|  |  |  |
| --- | --- | --- |
| **American University of Sharjah**  College of Engineering  Dept. of Computer Science &  Engineering  P. O. Box 26666  Sharjah, UAE |  | **Instructor:** Dr. Michel Pasquier  **Lab Instructor:** Praveena Kolli  **Office:** EB2-126  **Phone**: 971-6-5152352  **Email**: pkolli@aus.edu  **Semester**: Spring 2021 |

**CMP 305 – Data Structures and Algorithms**

**Route-Planning Project**

***Objectives:***

* To develop a real *application* using Stack, Queue, and other data structures.
* To *implement* Depth-First Search and Breadth-First Search traversals.
* To *implement* Dijkstra’s search algorithm using a Priority Queue.

***Due date:*** As announced on *iLearn.*

**Planning Routes on a Map Using Stacks and Queues**

Write a route-planning program, with the following functionality, exactly *as specified.* Note that further advice and implementation details are provided later throughout this document.

App menu: The program prints to the console a menu presenting the user with the *9 options* described hereafter. It performs whichever operation is selected by the user, prints the results and other appropriate output accordingly, then prints the menu again – until Quit is chosen.

1. Load map: The program *reads a map representation* from a plain text file and loads it into an appropriate data structure. If successful, it displays the map, as depicted hereafter in option #2. Otherwise, it prints some explicit error message e.g., “error: map file not found”. Sample code for reading a map from a text file is provided in Appendix 1.

2. Display map: The program *prints* to the console the currently *loaded map,* similar to the below*.* If the start and/or goal locations are defined, it displays them as well, as illustrated in options #3-4. If no map is loaded, it prints an error message e.g., “error: no map available.”

+---+---+---+---+---+

| A | D F |

+ + + +---+---+

| | | E |

+ + +---+---+ +

| | | |

+ + + +---+ +

| B | |

+---+---+---+ +---+

| C G |

+---+---+---+---+---+

3. Set start: The program prompts the user for a *location name* i.e., a single letter, and reads in the user input. If the given name is valid, based on existing locations on the map, it prints to the console the current map with the *start location* highlighted with the ‘(’ and ‘)’ symbols, as depicted hereafter on the left. If the location name is invalid, it prints an appropriate error message e.g., “error: location name must match one location on the map”.

+---+---+---+---+---+ +---+---+---+---+---+

|(A)| D F | |(A)| D F |

+ + + +---+---+ + + + +---+---+

| | | E | | | | E |

+ + +---+---+ + + + +---+---+ +

| | | | | | | |

+ + + +---+ + + + + +---+ +

| B | | | B | |

+---+---+---+ +---+ +---+---+---+ +---+

| C G | | C {G}|

+---+---+---+---+---+ +---+---+---+---+---+

4. Set goal: Similarly, the program prompts the user for a *location name* and reads in the input. If the name is valid, it prints to the console the current map with the *goal location* highlighted with the ‘{’ and ‘}’ symbols, as shown above on the right. If not, it prints an error message.

5. Find path with DFS: The program *runs* a *Depth-First Search* algorithm to find a path from the start location to the goal location. Further explanations about this search process are provided in the following pages. If any of the map or the start location or the goal location is not defined, the program prints an appropriate error message. If a solution path is found, it *prints* the *name* (if any) and the *coordinates* of all *locations along the path,* including start and goal locations e.g., as follows. If there is no solution, it prints an error message instead e.g., “error: no path found.”

solution path: A(1,1) (2,1) (3,1) B(4,1) (4,2) (4,3)

(3,3) (3,4) (3,5) (4,5) (4,4) (5,4) G(5,5)

6. Find path with BFS: Like the above, but the program *runs* a *Breadth-First Search* algorithm to find a path from start to goal. This process is further elaborated in the following pages. Note that, depending on the map, DFS and BFS algorithms may find different solution paths. If successful, the program *prints the path* coordinates, otherwise it prints an error message.

7. Find path with DA: Similar to the above, but the program now *runs* *Dijkstra’s algorithm* to find a path from start to goal that *minimizes some cost,* as explained in the following pages. For instance, the travelling cost may vary with the terrain or elevation at each location. This requires adding cost information to the map, as illustrated below. Named locations always have a cost of zero. Note that, in this example, the shortest path and the minimum-cost path are not the same.

+---+---+---+---+---+

|(A)| D 1 1 F |

+ + + +---+---+

| 1 | 1 | 1 E 1 |

+ + +---+---+ +

| 2 | 1 | 3 3 2 |

+ + + +---+ +

| B 2 3 | 2 1 |

+---+---+---+ +---+

| C 1 2 3 G |

+---+---+---+---+---+

8. Display path: The program *prints* to the console *the map* and the last *path found,* as illustrated in the diagrams below. Locations on the solution path are shown with a circle (‘o’) for DFS or BFS, and with the cost for Dijkstra’s algorithm. Locations visited by the search algorithm but not on the path are represented with a dot (‘.’). Unvisited locations are left blank. If there is no solution path, the program prints an appropriate error message.

BFS algorithm Dijkstra’s algorithm

+---+---+---+---+---+ +---+---+---+---+---+

|(A)| . . . . | |(A)| D 1 . . |

+ + + +---+---+ + + + +---+---+

| o | . | . . . | | 1 | 1 | 1 E 1 |

+ + +---+---+ + + + +---+---+ +

| o | . | o o o | | 2 | 1 | . . 2 |

+ + + +---+ + + + + +---+ +

| o o o | o o | | B 2 . | 2 1 |

+---+---+---+ +---+ +---+---+---+ +---+

| . o {G}| | . 3 {G}|

+---+---+---+---+---+ +---+---+---+---+---+

9. Quit: The program *prints* some “goodbye” *message* to the console and terminates.

**Methodology of Route-Planning Search**

As explained in detail in our CMP 305 class, the methodology behind this route-planning process is that storing choices i.e., locations, in a LIFO *stack* implicitly follows the principle of a *Depth-First Search* (DFS) traversal, whereas storing them in a FIFO *queue* instead implicitly follows the principle of a *Breadth-First Search* (BFS) traversal. The search algorithm is therefore the *same* in both cases, except for the data structure used to store the explored map locations.

Accordingly, DFS explores the map by arbitrarily following a path until it reaches a dead end, then it backs up to the latest path choice, follows the alternative path in turn until it reaches another dead end, then it backs up again, etc. This search process stops when the goal is reached, or when all possible paths have been explored and there is no solution path from start to goal.

By contrast, a BFS traversal of the map will consider all paths equally, stepping through each one step at a time, progressing further until one of them reaches the goal, or there is no solution.

Below is another, larger sample maze where DFS and BFS produce different results. In fact, DFS itself will produce different paths depending in which order possible moves are considered.

In the case of Dijkstra’s algorithm, the logic and code are still the *same,* except that the data structure used to store the explored locations is now a *priority queue.* A cost function must therefore be defined that determines how good a solution is. The cost of a path is the total cost of all locations on the path, and the algorithm will find the path of minimal cost. Note that the cost of passing through a location is arbitrary, and depends solely on the application considered.

A typical scenario is where different map locations are made of different terrain types, such as concrete, sand, or water, with costs such as 1, 2, or 3, respectively, as illustrated earlier. Those numbers could also represent the elevation, etc. The path found by the algorithm will be the “best” i.e., which minimizes the cost function, and not necessarily the shortest.

Map BFS (always)

+---+---+---+---+---+---+---+---+ +---+---+---+---+---+---+---+---+

|(A) F | |(A) . . . . . . . |

+ +---+---+---+---+---+---+ + + +---+---+---+---+---+---+ +

| | C | | o | . . . . . . . |

+ + +---+---+---+---+---+---+ + + +---+---+---+---+---+---+

| | | o o o o o o o . |

+---+---+ +---+ +---+ +---+ +---+---+ +---+ +---+ +---+

| B | D | {G}| | . . . | . . | . o {G}|

+ +---+---+---+---+ +---+---+ + +---+---+---+---+ +---+---+

| E | | . . . E |

+---+---+---+---+---+---+---+---+ +---+---+---+---+---+---+---+---+

DFS (below first, then right…) DFS (right first, then below…)

+---+---+---+---+---+---+---+---+ +---+---+---+---+---+---+---+---+

|(A) F | |(A) o o o o o o o |

+ +---+---+---+---+---+---+ + + +---+---+---+---+---+---+ +

| o | C | | | o o o o o o o |

+ + +---+---+---+---+---+---+ + + +---+---+---+---+---+---+

| o o o | | o o o o o o . |

+---+---+ +---+ +---+ +---+ +---+---+ +---+ +---+ +---+

| o o o | D | o o {G}| | B | D | o {G}|

+ +---+---+---+---+ +---+---+ + +---+---+---+---+ +---+---+

| o o o o o o . . | | E |

+---+---+---+---+---+---+---+---+ +---+---+---+---+---+---+---+---+

A *priority queue* is one where elements are always sorted in order of priority values or costs, instead of FIFO for plain queues. The C++ STL library provides a priority\_queue class, which is as easy to use as its stack or queue classes. Sample code is given in Appendix 2. Note that, since BFS finds the path with the fewest steps and Dijkstra’s algorithm finds the path with the minimum cost, the two algorithms will perform identically if all costs are the same.

**Details of Implementation**

In your program, you should represent a map using two classes: *Map* and *Location.* A map i.e., a *Map* instance, contains chiefly a rectangular grid of cells i.e., *Location* objects, as well as its *width* and *height*. Location coordinates conventionally use the *(row,column)* format. The location in the upper left-hand corner has coordinates (1,1).

Each grid cell or location has at most four neighbors: left, right, above, and below. Any two neighboring cells are either connected by empty space or separated by an obstacle, as shown in the above map examples. Empty space between two cells allows moving from one to the other, in either direction. The start and goal locations are specified by the user as explained earlier.

As recapped previously, the process of planning a route involves continually selecting connected locations based on the current location until the goal is reached. Detailed pseudo-code is given (from lecture slides) that shows how to use a *stack* and a *queue* to implement *Depth-First Search* and *Breadth-First Search* algorithms, respectively (as explained in detail in class). *Dijkstra’s algorithm* is just like BFS with costs, as explained in the previous section. Your task is to complete the implementation as per the specs and test it successfully on various sample maps.

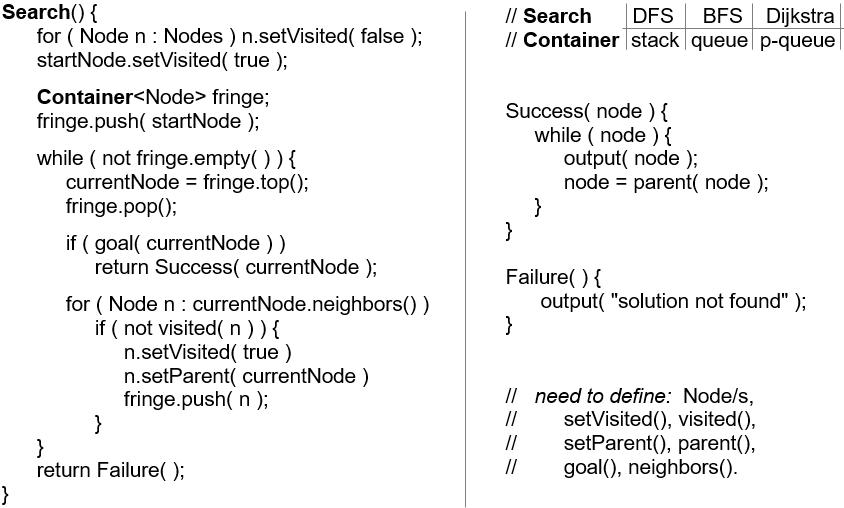
To this aim, you need to add some *attributes* to *Location* class instances, to store obstacle and neighbor information, and to mark a location as visited or not. As the given pseudo-code clearly shows, marking locations as visited is essential to *avoid* search *cycles* hence infinite loops. You also need to remember the location that was visited immediately before the current one, to allow retrieving and printing the solution path, as required in the application.

When implementing DFS and BFS search algorithms, you need to decide in which order the neighbor locations are explored. The order is arbitrary and does not really matter, but it must be consistent e.g., always check the above location first, then right, below, and left. As illustrated in the included map examples, the path found by DFS typically depends on this arbitrary order, whereas for both BFS and Dijkstra’s algorithm it does not.

Last but not least, note that thinking about this route-planning process and understanding how to use a stack, queue, or priority queue effectively to accomplish the task is an essential part of this project. Building a successful route-planning program will strengthen your comprehension of data structures, traversals, and search algorithms, improve your programming skills, and help you better understand tree and graph traversals as and when studied in subsequent chapters.

**Note:** Code matters. Remember: first, *make it work* i.e., meeting all specs and requirements, then *make it nice* i.e., readable and well structured, then *make it fast* i.e., efficient as well – both memory and performance wise. For a start, you should always name appropriately all attributes, variables, functions, classes, etc. and include comments as necessary for each logical block… In the end, you ought to fully test your program on the sample maps given to you, and preferably some others of your own making. Route maps are simple text files hence easy to edit.

**Pseudo code for the Search algorithm** (same as the slides)



**Appendix 1:** Sample code to read a map from a text file.

#include <iostream>

#include<fstream>

#include<string>

#include<cstring>

#include<cctype>

using namespace std;

int main() {

ifstream in("map.txt");

char str1[100], str2[100], str3[100];

in.getline(str1, 100);

in.getline(str2, 100);

in.getline(str3, 100);

int line = 1, cell = 1;

while (!in.eof()) {

int i = 0, j = 0, k = 1,cost=-1,cell =0;

char name=' ';

bool above, below, right, left;

cout << "line " << line << endl;

while (i < strlen(str1) - 1) {

above = below = right = left = true;

name =' ';

if (str1[i] == '+') i++; *// new path*

if (str1[i] == '-') above = false;

else if (str1[i] == ' ') above = true;

*// else error*

i = i + 3; // path left

if (str2[j] == '|') left = false;

else if (str2[j] == ' ') left = true;

*// else error*

j = j + 2; *// cost*

if (str2[j]!=' ')

if (isdigit(str2[j])) cost= (str2[j] - '0');

else { cost = 0; name = str2[j]; }

else cost = 0;

j = j +2; *//path right*

if (str2[j] == '|') right = false;

else if (str2[j] == ' ') right = true;

*// else error*

//path below

if (str3[k] == ' ') below = true;

else if (str3[k] == '-') below = false;

// else error

k = k + 4; // wall below

cout << "cell = " << cell++ << endl;

cout << "above: " << above << "\t";

cout << "below: " << below << endl;

cout << "right: " << right << "\t";

cout << "left: " << left << endl;

cout << "cost: " << cost << endl;

cout<<"name: "<<name<<endl;

cout << endl << endl;

}

strcpy(str1, str3);

cout << str1 << endl;

in.getline(str2, 100);

in.getline(str3, 100);

cout << str2 << endl;

cout << str3 << endl;

line++;

}

return 0;

}

**Appendix 2:** Sample code illustrating how to use a Priority Queue.

#include <iostream>

#include <queue>

#include <string>

using namespace std;

class student {

private:

string name;

int grade;

public:

student( string name, int grade ) : name(name), grade(grade) {}

string getName() const { return name; }

int getGrade() const { return grade; }

friend ostream &operator<<( ostream &output, const student &s ) {

output << "Student : " << s.name << "\t Grade : " << s.grade;

return output;

}

};

struct compareGrades { *// defining the comparison operator*

bool operator() (student const& s1, student const& s2) {

return s1.getGrade() < s2.getGrade();

}

};

int main() {

priority\_queue<student, vector<student>, compareGrades> pq;

pq.push( student("Ahmad", 45) );

pq.push( student("John", 90) );

pq.push( student("Ali", 70) );

pq.push( student("Mohd", 97) );

cout << "priority queue of students, highest grades first ::" << endl;

while (!pq.empty()) {

student s = pq.top();

pq.pop();

cout << s << "\n";

}

return 0;

}