CS 201, Fall 2023
Homework Assignment 4

Due: 23:59, December 22, 2023

Assist the young rabbit in locating its mother!

A charming little rabbit is separated from her mom in a zoo, which comprises numerous cages, each housing a different wild animal. To reunite with its mother, the rabbit must navigate through specific cages interconnected in a directed manner. Unfortunately, the rabbit may be hunted by the animal inside each cage. These events are represented with a *cage weight* that corresponds to *the probability that the rabbit is not hunted by the animal in that cage*. Given these probabilities and the connections between the cages, your task is to aid the rabbit in discovering both the safest and the most dangerous paths (sequences of cages) in the zoo. Your solution **MUST** utilize a non-recursive approach, employing a **stack** for searching the desired paths.

A sample graph illustrating the scenario is depicted in Figure 1. In this figure, every vertex corresponds to a cage, and each edge represents a connection from one cage to another. Furthermore, each vertex is assigned a weight $w \in (0, 1]$, denoting the probability that the rabbit will not be hunted by the animal in that cage. The *safest path* is defined as the connected sequence of vertices for which the product of the vertex weights is maximized. The *most dangerous* path is defined similarly, where the product of the vertex weights is minimized. The paths found must not have any cycles.

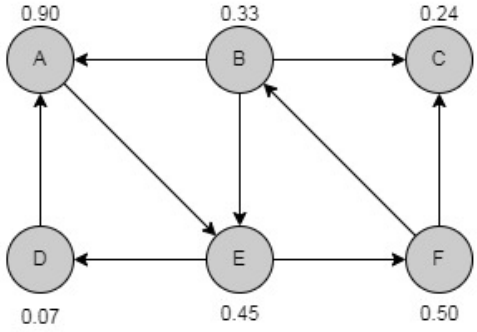

Figure 1: Sample of a Zoo Map.

We will provide you with two files: `cageFile` and `zooFile`. The "`cageFile`" includes the names of the cages and their associated probabilities of not being hunted. Each line has a cage name and the corresponding probability separated by a semi-colon. Cage names are supposed to be unique and consist of only English letters, and they can be given in any arbitrary order. The file for the example graph of Figure 1 is:

C;0.24
D;0.07
F;0.50
A;0.90
E;0.45
B;0.33

The "`zooFile`" includes the edge list of the zoo graph where each line describes an edge with its origin cage and destination cage, separated by a semi-colon. The edge list of the zoo can be given in any arbitrary order. You can assume that there is at most one directed edge between any pair of cages. The following shows the zoo file for the example graph of Figure 1:

D;A
B;A
E;D
A;E
B;E
F;B
B;C
E;F
F;C

Your solution must be implemented in a class called **ZooMap**. Below is the required public part of the `ZooMap` class. The interface for the class must be written in a file called **ZooMap.h** and its implementation must be written in a file called **ZooMap.cpp**. You can define additional public and private member functions and data members in this class. You can also define additional classes in your solution.

```
class ZooMap {

public:
    ZooMap(const string cageFile, const string zooFile);
    ~ZooMap();

    void displayAllCages() const;
    void displayAdjacentCages(const string cageName) const;
    void displayZooMap() const;

    void findSafestPath(const string startCage, const string endCage);
    void findMostDangerousPath(const string startCage, const string endCage);

};
```

The member functions are defined as follows:

ZooMap: Constructor. Reads the zoo information from the files `cageFile` and `zooFile` given as parameters and stores the zoo graph. The graph edges must be stored by using adjacency lists using linked lists. The zoo graph will not change once the `ZooMap` object is constructed by using the information provided in the given files. You can also assume that there will be at least one cage and one edge in the input files (i.e., the files will not be empty) and the file contents are correct.

displayAllCages: Displays all cages in the zoo graph. The cage names must be displayed in ascending alphabetical order.

displayAdjacentCages: Displays the cages adjacent to the given cage (i.e., all cages that have a connection from the given cage). The adjacent cage names must be displayed in ascending alphabetical order. You can assume that the given cage name exists in the zoo.

displayZooMap: Displays all adjacent cages for all cages in the graph. In other words, this function displays the whole zoo graph. All cage names (origin cages and their adjacent cages) must be displayed in ascending alphabetical order.

findSafestPath: Finds and displays the safest path from the start cage to the end cage. Safest path is defined as the path that has the highest probability that the rabbit may not be hunted, if it travels that path. It will also compute the corresponding probability (not being hunted) for the safest path. Assume that there would be at most one safest path between any pair of cages (the inputs will be provided by complying with this assumption). You can also assume that the given cage names exist in the zoo.

findMostDangerousPath: Finds and displays the most dangerous path from the start cage to the end cage. Most dangerous path is defined as the path that has the lowest probability that the rabbit may not be hunted, if it travels that path. It will also compute the corresponding probability (not being hunted) for the most dangerous path. Assume that there would be at most one most dangerous path between any pair of cages (the inputs will be provided by complying with this assumption). You can also assume that the given cage names exist in the zoo.

Here is an example test program that uses this class and the corresponding output. We will use a similar program to test your solution so make sure that the name of the class is **ZooMap**, its interface is in the file called **ZooMap.h**, and the required functions are defined as shown above. Your implementation should use the **exact** format given in the example output to display the expected messages as the result of the defined functions. Please be aware that in order to display the probability, it is essential to employ precisely six digits following the decimal point.

Example test code:

```
#include "ZooMap.h"

int main() {

    ZooMap zm("cage.txt", "zoo.txt");

    zm.displayAllCages();
    cout << endl;

    zm.displayAdjacentCages("E");
    cout << endl;

    zm.displayAdjacentCages("C");
    cout << endl;

    zm.displayZooMap();
    cout << endl;

    zm.findSafestPath("E", "A");
    cout << endl;

    zm.findSafestPath("D", "C");
    cout << endl;

    zm.findMostDangerousPath("E", "A");
    cout << endl;

    zm.findMostDangerousPath("D", "C");
    cout << endl;

    zm.findMostDangerousPath("C", "F");

    return 0;
}
```

Output of the example test code:

```
6 cages and 9 connections have been loaded.

The cages in the zoo are:
A, B, C, D, E, F,
```

```
The cages adjacent to E are:
E -> D, F,

The cages adjacent to C are:
C ->

The whole zoo map is:
A -> E,
B -> A, C, E,
C ->
D -> A,
E -> D, F,
F -> B, C,

Safest path from E to A is:
E -> F -> B -> A
Probability: 0.066825

Safest path from D to C is:
D -> A -> E -> F -> C
Probability: 0.003402

Most dangerous path from E to A is:
E -> D -> A
Probability: 0.028350

Most dangerous path from D to C is:
D -> A -> E -> F -> B -> C
Probability: 0.001123

No path exists from C to F.
```

IMPORTANT NOTES:
Do not start your homework before reading these notes!!!

NOTES ABOUT IMPLEMENTATION:

1. You ARE NOT ALLOWED to modify the given parts of the header file. You MUST use the nonrecursive solution using a stack to implement the exhaustive depth-first search algorithm for finding the requested paths between the cages. You will get no points if you use any other algorithm for the solution of the search problem. You can use other data structures to help with your implementation but the main search algorithm must be implemented using a stack.
2. You MUST implement the stack and any additional container (e.g., list) by yourself. You ARE NOT ALLOWED to use the data structures and related functions in the C++ standard template library (STL) or any external library.
3. Moreover, you ARE NOT ALLOWED to use any global variables or any global functions.
4. Output message for each operation MUST match the format shown in the output of the example code.
5. Your code MUST NOT have any memory leaks. You will lose points if you have memory leaks in your program even though the outputs of the operations are correct. To detect memory leaks, you may want to use Valgrind which is available at `http://valgrind.org`.

NOTES ABOUT SUBMISSION:

1. In this assignment, you must have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your class. Your class name MUST BE `ZooMap` and your file names MUST BE `ZooMap.h` and `ZooMap.cpp`. Note that you may write additional class(es) in your solution.

2. The code (`main` function) given above is just an example. We will test your implementation using different scenarios, which will contain different function calls. Thus, do not test your implementation only by using this example code. We recommend you to write your own driver files to make extra tests. However, you MUST NOT submit these test codes (we will use our own test code). In other words, do not submit a file that contains a function called `main`.

3. You should put all of your `.h` and `.cpp` files into a folder and zip the folder (in this zip file, there should not be any file containing a `main` function). The name of this zip file should conform to the following name convention: secX-Firstname-Lastname-StudentID.zip where X is your section number. The submissions that do not obey these rules will not be graded. Please do not use Turkish letters in your file and folder names.

4. Make sure that each file that you submit (each and every file in the archive) contains your name, section, and student number at the top as comments.

5. You are free to write your programs in any environment (you may use Linux, Windows, MacOS, etc.). On the other hand, we will test your programs on "dijkstra.ug.bcc.bilkent.edu.tr" and we will expect your programs to compile and run on the dijkstra machine. If we could not get your program properly work on the dijkstra machine, you would lose a considerable amount of points. Thus, we recommend you to make sure that your program compiles and properly works on dijkstra.ug.bcc.bilkent.edu.tr before submitting your assignment.

6. This assignment is due by 23:59 on Friday, December 22, 2023. You should upload your work to Moodle before the deadline. No hardcopy submission is needed. The standard rules about late homework submissions apply. Please see the course home page for further discussion of the late homework policy.

7. We use an automated tool as well as manual inspection to check your submissions against plagiarism. Please see the course home page for further discussion of academic integrity and the honor code for programming courses in our department.

8. This homework will be graded by your TA **Saeed Karimi** (saeed.karimi@bilkent.edu.tr). Thus, you may ask your homework related questions directly to him. There will also be a forum on Moodle for questions.