[CENG 315 All Sections] Algorithms

Dashboard / My courses / 571 - Computer Engineering / CENG 315 All Sections / December 5 - December 11 / THE5

Description Submission Edit Submission view

THE5

Available from: Friday, December 9, 2022, 11:59 AM
Due date: Friday, December 9, 2022, 11:59 PM
Requested files: the5.cpp, test.cpp (Download)
Type of work: Individual work

Problem:

In this exam, you are given a maze consisting of various rooms connected to each other via a direct door. In one of those rooms, there is a secret treasure and your purpose is to find that treasure. You do not know in which room the treasure is placed. Therefore starting from the entrance, you search for the treasure walking through room-by-room. During the search, you print the path that you follow until you reach the treasure.

In the mysterious maze, you may encounter with strange items. Find the treasure ☺

Here are the details of the problem structure:

- The maze is actually a connected undirected graph. Each room is a node of the graph. If a room is connected to an other room, there is an edge between those two rooms.
- Each room is defined in the type of ***Struct Room***. This structure has 3 components:
  - *int id:* Each room has a unique id.
  - *char content*: Shows the content of the room. All rooms have the content of '-' character except the room containing the treasure. That room has the content of '*' character representing the treasure.
  - *vector<Room*> neighbors*: Holds a pointer for the rooms which are connected to the current room via a door.
- If a Room Y is defined as a neighbor to Room X, then you can be sure that Room X is also defined as a neighbor to Room Y in its neighborhood vector.
- The rooms of the maze will be given to the function as in the type of *vector<Room*>*.
- You are expected to return the path as vector of ids of rooms which are visited.

Here are the details of how to search/traverse the maze:

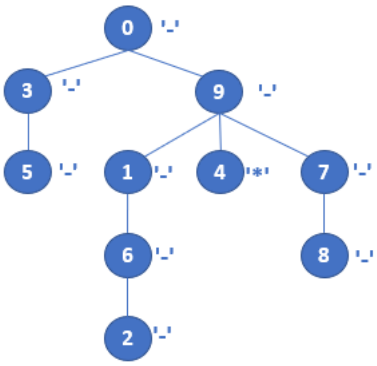
- You will actually do a kind of DFS. You will start from the first room (first means the firstly defined room, not the room with the first id) to traverse. You will pass to one of its neighbor rooms, and then to one of the neighbors of it, and to one of the neighbors of it, and so on. As you pass through a new room each time, you will add the id of that room to the output path. Upto here, it is exactly DFS.

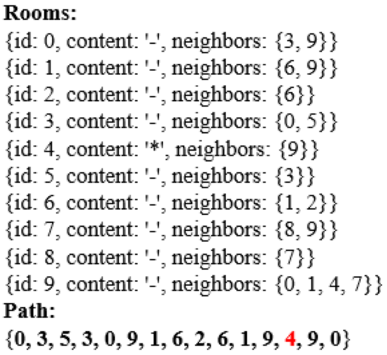
- When you come to an end, that is a room with no unvisited neighbor, then you should turn back. While going back, you should also add the ids of the rooms that you need to visit one more time into the output path. For instance, assume that Room 5 is neighbor to Room 12 and assume that you come to Room 5 at some point and have not visited Room 12, yet. Also assume Room 12 is not neighbor to any other nonvisited room. Then, in your output path a pattern like the following have to exist: 5, 12, 5 . That means "you pass through Room 5, then Room 12, then you turn back to Room 5 again since there is not left any nonvisited room neighbor to Room 12. In short, in addition to usual DFS output, you are expected to print the nodes at each time you visit.
- When you find the treasure (The Room whose content is '*'), you should turn back totally. That is, you need to go back over the route that you follow. You should not go into any new room. During the going back, you again add the ids of the rooms that you visit.
- For the neighbor selection, you need to follow the order in which the rooms are defined as a neighbor for that Room. For instance, if the neighbors of Room 5 are ordered as <Room 12, Room 7, Room 9> inside the neighbor vector, then you should select Room 12 first. After completing Room 12, you should continue from Room 7 and next from Room 9. Assume that Room 7 was visited before. Then you should follow Room 9 after completing the Room 12 and its neighbors. In other words, you should skip Room 7.
- There will always be exactly one room including the treasure.

Example IO:

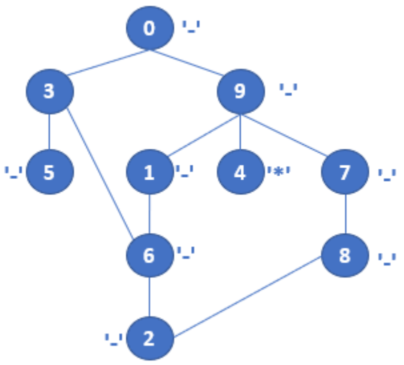
Please pay attention to the ordering of the neighbors for each node. It affects the resulting path!

EXAMPLE-1



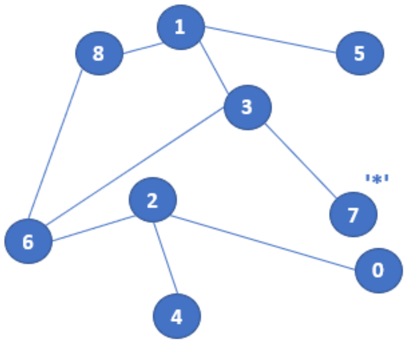

Rooms:
{id: 0, content: '-', neighbors: {3, 9}}
{id: 1, content: '-', neighbors: {6, 9}}
{id: 2, content: '-', neighbors: {6}}
{id: 3, content: '-', neighbors: {0, 5}}
{id: 4, content: '*', neighbors: {9}}
{id: 5, content: '-', neighbors: {3}}
{id: 6, content: '-', neighbors: {1, 2}}
{id: 7, content: '-', neighbors: {8, 9}}
{id: 8, content: '-', neighbors: {7}}
{id: 9, content: '-', neighbors: {0, 1, 4, 7}}
Path:
{0, 3, 5, 3, 0, 9, 1, 6, 2, 6, 1, 9, 4, 9, 0}

EXAMPLE-2



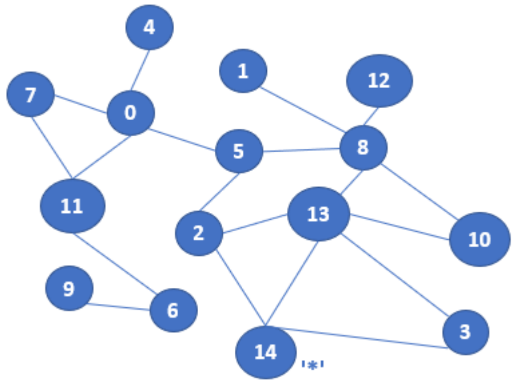
Rooms:
{id: 0, content: '-', neighbors: {3, 9}}
{id: 1, content: '-', neighbors: {6, 9}}
{id: 2, content: '-', neighbors: {6, 8}}
{id: 3, content: '-', neighbors: {0, 5, 6}}
{id: 4, content: '*', neighbors: {9}}
{id: 5, content: '-', neighbors: {3}}
{id: 6, content: '-', neighbors: {1, 2, 3}}
{id: 7, content: '-', neighbors: {8, 9}}
{id: 8, content: '-', neighbors: {2, 7}}
{id: 9, content: '-', neighbors: {0, 1, 4, 7}}
Path:
{**0, 3, 5, 3, 6, 1, 9, 4, 9, 1, 6, 3, 0**}

EXAMPLE-3



Rooms:
{id: 0, content: '-', neighbors: {2}}
{id: 1, content: '-', neighbors: {8, 5, 3}}
{id: 2, content: '-', neighbors: {6, 4, 0}}
{id: 3, content: '-', neighbors: {1, 7, 6}}
{id: 4, content: '-', neighbors: {2}}
{id: 5, content: '-', neighbors: {1}}
{id: 6, content: '-', neighbors: {8, 3, 2}}
{id: 7, content: '*', neighbors: {3}}
{id: 8, content: '-', neighbors: {1, 6}}
Path:
{**0, 2, 6, 8, 1, 5, 1, 3, 7, 3, 1, 8, 6, 2, 0**}

EXAMPLE-4



Rooms:
{id: 0, content: '-', neighbors: {7, 4, 11, 5}}
{id: 1, content: '-', neighbors: {8}}
{id: 2, content: '-', neighbors: {13, 5, 14}}
{id: 3, content: '-', neighbors: {14, 13}}
{id: 4, content: '-', neighbors: {0}}
{id: 5, content: '-', neighbors: {0, 8, 2}}
{id: 6, content: '-', neighbors: {9, 11}}
{id: 7, content: '-', neighbors: {11, 0}}
{id: 8, content: '-', neighbors: {10, 5, 1, 12, 13}}
{id: 9, content: '-', neighbors: {6}}
{id: 10, content: '-', neighbors: {8, 13}}
{id: 11, content: '-', neighbors: {0, 6, 7}}
{id: 12, content: '-', neighbors: {8}}
{id: 13, content: '-', neighbors: {8, 2, 3, 14, 10}}
{id: 14, content: '*', neighbors: {3, 2, 13}}
Path:
{0, 7, 11, 6, 9, 6, 11, 7, 0, 4, 0, 5, 8, 10, 13, 2, 14, 2, 13, 10, 8, 5, 0}

Constraints:

- Maximum number of nodes in a maze graph will be **10000**.

Evaluation:

- After your exam, black box evaluation will be carried out. You will get full points if your function returns the correct result without exceeding time limit.

Specifications:

- There are **only 1 task** to be solved in **12 hours** in this take home exam.
- You will implement your solutions in **the5.cpp** file.
- Do **not** change the first line of **the5.cpp**, which is **#include "the5.h"**
- *<iostream>*, *<climits>*, *<vector>*, *<string>*, *<stack>*, *<queue>* are included in "the5.h" for your convenience.
- Do **not** change the arguments and return **types** of the function **maze_trace().** (You should change return **value**, on the other hand.)
- Do **not** include any other library or write include anywhere in your **the5.cpp** file (not even in comments)

Compilation:

- You are given **test.cpp** file to **test** your work on **ODTÜClass** or your **locale**. You can and you are encouraged to modify this file to add different test cases.
- If you want to **test** your work and see your outputs you can **compile and run** your work on your locale as:

```
>g++ test.cpp the5.cpp -Wall -std=c++11 -o test

> ./test
```

- You can test your **the5.cpp** on virtual lab environment. If you click **run**, your function will be compiled and executed with **test.cpp**. If you click **evaluate**, you will get a feedback for your current work and your work will be **temporarily** graded for **limited** number of inputs.
- The grade you see in lab is **not** your final grade, your code will be re-evaluated with **completely different** inputs after the exam.

The system has the following limits:

- a maximum execution time of 32 seconds
- a 192 MB maximum memory limit
- an execution file size of 1M.
- Solutions with longer running times will not be graded.
- If you are sure that your solution works in the expected complexity constrains but your evaluation fails due to limits in the lab environment, the constant factors may be the problem.

```
vector<int> maze_trace(vector<Room*> maze);
```

Requested files

the5.cpp

```
#include "the5.h"

/*
    in the5.h "struct Room" is defined as below:

    struct Room {
        int id;
        char content;
        vector<Room*> neighbors;
    };

*/


vector<int> maze_trace(vector<Room*> maze) {

    vector<int> path;

    //your code here


    return path; // this is a dummy return value. YOU SHOULD CHANGE THIS!
}



```

test.cpp

```
// this file is for you for testing purposes, it won't be included in evaluation.

#include <iostream>
#include <random>
#include <ctime>
#include <cstdlib>
#include "the5.h"

void randomGraph(vector<Room*>& maze, int size) {

    int numOfVerts = size;
    int degree = 4;
    int numOfEdges = (degree * numOfVerts) / 3;
    numOfEdges = rand() % numOfEdges;
    numOfEdges = numOfEdges < numOfVerts ? numOfVerts : numOfEdges;

    // generate rooms
    for (int i = 0; i < numOfVerts; i++)
    {
        Room* room = new Room;
        room->id = i;
        room->content = '-';
        maze.push_back(room);
    }


    int r = rand() % numOfVerts;
    maze[r]->content = '*';

    // generate edges
    vector<vector<int>> edges;

    for (int i = 0; i < numOfEdges; ) {
        int v1 = rand() % numOfVerts;
        int v2 = rand() % numOfVerts;

        if (v1 == v2)
            continue;

        else {
            bool retry = false;
            for (int j = 0; j < edges.size(); j++) {
                if ((edges[j][0] == v1 && edges[j][1] == v2) || (edges[j][0] == v2 && edges[j][1] == v1)) {
                    retry = true;
                    break;
                }
            }

            if (retry)
                continue;
            if (maze[v1]->neighbors.size() == degree || maze[v2]->neighbors.size() == degree)
                continue;

            vector<int> edge;
            edge.push_back(v1);
            edge.push_back(v2);
            edges.push_back(edge);
            maze[v1]->neighbors.push_back(maze[v2]);
            maze[v2]->neighbors.push_back(maze[v1]);
            i++;
        }
    }

    // define components
    vector<vector<int>> components; // disconnected subgraphs
    for (int i = 0; i < numOfVerts; i++) {
        vector<int> component;
        component.push_back(i);
        component.push_back(i);
        components.push_back(component);
    }

    for (int i = 0; i < numOfEdges ; i++) {
        int v1 = edges[i][0];
        int v2 = edges[i][1];
        if (components[v1][0] == components[v2][0])
            continue;
        else {
            int c1 = components[v1][0];
            int c2 = components[v2][0];

            for (int c = 1; c < components[c2].size(); c++) {
                components[c1].push_back(components[c2][c]);
                components[components[c2][c]][0] = c1;
            }
        }
    }

    vector<int> component_ids;
    for (int i = 0; i < numOfVerts; i++) {
        if (components[i][0] == i)
            component_ids.push_back(i);
    }

    // make connected
    for (int i = 1; i < component_ids.size(); i++) {
        int c1 = component_ids[0];
        int c2 = component_ids[i];

        int ind1 = rand() % (components[c1].size()-1) + 1;
        int ind2 = rand() % (components[c2].size()-1) + 1;

        int v1 = components[c1][ind1];
        int v2 = components[c2][ind2];

        maze[v1]->neighbors.push_back(maze[v2]);
        maze[v2]->neighbors.push_back(maze[v1]);

        for (int c = 1; c < components[c2].size(); c++)
            components[c1].push_back(components[c2][c]);
    }

```

```
}


void manualGraph(vector<Room*>& maze, int size)
{
    for (int i = 0; i < size; i++)
    {
        Room* room = new Room;
        room->id = i;
        room->content = '-';
        maze.push_back(room);
    }

    // Do not forget to change the size at the beginning of the test()

    // EXAMPLE-1
    /*
    maze[4]->content = '*';

    maze[0]->neighbors.push_back(maze[3]);
    maze[0]->neighbors.push_back(maze[9]);
    maze[1]->neighbors.push_back(maze[6]);
    maze[1]->neighbors.push_back(maze[9]);
    maze[2]->neighbors.push_back(maze[6]);
    maze[3]->neighbors.push_back(maze[0]);
    maze[3]->neighbors.push_back(maze[5]);
    maze[4]->neighbors.push_back(maze[9]);
    maze[5]->neighbors.push_back(maze[3]);
    maze[6]->neighbors.push_back(maze[1]);
    maze[6]->neighbors.push_back(maze[2]);
    maze[7]->neighbors.push_back(maze[8]);
    maze[7]->neighbors.push_back(maze[9]);
    maze[8]->neighbors.push_back(maze[7]);
    maze[9]->neighbors.push_back(maze[0]);
    maze[9]->neighbors.push_back(maze[1]);
    maze[9]->neighbors.push_back(maze[4]);
    maze[9]->neighbors.push_back(maze[7]);
    */

    // EXAMPLE-2
    /*
    maze[4]->content = '*';

    maze[0]->neighbors.push_back(maze[3]);
    maze[0]->neighbors.push_back(maze[9]);
    maze[1]->neighbors.push_back(maze[6]);
    maze[1]->neighbors.push_back(maze[9]);
    maze[2]->neighbors.push_back(maze[6]);
    maze[2]->neighbors.push_back(maze[8]);
    maze[3]->neighbors.push_back(maze[0]);
    maze[3]->neighbors.push_back(maze[5]);
    maze[3]->neighbors.push_back(maze[6]);
    maze[4]->neighbors.push_back(maze[9]);
    maze[5]->neighbors.push_back(maze[3]);
    maze[6]->neighbors.push_back(maze[1]);
    maze[6]->neighbors.push_back(maze[2]);
    maze[6]->neighbors.push_back(maze[3]);
    maze[7]->neighbors.push_back(maze[8]);
    maze[7]->neighbors.push_back(maze[9]);
    maze[8]->neighbors.push_back(maze[2]);
    maze[8]->neighbors.push_back(maze[7]);
    maze[9]->neighbors.push_back(maze[0]);
    maze[9]->neighbors.push_back(maze[1]);
    maze[9]->neighbors.push_back(maze[4]);
    maze[9]->neighbors.push_back(maze[7]);
    */

    // EXAMPLE-3
    /*
    maze[7]->content = '*';

    maze[0]->neighbors.push_back(maze[2]);
    maze[1]->neighbors.push_back(maze[8]);
    maze[1]->neighbors.push_back(maze[5]);
    maze[1]->neighbors.push_back(maze[3]);
    maze[2]->neighbors.push_back(maze[6]);
    maze[2]->neighbors.push_back(maze[4]);
    maze[2]->neighbors.push_back(maze[0]);
    maze[3]->neighbors.push_back(maze[1]);
    maze[3]->neighbors.push_back(maze[7]);
    maze[3]->neighbors.push_back(maze[6]);
    maze[4]->neighbors.push_back(maze[2]);
    maze[5]->neighbors.push_back(maze[1]);
    maze[6]->neighbors.push_back(maze[8]);
    maze[6]->neighbors.push_back(maze[3]);
    maze[6]->neighbors.push_back(maze[2]);
    maze[7]->neighbors.push_back(maze[3]);
    maze[8]->neighbors.push_back(maze[1]);
    maze[8]->neighbors.push_back(maze[6]);
    */

    // EXAMPLE-4
    maze[14]->content = '*';

    maze[0]->neighbors.push_back(maze[7]);
    maze[0]->neighbors.push_back(maze[4]);
    maze[0]->neighbors.push_back(maze[11]);
    maze[0]->neighbors.push_back(maze[5]);
    maze[1]->neighbors.push_back(maze[8]);
    maze[2]->neighbors.push_back(maze[13]);
    maze[2]->neighbors.push_back(maze[5]);
    maze[2]->neighbors.push_back(maze[14]);
    maze[3]->neighbors.push_back(maze[14]);
    maze[3]->neighbors.push_back(maze[13]);
    maze[4]->neighbors.push_back(maze[0]);
    maze[5]->neighbors.push_back(maze[0]);
    maze[5]->neighbors.push_back(maze[8]);
    maze[5]->neighbors.push_back(maze[2]);
    maze[6]->neighbors.push_back(maze[9]);
    maze[6]->neighbors.push_back(maze[11]);
    maze[7]->neighbors.push_back(maze[11]);
    maze[7]->neighbors.push_back(maze[0]);
    maze[8]->neighbors.push_back(maze[10]);
    maze[8]->neighbors.push_back(maze[5]);
    maze[8]->neighbors.push_back(maze[1]);
    maze[8]->neighbors.push_back(maze[12]);
    maze[8]->neighbors.push_back(maze[13]);
    maze[9]->neighbors.push_back(maze[6]);
    maze[10]->neighbors.push_back(maze[8]);
    maze[10]->neighbors.push_back(maze[13]);
    maze[11]->neighbors.push_back(maze[0]);
    maze[11]->neighbors.push_back(maze[6]);
    maze[11]->neighbors.push_back(maze[7]);
    maze[12]->neighbors.push_back(maze[8]);
    maze[13]->neighbors.push_back(maze[8]);
    maze[13]->neighbors.push_back(maze[2]);
    maze[13]->neighbors.push_back(maze[3]);
    maze[13]->neighbors.push_back(maze[14]);
    maze[13]->neighbors.push_back(maze[10]);
    maze[14]->neighbors.push_back(maze[3]);
    maze[14]->neighbors.push_back(maze[2]);
    maze[14]->neighbors.push_back(maze[13]);

}


void printGraphInLine(vector<Room*> maze){

    std::cout << "{\n";
    for(int i = 0; i < maze.size(); i++){
        std::cout << " ROOM " << i << "," << std::endl;
        std::cout << "     content: '" << maze[i]->content << "'," << std::endl;
        std::cout << "     neighbors: ";
        for (int j = 0; j < maze[i]->neighbors.size(); j++) {
            std::cout << maze[i]->neighbors[j]->id;
            if (j == maze[i]->neighbors.size() - 1)
                std::cout << std::endl;
            else
                std::cout << ", ";
        }
    }
    std::cout << "}" << std::endl;

}

void printVectorInLine(vector<int> output) {

    for(int i = 0; i < output.size(); i++) {
        std::cout << output[i];
        if (i == output.size() - 1)
            continue;
        else
            std::cout << ", ";
    }
    std::cout << endl;

}



void test(){
    clock_t begin, end;
    double duration;

    int size = 15;
    vector<int> path;
    vector<Room*> maze;
    //randomGraph(maze, size);
    manualGraph(maze, size);

    if ((begin = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    path = maze_trace(maze);

    if ((end = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    duration = ((double) end - begin) / CLOCKS_PER_SEC;
    std::cout << "Duration: " << duration << " seconds." << std::endl;

    std::cout << "Given maze: "<< std::endl;
    printGraphInLine(maze);

    std::cout << "\nNumber of Rooms: \n" << size << std::endl;

    std::cout << "\nMaze Trace: " << std::endl;
    std::cout << "\nReturned path :";
    printVectorInLine(path);

    std::cout << "---------------------------------------------------";
    std::cout << "\n" << std::endl;

}

int main()
{
    srandom(time(0));
    test();
    return 0;
}
```

VPL

ODTÜClass Archive
2021-2022 Summer
2021-2022 Spring
2021-2022 Fall
2020-2021 Summer
2020-2021 Spring
2020-2021 Fall
2019-2020 Summer
2019-2020 Spring
2019-2020 Fall
Class Archive
ODTÜClass 2021-2022 Summer School

Get the mobile app