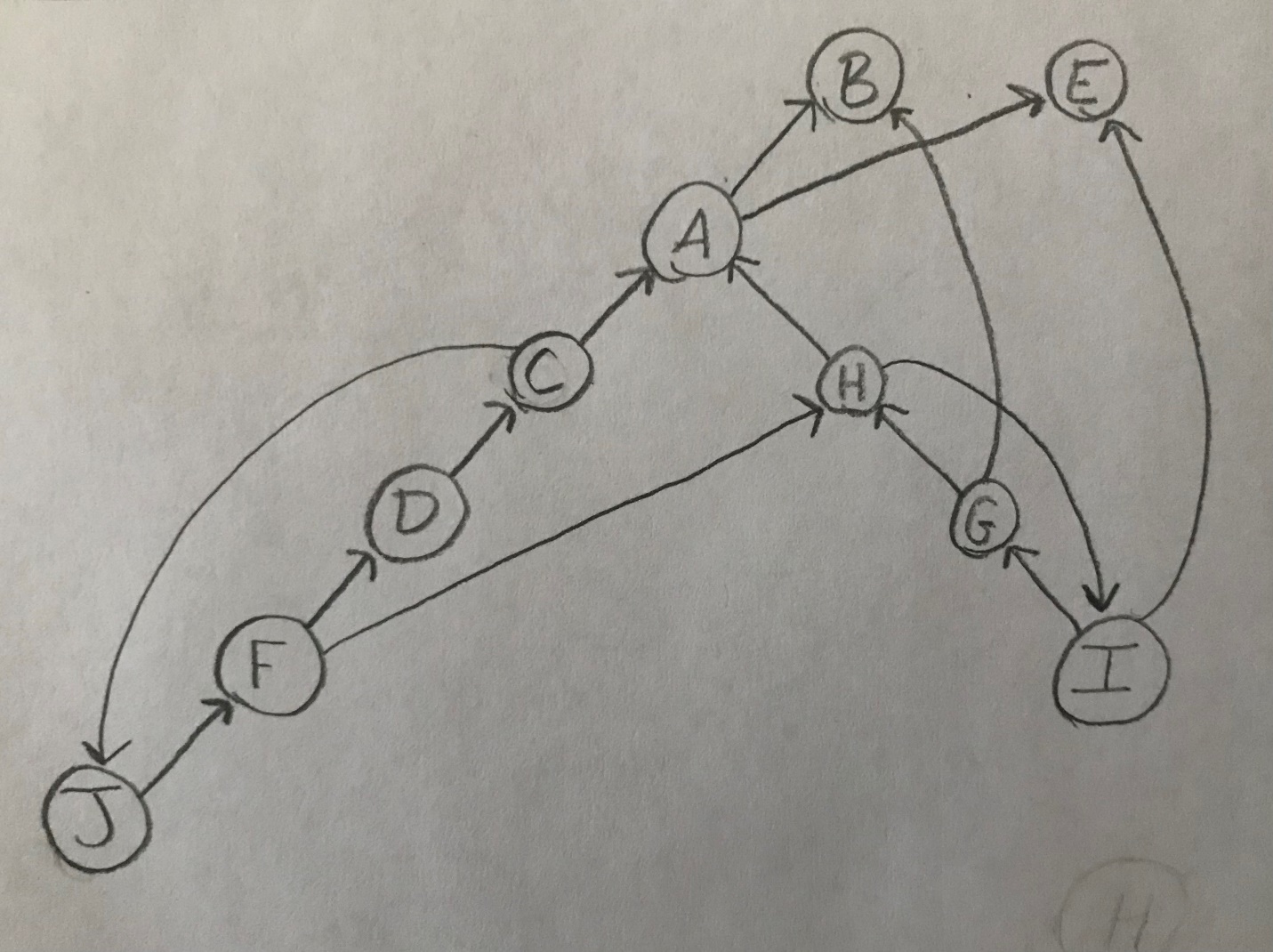
From Question 1, the DFS search yields the following pre and post-visit numbers starting at vertex B:

|  |  |  |
| --- | --- | --- |
| 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 GR, where GR is a directed graph with all edge directions reversed.



Pop the elements off the stack and perform DFS on them in GR.

|  |  |  |
| --- | --- | --- |
| 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 () such that there are no edges between vertices in the same set (for instance, if then there is no edge between u and v).

1. 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 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.

;

Below is the implementation of the algorithm written in 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 search half of matrix

      // also do not need to compare array[i][i] because all diagonal elements are 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 opposite 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");

   }

}

C

B

A

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:



A

B

D

C

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:



1. 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 () such that there are no edges between vertices in the same set (for instance, if then there is no edge between u and v).

In an undirected graph G(V,E), let where . To create a cycle from , the cycle must end at the point of origin, . Let the origin be painted red. Let every vertex whose distance from is an odd number be painted blue and every vertex whose distance from 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, , will eventually be painted blue when the cycle completes. Likewise, if there exists a cycle of even length, then the origin, , 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 .

*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 variable will be set to that distance if it is the shortest path encountered at that time, and the variable will be set to one. If there is another shortest path (equal length to variable) then the variable is incremented. When the DFS is complete, the variable is returned.

The following algorithm was implemented in 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 represent 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 to 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 minimum, 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;

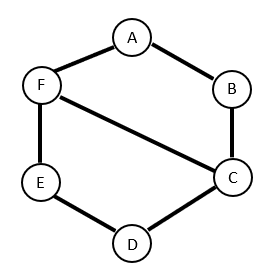
   int vertex\_5 = 5;

   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.



Question 5:

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

1. 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 |

1. Show the final shortest-path tree.

