CSCE4133/5133 – Algorithms
Fall 2023

Assignment 2
Linked List and Stack

Out Date: Sep. 18, 2023
Due Date: **Oct. 2, 2023**

Instructions:

- **Written Format & Template:** Students can use either Google Doc or Latex
- Write your full name, email address and student ID in the report.
- Write the number of "Late Days" the student has used in the report.
- Submission via BlackBoard
- **Policy:** Can be solved in groups (two students per group at max) & review the late-day policy.
- **Submissions:** Your submission should be a PDF for the written section & a zip file with your source code implementation.
- For more information, please visit course website for the homework policy.

Searching Algorithms: Breadth First Search and Depth First Search

1. Overview

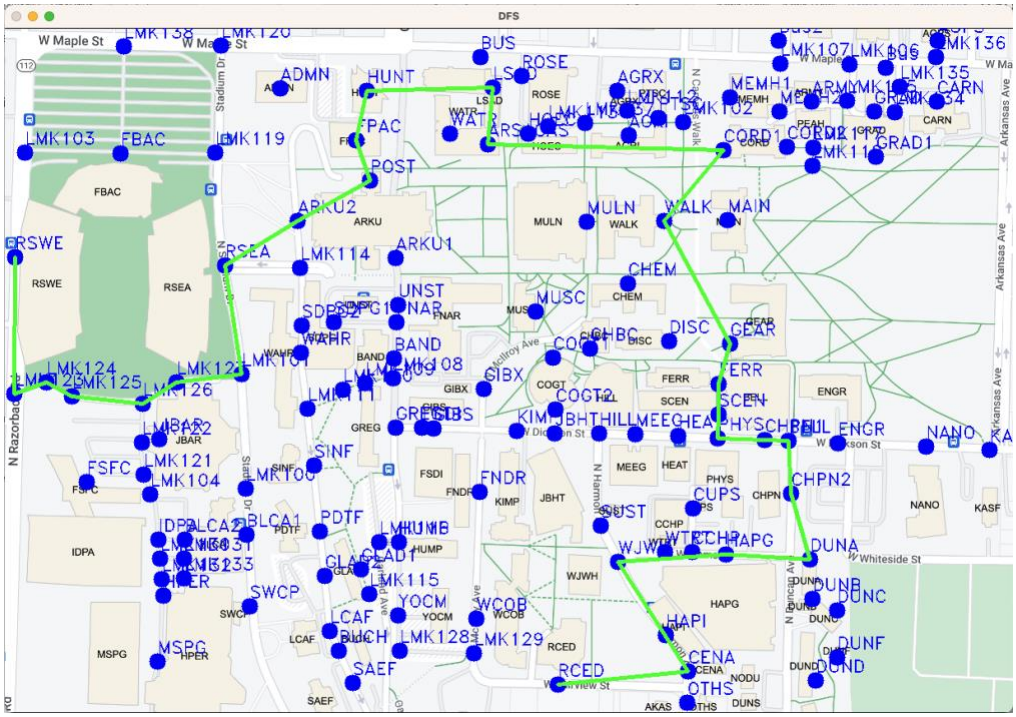

Figure 1: The path from the RSWE building to the RCED building searched by BFS

In homework 1, we have studied about the linked list and queue implementation. The linked list is used to build the adjacency matrix of the graph. Meanwhile, the queue is used to implement the graph searching algorithm, i.e., breadth first search. In this homework, we are going to implement two common graph searching algorithms studied in the previous lecture, i.e., breadth first search (BFS) and depth first search (DFS). In this assignment, the student is ***REQUIRED*** to implement in C++. The student is only needed to write the missing implementation following the instruction and requirements.

In the second homework, we are going to implement BFS and DFS that aims to find a path from a given building to another building. ***For graduate student, you are required to implement an additional question which is finding the articulation points and bridges on the graph.*** The map is represented as a graph structure in which each node represents one building (or a landmark), and an undirected edge between two nodes represents the path between buildings/landmarks. A list of edges in a graph is represented by an adjacency list[1] that is implemented by a linked list. The BFS algorithm is implemented by a queue and the DFS is implemented by either a stack or a recursive function. In this assignment, we provide you the implementation of linked list, queue, and stack.

In this assignment, you are ***REQUIRED*** to implement ***BFS***, ***DFS, Finding Articulation Points and Bridges*** (Graduate Students) algorithms. If you think the homework is too long,

don't worry, it is long because we provide you with the detail of the descriptions. There are some hints that may help to be success in this homework:

- You should start your homework early.
- You should read the descriptions of homework carefully before you start working on the homework.
- Before implementing the required functions, you should try to compile the source code first. Basically, the source code with an empty implementation can be compiled successfully. This step will help you to understand the procedure of each problem.
- If you forgot BFS and DFS algorithms, you are encouraged to revise these structures in the lecture slides.

The source code is organized as follows:

File or Folder	***Required Implementation***	***Purpose***
`assets/`	No	Contain the data of homework
`include/`	No	Contain headers files
`src/`	No	Contain source code files
`bin/`	No	Contain an executable file
`objs/`	No	Contain object files
`include/linked_list.hpp`	No	Define the linked list structure
`include/queue.hpp`	No	Define the queue structure
`include/stack.hpp`	No	Define the stack structure
`include/graph.hpp`	No	Define the graph structure
`include/search.hpp`	No	Define the header of BFS, DFS, recursive DFS, and Finding Articulation Points and Bridges algorithms
`src/linked_list.cpp`	No	Implement the linked list structure
`src/queue.cpp`	No	Implement the queue structure
`src/stack.cpp`	No	Implement the stack structure
`src/graph.cpp`	No	Implement the graph structure and algorithms
`src/bfs.cpp`	Yes	Implement the BFS algorithm
`src/dfs.cpp`	Yes	Implement the DFS algorithm
`src/rdfs.cpp`	Yes	Implement the Recursive DFS algorithm
`stc/articulation_and_bridge.cpp`	Yes	Implement the searching Articulation Points and Bridges algorithm
`src/main.cpp`	No	Implement the main program
`Makefile.windows`	No	Define compilation rules used on Windows
`Makefile.linux`	No	Define compilation rules used on Linux/Mac OS

2. Implementation

a. Queue, Stack, and Graph Usages

Before we start implement BFS and DFS, we first investigate the usages of ***Queue, Stack***, and ***Graph***. These classes have been already implemented in the ***src/queue.cpp***, ***src/stack.cpp***, and ***src/graph.cpp***.

Class	***Method***	***Meaning***	***Sample Code***
Queue	Queue	Initialize an empty queue	Queue<int> queue;
	Push	Push a new value into the tail of the queue	queue.push(u);
	Pop	Pop out the head value of the queue	u = queue.pop();
	Empty	Check the current queue is either empty or not (return true if it's empty).	queue.empty()
Stack	Stack	Initialize an empty stack	Stack<int> stack;
	Push	Push a new value into the head of the stack	stack.push(u);
	Pop	Pop out the head value of the stack	u = stack.pop();
	Empty	Check the current stack is either empty or not (return true if it's empty).	stack.empty()
Graph	Graph	Initialize a graph with n vertices	Graph G(6);
	insertEdge	Insert an undirected (or directed) edge into the graph	G.insertEdge(0, 1); G.insertEdge(0, 1, directed=true);

b. Breadth First Search

In this section, you are required to implement the BFS algorithm by the queue. The definition of BFS is defined in ***include/search.hpp*** and you have to implemented BFS in ***src/bfs.cpp.***

```
int bfs(Graph &G, int start, int destination, int numberOfBuilding, std::vector<int> &path) {
    // Please ignore the parameter of numberOfBuilding, this parameter is only used in DFS and Recursive
DFS
    // Return the number of shortest path

    int N = G.n; // Number of nodes in the graph
    //YOUR CODE HERE

    // Finally return the number of shorest paths
    return 0; // Modify this return value to the correct one
}
```

Given a graph G, a start vertex, and a destination vertex, you have to implement the BFS algorithm to find a path from "start" to "destination". Please ignore the argument numberOfBuilding in this question. You have to implement the BFS algorithm to find the shortest path and store the path from start to destination into std::vector<int> &path. In addition, in your BFS, you ***have to return the number of shortest paths***, i.e., the number of possible paths that has the same number of buildings with the shortest path one. If you cannot return the correct number of shortest paths, you will not receive a full grade in this question.

c. Depth First Search

In this section, you are required to implement the DFS algorithm by the stack. The definition of DFS is defined in ***include/search.hpp*** and you have to implemented DFS in ***src/dfs.cpp.***

```
int dfs(Graph &G, int start, int destination, int numberOfBuilding, std::vector<int> &path) {
    int N = G.n; // Number of nodes in the graph
    //YOUR CODE HERE


    return 0; //You do not need to count the number of paths in this question, therefore, just simply return 0
}
```

Given a graph G, a start vertex, and a destination vertex, you have to implement the DFS algorithm without recursion to find a path from "start" to "destination". You have to implement the DFS algorithm to find the path and store the path from start to destination into std::vector<int> &path. In addition, in your DFS, ***the number of buildings/nodes in your path produced by DFS cannot be more than*** *numberOfBuilding* (including the start and destination nodes). If the number of nodes in your path is higher than numberOfBuilding, you will not receive a full grade in this question.

d. Recursive Depth First Search

In this section, you are required to implement the Recursive DFS algorithm by the stack. The definition of Recursive DFS is defined in ***include/search.hpp*** and you have to implemented Recursive DFS in ***src/rdfs.cpp.***

```
int rdfs(Graph &G, int start, int destination, int numberOfBuilding, std::vector<int> &path) {
    int N = G.n; // Number of nodes in the graph


    // YOUR CODE HERE


    return 0; //You do not need to count the number of paths in this question, therefore, just simply return 0
}
```

Given a graph G, a start vertex, and a destination vertex, you have to implement the Recursive DFS algorithm using recursion to find a path from "start" to "destination". You have to

implement the Recursive DFS algorithm to find the path and store the path from start to destination into std::vector<int> &path. In addition, in your Recursive DFS, ***the number of buildings/nodes in your path produced by Recursive DFS cannot be more than numberOfBuilding*** (including the start and destination nodes). If the number of nodes in your path is higher than numberOfBuilding, you will not receive a full grade in this question.

e. Articulation Points and Bridges (Graduate Student Only)

In this section, you are required to implement the algorithm to finds the articulation points and bridges. ***A node in the graph is an articulation point if and only if we remove this node out of the graph, the graph will be not connected any more*** (A graph is connected if and only if there is at least one path between any pair of nodes in the graph). ***An edge in the graph is a bridge if and only if we remove this edge out of the graph, the graph will be not connected any more.*** Additionally***, you have to implement your algorithm so that the time complexity of your algorithm is O(N) where N is the number of nodes.*** If the time complexity of your algorithm is not O(N), you will not receive a full grade. The practical application of this algorithm is to determine which edges or nodes that can be remove out of the graph while maintaining the graph as the connected graph. For example, when we would like to perform some construction on buildings and roads, we need to avoid the buildings or roads that are the articulation points or bridges in the graph so that we still maintain the traffic flow on campus. The definition of algorithm is defined in ***include/search.hpp*** and you have to implement this algorithm in ***src/articulation_and_bridge.cpp.***

```
void findArticulationPointsAndBridges(Graph &G, std::vector<int> &articulationPoints,
std::vector<std::pair<int, int> > &bridges) {
    int n = G.n; // Number of Nodes
    // YOUR CODE HERE
}
```

Given a graph G, you have to implement the algorithm to find the list of articulation points and a list of bridges. The list of articulation points stored into std::vector<int> &articulationPoints is a list of integers which are the node index. The list of bridges stored into std::vector<std::pair<int, int> > &bridges is a list of pairs of integers. Each pair consists of two nodes representing the edge.

f. Main Program

The main program is fully implemented in ***src/main.cpp***. This program contains the unit tests to test your implementation. This program also contains the visualization of the searching results on the campus map.

g. Compilation and Testing

In this homework, we use the same environment (Make, GCC/G++, OpenCV Library) used in the previous homework to compile the source code.

There are two files named ***Makefile.windows*** and ***Makefile.linux***. If you are using Windows, rename ***Makefile.windows*** to ***Makefile***. If you are using Linux or Mac OSX, rename ***Makefile.linux*** to ***Makefile***. If you do not want to use OpenCV for visualization,

open file ***Makefile*** and edit line ***OPENCV=1*** to ***OPENCV=0.*** Please noted that if you are using OpenCV 4+, you may have to edit lien ***OPENCV4=0*** to ***OPENCV4=1***. To compile the source code, you open the terminal and go to the source code folder and type **make** (or **mingw32-make** if you are using Windows)**.** If you compile successfully, it will give you no errors and output a folder named bin that contains the executable file.

For Windows users, if you would like to compile with OpenCV, you have to copy the following files from ***C:\opencv-3.4.13\build\bin*** to your Homework folder:

- libopencv_core3413.dll
- libopencv_highgui3413.dll
- libopencv_imgcodecs3413.dll
- libopencv_imgproc3413.dll