CPT212
DESIGN & ANALYSIS OF ALGORITHMS

Academic Session 2021/2022, Semester II

ASSIGNMENT II:
GRAPH ALGORITHMS

For the attention of:
Dr. Nur Intan Raihana Ruhaiyem

No	Name	Matric No	Problem
1	Ling Shao Doo	153152	Shortest path
2	Loh Zhi Xuan	152734	
3	Khoo Ke Rou	153149	
4	Lim Phei San	152726	Strong connectivity

Date of Submission:
10 July 2022

Table of Contents

1. Data Structure

The data structure that we had chosen to represent the graph is adjacency matrix. The adjacency matrix is implemented using a 2D vector in the code (source file).

1.1. Brief explanation on the data structure chosen

In this matrix implementation, each of the rows and columns represent a city in the graph. The value that is stored in the cell at the intersection of row v and column w indicates that there is an edge from city v to city w, in other words, city w is adjacent to city v. [1] For example, let's say a 2,1 is a cell that contains the value of 9 in the adjacency matrix, then the cell is located at the second row and first column of the adjacency matrix, which implies that there is a directed edge of weight 9 that starts and ends at the cities correspond to the second row and the first column of the adjacency matrix respectively.

1.2. Justification on the choice

There are several reasons for us to choose an adjacency matrix as our data structure to store the graph. First, any edge between any two cities can be obtained from the adjacency matrix in O (1) time as the exact location of the edge can be checked directly from the matrix. This helps to speed up the time needed to perform the functions especially to find the shortest path as it is easy to determine whether an edge exists in the graph. Similar to that, adding an edge to the graph will take only O (1) time as well because we only need to insert the weight to the corresponding cell directly. Besides, adjacency matrix is efficient for small and it suits dense graphs better. Since the default graph contains only five cities, the graph can be considered as a small graph, and the graph might grow denser as the number of edges of the graph might increase after the functions are being performed on it.

2. Graph Functions

2.1. Fundamental functions

The fundamental functions involved are functions that represent the directed graph using adjacency matrix and some operations that could be performed on the graph for various objectives. We have taken the code from the internet which we have mentioned its source in the reference and have done some modifications on the functions to suit our requirements. In the original code [2], the class, WeightedGraph, consists of various graph functions while the class city consists of a function that stores the name of city as string. The code support C++11, one of the versions of standard programming language C++.

2.1.1. Description and Justification of fundamental functions:

2.1.1.1. Class WeightedGraph

2.1.1.1.1 void initialization (int size) //size = number of vertices

- Initialize the size of vector cities, edges, and marks. Set the value of every city and weight of edges to NULL to indicate that all the cities are not connected for now.

2.1.1.1.2 void add_city(city* aCity)

-

2.1.1.1.3 void add_edge(int fromCity, int toCity)

- Using Adjacency matrix data structure to store the weight of edges.
- The cell which is not null implies that there exists an edge which is directed from the city of the corresponding row to the city of the corresponding column.

2.1.1.1.4 void remove_edge(int fromCity, int toCity)

- Set the weight of the edge between two desired cities to 0.

2.1.1.1.5 int weight_is(int fromCity, int toCity)

- Get the weight of the edge between two desired cities.

2.1.1.1.6 int index_is(city* aCity)

- Get the integer value of the index of the desired city by passing in the vector, cities.

2.1.1.1.7 void clear_marks()

- Reset all the cities as unvisited.

2.1.1.1.8 void mark_city(city* aCity)

- Mark aCity as visited.

2.1.1.1.9 bool is_marked(city* aCity)

- Check whether a city is visited, return TRUE if it is, else return FALSE.

2.1.1.1.10 city* get_unmarked()

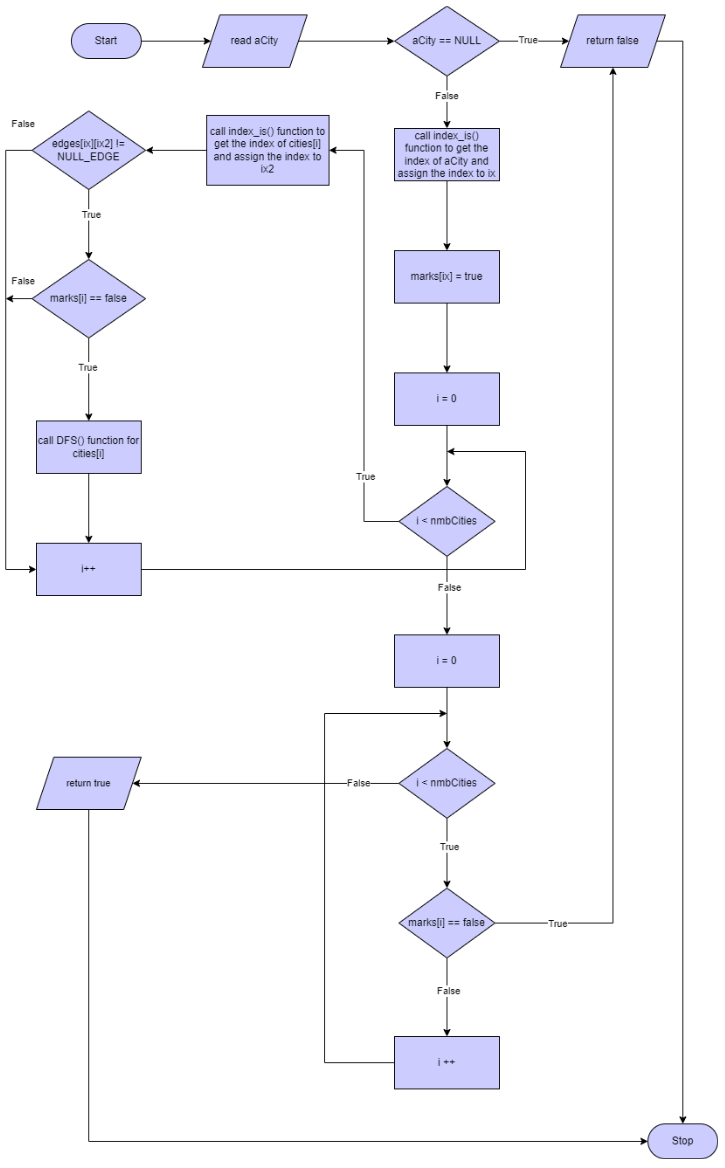
- Return the cities that are not yet visited, if such city does not exist, return NULL.

2.1.1.1.11 City* getCity(int index)

- Return the name of the city.

2.1.1.1.12 bool DFS(city* aCity)

- This function is edited to return TRUE if all the cities are visited, else it returns FALSE. This modification is made to suit the requirement of Function 1 (strong connectivity).
- DFS will traverse from the chosen city s to an unvisited city u that is adjacent to it and go all the way deeper by visiting the next city v that is adjacent to u and so on. It does so by recursively calling itself using the next unvisited city as parameter.
- After visiting a city that has no unvisited city adjacent to it, DFS will backtrack along the route to look for an unvisited city which is adjacent to the cities in the route.
- Below shows the flowchart of this algorithm:


Start
read aCity
aCity == NULL
True
return false
False
call index_is() function to get the index of aCity and assign the index to ix
marks[ix] = true
i = 0
i < nmbCities
True
call index_is() function to get the index of cities[i] and assign the index to ix2
edges[ix][ix2] != NULL_EDGE
False
True
marks[i] == false
False
True
call DFS() function for cities[i]
i++
False
i = 0
i < nmbCities
False
return true
True
marks[i] == false
True
False
i ++
Stop

2.1.1.1.13 void printCycles(int& counting, int sequenceArr[])

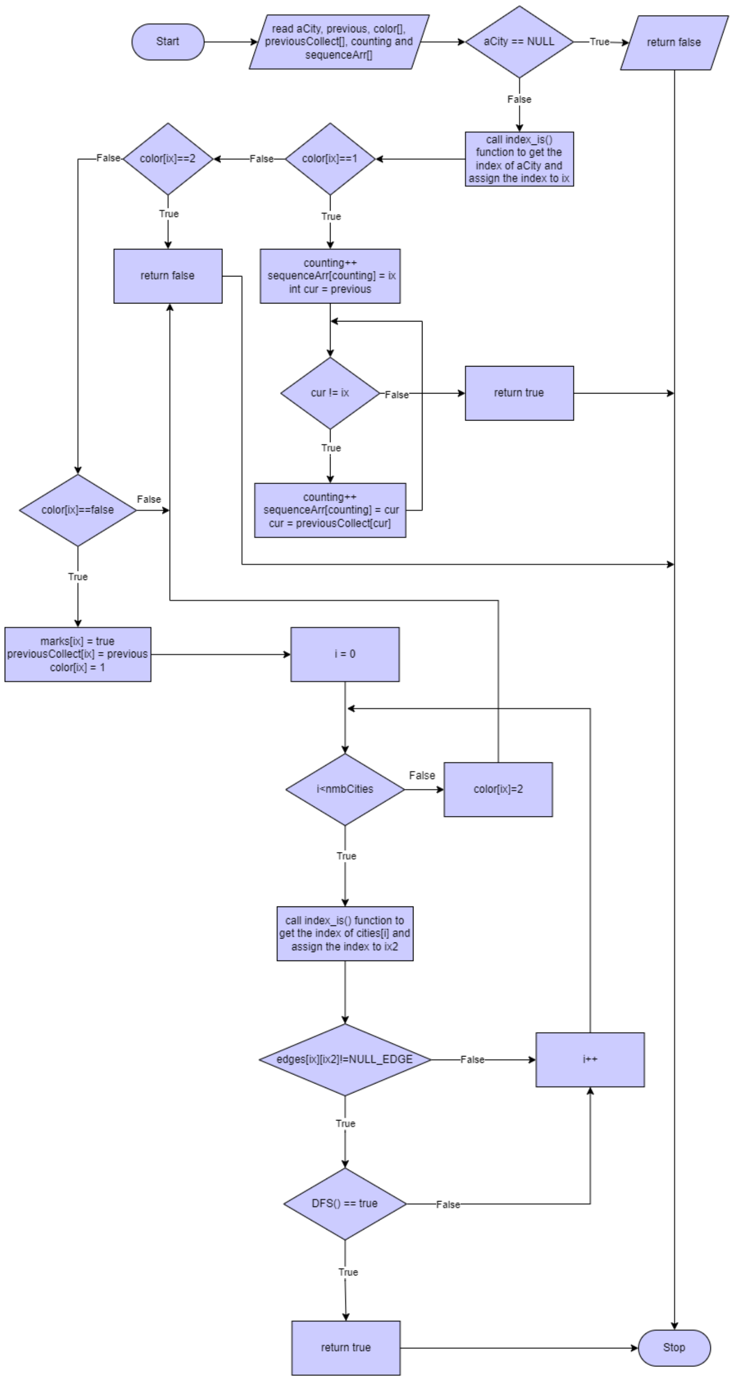
- This function is to print the detected cycle.

2.1.1.1.14 bool isGraphCyclic(int& counting, int sequenceArr[])

- This function is edited to return TRUE if there is a cycle in the graph, else it returns FALSE.

2.1.1.1.15 bool DFS_for_cycle(city* aCity,int previous, int color[],int previousCollect[],int& counting, int sequenceArr[])

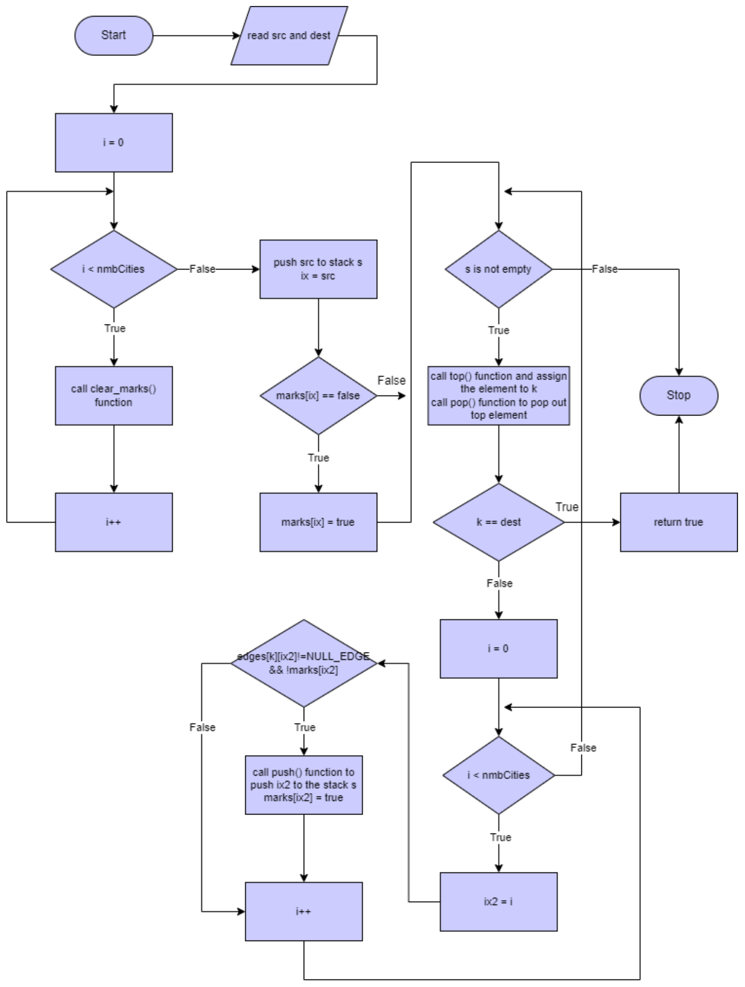
- Below shows the flowchart of this algorithm:



Start
read aCity, previous, color[], previousCollect[], counting and sequenceArr[]
aCity == NULL
True
return false
False
call index_is() function to get the index of aCity and assign the index to ix
color[ix]==1
False
color[ix]==2
False
True
return false
True
counting++
sequenceArr[counting] = ix
int cur = previous
cur != ix
False
return true
True
counting++
sequenceArr[counting] = cur
cur = previousCollect[cur]
color[ix]==false
False
True
marks[ix] = true
previousCollect[ix] = previous
color[ix] = 1
i = 0
i<nmbCities
False
color[ix]=2
True
call index_is() function to get the index of cities[i] and assign the index to ix2
edges[ix][ix2]!=NULL_EDGE
False
i++
True
DFS() == true
False
True
return true
Stop

2.1.1.1.16 bool checkPath(int src, int dest)

- This function is to check whether there is a path between source city and destination city. Return TRUE if there is a path between them, else return FALSE.
- Below shows the flowchart of this algorithm:



2.1.1.1.17 void printPath(int parent[], int src. Int dest)

- This function is to print the path between the source city and destination city.

2.1.1.1.18 int removeMin(int dist[], bool sptSet[], int p[], int dest)

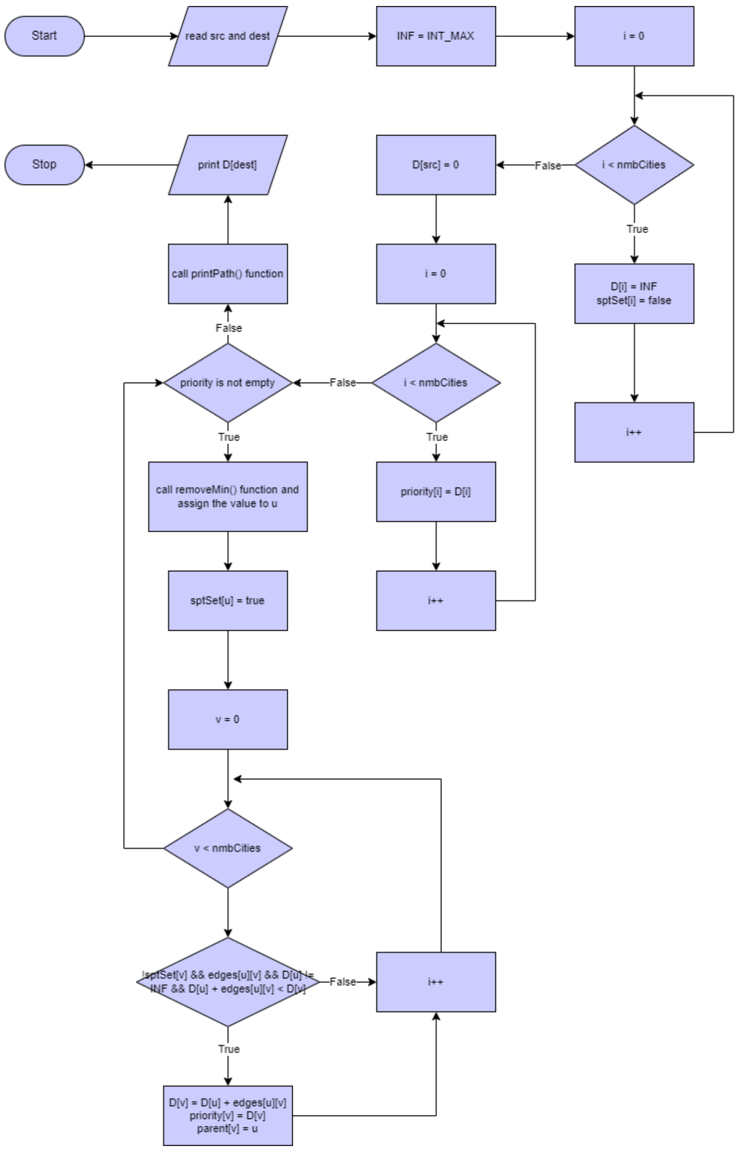
-

2.1.1.1.19 bool is_empty(int queue[])

-

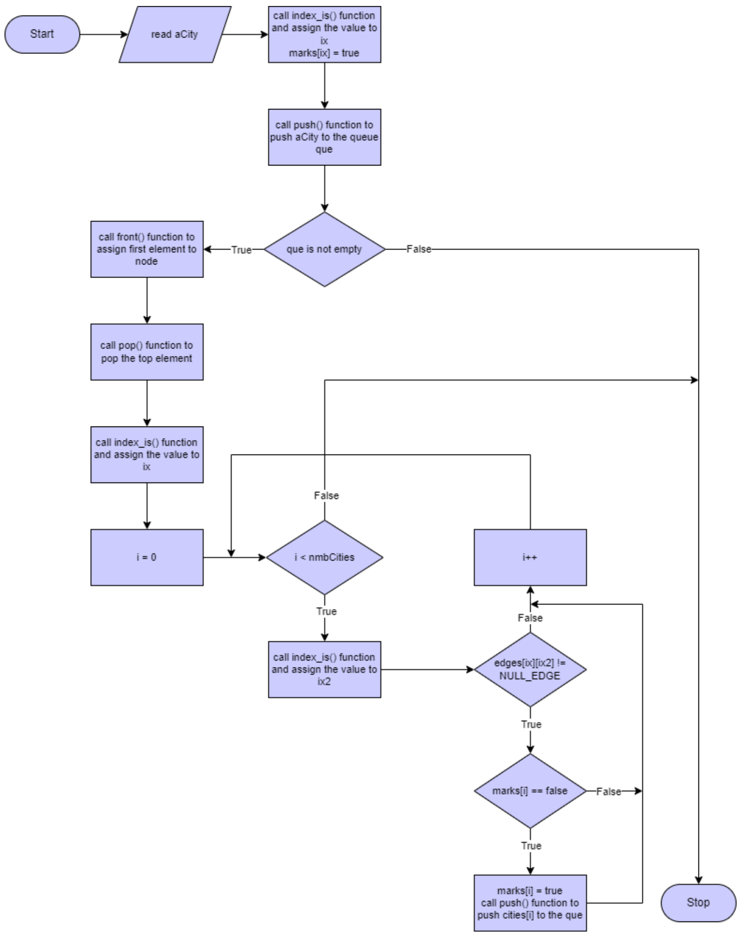
2.1.1.1.20 void findPath(int src, int dest)

- This function is to find a path between the source city and destination city.
- Below shows the flowchart of this algorithm:



2.1.1.1.21 void BFS(city* aCity)

- This function performs Breadth First Search (BFS) and prints the cities found and connected with each other.
- Below shows the flowchart of this algorithm:



Start
read aCity
call index_is() function and assign the value to ix marks[ix] = true
call push() function to push aCity to the queue que
que is not empty
True
False
call front() function to assign first element to node
call pop() function to pop the top element
call index_is() function and assign the value to ix
i = 0
False
i < nmbCities
True
i++
False
call index_is() function and assign the value to ix2
edges[ix][ix2] != NULL_EDGE
True
marks[i] == false
False
True
marks[i] = true call push() function to push cities[i] to the que
Stop

2.1.1.1.22 void displayGraph()

- This function is added to display the cities of the graph and the weight of each edge between cities.
- The graph will be displayed using Adjacent Matrix data structure.

2.1.1.1.23 bool checkEdge(int sec, int dest)

- This function is to check whether there is an edge between source city and destination city.

2.2. Additional Functions

2.2.1. Description and Justification of Additional Functions:

2.2.1.1. void defaultCity(Graph, numberOfCities)

- Set the default city names when the program is started.

2.2.1.2. void defaultEdge(Graph)

- Set the default edges between default cities and the default weight for each edge.
- When the initialize graph function is called, this function will be called to reset the edges back to default one.

2.2.1.3. void initializeGraph(Graph, numberOfCities)

- Reset the graph to the default graph

2.2.1.4. void addRandomEdge(Graph, numberOfCities)

- Function to add random edges of weight between random cities.
- Below shows the flowchart of this function:

2.2.1.5. void removeEdge(Graph, numberOfCities)

- Function to remove a certain edge between two cities.
- Below shows the flowchart of this function:

2.2.1.6. void transposeGraph(Graph, Transpose Graph, numberOfCities)

- This function is added to reverse all the direction of edges between cities but maintain the original weight and the end cities of the edges which is used in Function 1 (strong connectivity).
- Below shows the pseudocode of this function:

2.2.1.7. void strongConnectivity(Graph, numberOfCities)

- Function 1, the problem-solving method, discussed in Section 3.2.

2.2.1.8. void cycle(Graph, numberOfCities)

- Function 2, the problem-solving method, discussed in Section 3.3 is a function to pass graphs to the class and call class function and PRINT whether there is a cycle or no cycle in the graph.

2.2.1.9. void shortestPath(Graph, numberOfCities)

- Function 3, the problem-solving method, discussed in Section 3.4.

3. Description of problem-solving method

3.1. Menu

3.1.1. Purpose

The main purpose of the Menu is to print out a menu that provides five options for the user as follows:

- Option 1: Strong connectivity (To check if the graph is strongly connected, generate random edges between random vertices if it is not, and print out the resulting graph).
- Option 2: Cycle detection (To check if the graph contains a cycle, generate random edges between random vertices if it does not, and print out the resulting cycle).
- Option 3: Shortest path (To find the shortest path between two vertices selected by the user, generate random edges between random vertices if such path does not exist, and print out the resulting path).
- Option 4: Reset graph (To initialize the graph into the default graph as given in the instruction).
- Option 5: Minimum Spanning Tree
- Option 6: Remove Edge (To remove the edge between two cities selected by the users).
- Option 7: Exit (To let the user end/terminate the program).

3.1.2. Flowchart

3.1.3. Explanation

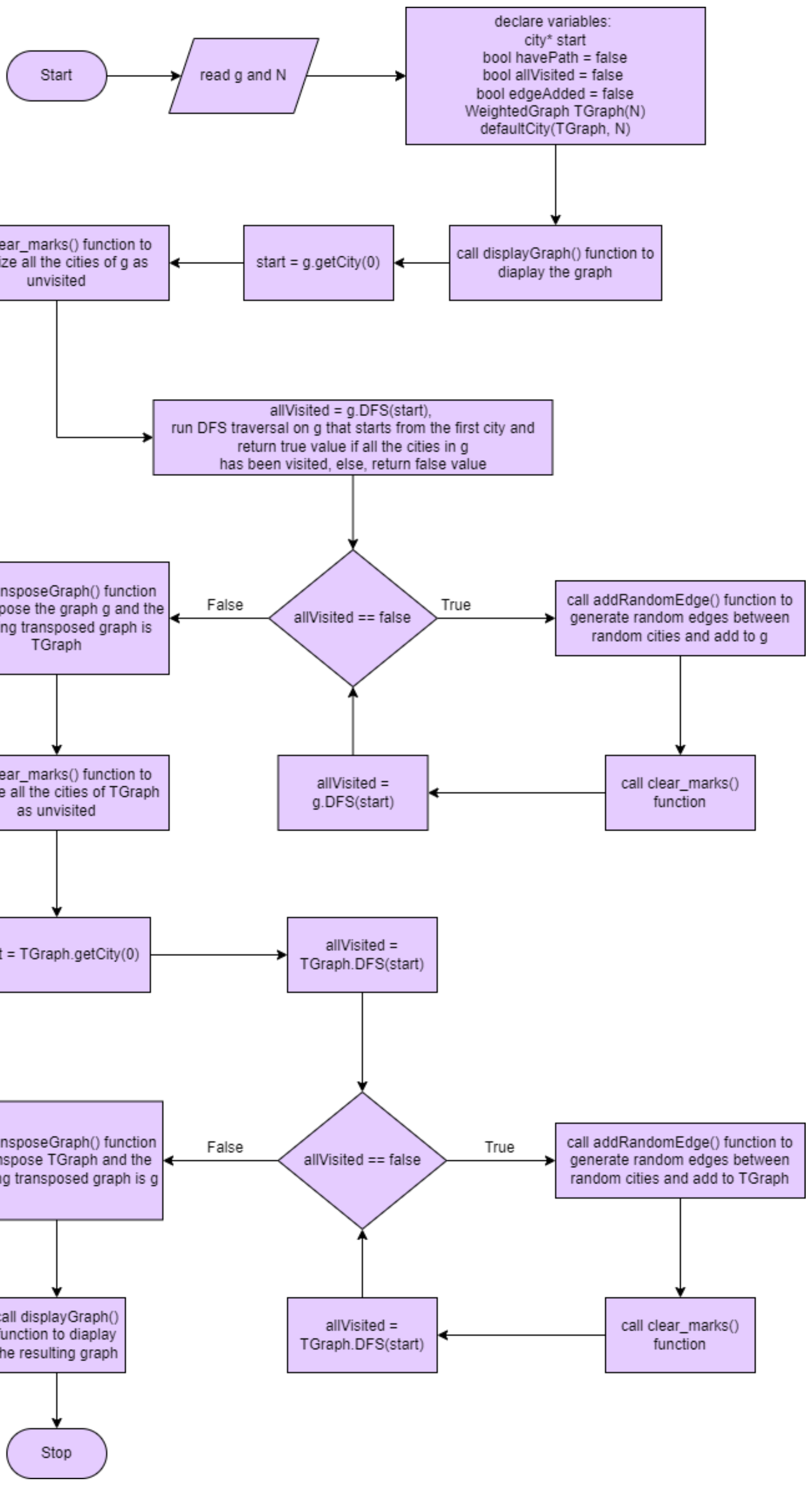
The program will print out the menu at the beginning of the program where the graph has already been initialized to the default graph, and prompt the user to select one of the five options provided as mentioned earlier. Based on the pseudocode, we can see that as long as the user does not choose Option 5 (the exit option), the user can keep on selecting one of the options available. The graph will not be reset to the default graph after performing one of the functions from Option 1 to Option 3. Besides, input validation is included here so if the user enters input of the wrong data type, a reminder will be shown and the menu will be printed out again to get the choice of the user.

3.2. Function 1: Strong connectivity

3.2.1. Purpose

The purpose of this function is to check if the graph is strongly connected. If it is not, random edges will be generated between random vertices until the graph is strongly connected. Finally, the resulting graph will be printed.

3.2.2. Flowchart


Start
read g and N
declare variables:
city* start
bool havePath = false
bool allVisited = false
bool edgeAdded = false
WeightedGraph TGraph(N)
defaultCity(TGraph, N)
call displayGraph() function to diaplay the graph
start = g.getCity(0)
call clear_marks() function to initialize all the cities of g as unvisited
allVisited = g.DFS(start), run DFS traversal on g that starts from the first city and return true value if all the cities in g has been visited, else, return false value
allVisited == false
False
True
call transposeGraph() function to transpose the graph g and the resulting transposed graph is TGraph
call addRandomEdge() function to generate random edges between random cities and add to g
call clear_marks() function
allVisited = g.DFS(start)
call clear_marks() function to initialize all the cities of TGraph as unvisited
start = TGraph.getCity(0)
allVisited = TGraph.DFS(start)
allVisited == false
False
True
call transposeGraph() function to transpose TGraph and the resulting transposed graph is g
call addRandomEdge() function to generate random edges between random cities and add to TGraph
call clear_marks() function
allVisited = TGraph.DFS(start)
call displayGraph() function to diaplay the resulting graph
Stop

3.2.3. Explanation

For a graph to be strongly connected, each vertex of the graph must reach all other vertices. Therefore, two depth-first search (DFS) traversals has to be done on the graph. The first DFS traversal is to check if the starting vertex can reach all other vertices. If there is any vertex that is not visited during the traversal, the graph is not strongly connected. Else, the second DFS traversal will take place. The second DFS traversal is to check if all other vertices can reach the starting vertex, however, this traversal is performed on the transpose of the graph where the graph is transposed by reversing the directions of the edges. If there is any vertex that is not visited during the traversal, the graph is not strongly connected. Else, the graph is considered strongly connected. Thus, in order to make sure a graph is strongly connected, one or more edges have to be added to the graph so that all vertices are being visited in both DFS traversals. (The functions used here are already discussed in Section 2.1 and 2.2.)

To solve this problem, six steps need to be taken to produce a strongly connected graph based on the input graph (may be the default graph or modified graph). The steps are as follows which also correspond to the pseudocode in Section 3.2.2:

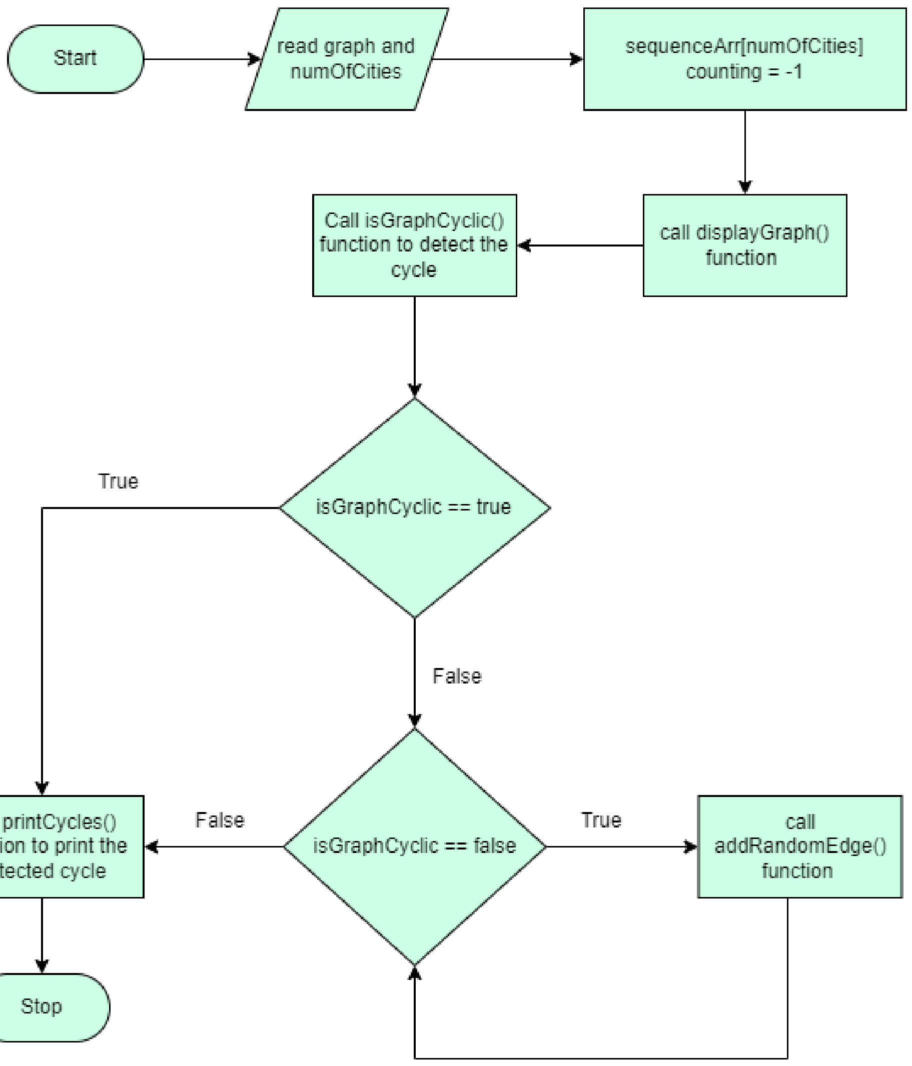
1. First, every vertex of the graph is initialized as unvisited.
2. Second, an arbitrary vertex is chosen, which is the first vertex (AU) in this case. Then a DFS traversal (the first traversal) is performed on the graph from the chosen vertex. If there is any vertex that is not visited during the first DFS traversal, random edges between random vertices need to be generated and added to the graph, and DFS traversal is performed on the graph from the same chosen vertex again until all vertices have been visited.
3. Third, the graph is transposed by reversing the directions of the edges of the graph.
4. Fourth, every vertex of the transposed graph is initialized as unvisited.
5. Fifth, a DFS traversal (the second traversal) is performed on the graph from the same vertex (AU) which is also the starting vertex in the first traversal. If there is any vertex that is not visited during the second traversal, random edges between random 6 vertices need to be added to the graph, and DFS traversal is performed on the graph from the same vertex again until all vertices have been visited.
6. Sixth, the transposed graph is transposed again to obtain the final strongly connected graph.

3.3. Function 2: Cycle detection

3.3.1. Purpose

The purpose of this function is to check if the graph has a cycle. If it does not have one, random edges will be generated between random vertices until a cycle is detected. Finally, the resulting graph will be printed.

3.3.2. Flowchart


Start
read graph and numOfCities
sequenceArr[numOfCities]
counting = -1
call displayGraph() function
Call isGraphCyclic() function to detect the cycle
True
isGraphCyclic == true
False
call printCycles() function to print the detected cycle
False
isGraphCyclic == false
True
call addRandomEdge() function
Stop

3.3.3. Explanation

Depth First Search for Cycle Detection

For a directed graph to have a cycle, the graph needs to have at least one back edge present in the graph. A back edge is an edge from a vertex to one of its ancestors [4] in the tree produced by Depth First Search (DFS). Parallel edges and self-loops have back edges that cause the cycle between vertices. Using DFS to mark the visited vertices will allow the program to find out which child nodes of a DFS tree are pointed back to its ancestors and thus prove the existence of a cycle in the graph.

Algorithm used (DFS and coloring method)

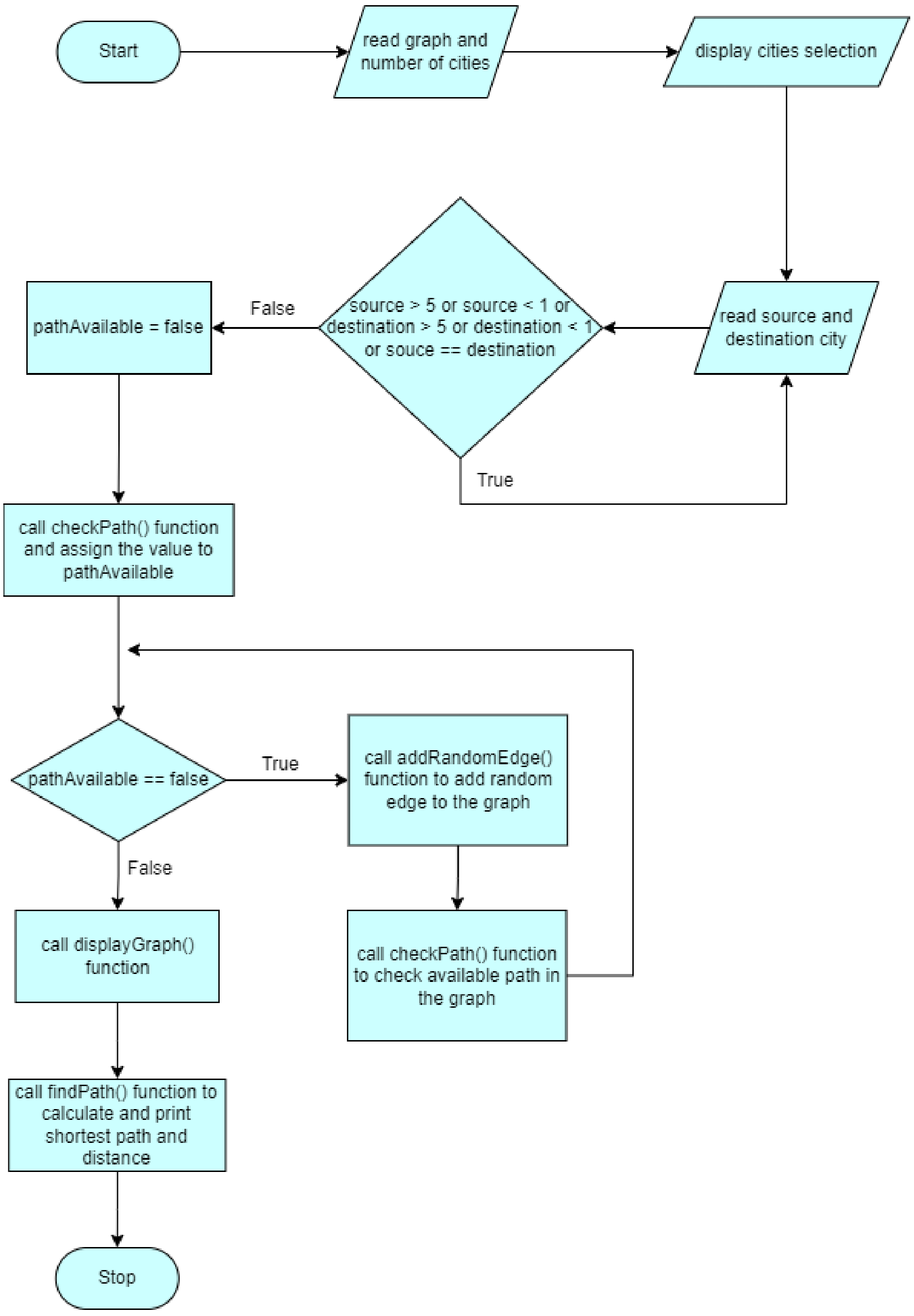
1. DFS will be used to traverse the graph and mark the visited vertices with color.
2. There are three types of color used here, where the color is an array that holds the color of the visited vertices. The color 0, indicate not visited; color 1, indicate visited once; color 2, indicate all vertices are visited.
3. When DFS traverses the graph, if the previously visited vertex is visited again by its descendant, it shows that there is a back edge connected to the vertex.
4. Therefore, the repeat visited vertices will be marked as part of the cycle array and trace back all the vertices connected in a closed chain.
5. Any graph with no cycle, void cycle will call function addRandomEdges() to generate a new edge, each generation will go through function isGraphCyclic() to check for the present cycle in the graph.

3.4. Function 3: Shortest path

3.4.1. Purpose

The purpose of this function is to check the shortest path available from the source vertex to destination vertex set by the user. If there is no path between the selected vertices, the function will add random edges until a path is detected. The updated graph, shortest path, and distance between the source and destination vertices will be printed.

3.4.2. Flowchart


Start
read graph and number of cities
display cities selection
read source and destination city
source > 5 or source < 1 or destination > 5 or destination < 1 or souce == destination
False
True
pathAvailable = false
call checkPath() function and assign the value to pathAvailable
pathAvailable == false
True
call addRandomEdge() function to add random edge to the graph
call checkPath() function to check available path in the graph
False
call displayGraph() function
call findPath() function to calculate and print shortest path and distance
Stop

3.4.3. Explanation

In order to map the shortest path between the selected vertices, the destination vertex must be reachable from the source vertex. Hence, a modified DFS is used to search for the destination vertex from the source vertex. If there is no path from source to destination, the function will add random edges and check for path until the destination vertex is reachable. Then, the function will check for the shortest path between the source and destination using a modified Dijkstra's Algorithm. The function will go to every vertex using priority queue to find the shortest path possible until a shortest path tree is mapped out. The shortest distance of each vertex will be kept in an array while the vertices of the shortest will be mapped by using parent arrays.

Algorithm used

Depth First Search (DFS) Algorithm

1. DFS is used to traverse the graph.
2. A code is added to check if destination vertex is visited. If not, the algorithm will return false.

Dijkstra's Algorithm

1. BFS is used to traverse the graph and mark the vertices as included in the shortest path tree.
2. Using the priority queue where the key is shortest distance and value is vertex index, the vertex with shortest distance from current vertex is checked first.
3. When checking for adjacent vertices, the shortest distance and shortest path tree is updated if a shorter distance is found.
4. To print the path, the parent array is called recursively from destination to source then printed in the correct order.

3.5. Function 4: Minimum Spanning Tree

3.5.1. Purpose

3.5.2. Flowchart

3.5.3. Explanation

3.6. Function 5: Reset Graph

This function is included to reset the graph back to the default one as the graph may have been modified by the functions above, and the default graph will be printed.

3.7. Function 6: Remove Edge

This function is included to remove the edge between the source city and destination city if the edge exists.

4. Discussion

5. References