Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Programming Language I • DIM0120
◁ **MAGOS**: **MA**ze **G**enerat**O**r and **S**olver, Programming Project ▷
November 18, 2018

Contents

1 Introduction

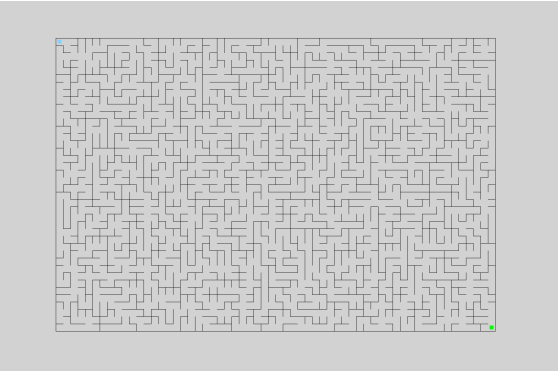
In this programming project your job is to develop a *perfect maze* **generator** and a **solver**. A perfect maze is a maze where each location in the maze is reachable from any other location. This also means that there is path between the maze's entrance and exit locations.

The **MAGOS** project should work in two steps. Firs it reads from input the required number of rows and columns of the maze and builds a perfect maze based on these information. In the second step, the program should find a solution (a path) connecting two cells, the maze's entrance and the exit.

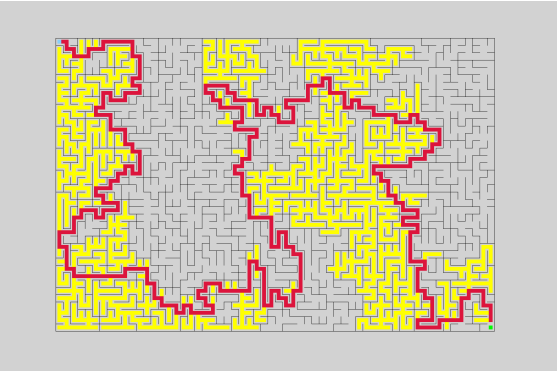
The program output should be written to a set of images depicting, step by step, both building and solving processes. The main challenge in this programming project is to identify a suitable maze representation so that it supports both building and solving processes, choose (and experiment with) an algorithm to build a maze (see for instance here), and design a algorithm to find the maze's solution.

To complete the task you have to master your skills in problem solving and system modeling, as well as figuring it out the best data structures to implement this project efficiently. You might need to use sequence containers (e.g. deque, stack, queues, vector, list), and an associative container (e.g. any type of dictionary).

The Figure 1a presents an example of one output image depicting the final step of the building process of a 40×60 maze, and the Figure 1b shows an output image presenting the corresponding solution (path in red) for that maze, connecting the top-left cell (the maze entrance) to the bottom-right cell (the maze exit).



(a) The last image of the building process of a 40×60 maze. All locations of the maze are connected.



(b) The last image of the solving process, showing the solution in red, and the failed attempts in yellow. The location of the maze that are not yellow or red are locations that have not been visited by the solution algorithm.

Figure 1: Building and solving a 40×60 maze.

2 Building the Maze

One strategy to build a perfect maze is to model a maze as a *matrix* of **cells**. Initially, all cells have four **walls**, that might be knocked down during the building process, so that **paths** among cells start to form. See Figure 2 for an example of a 40×60 maze at the beginning of the building process.

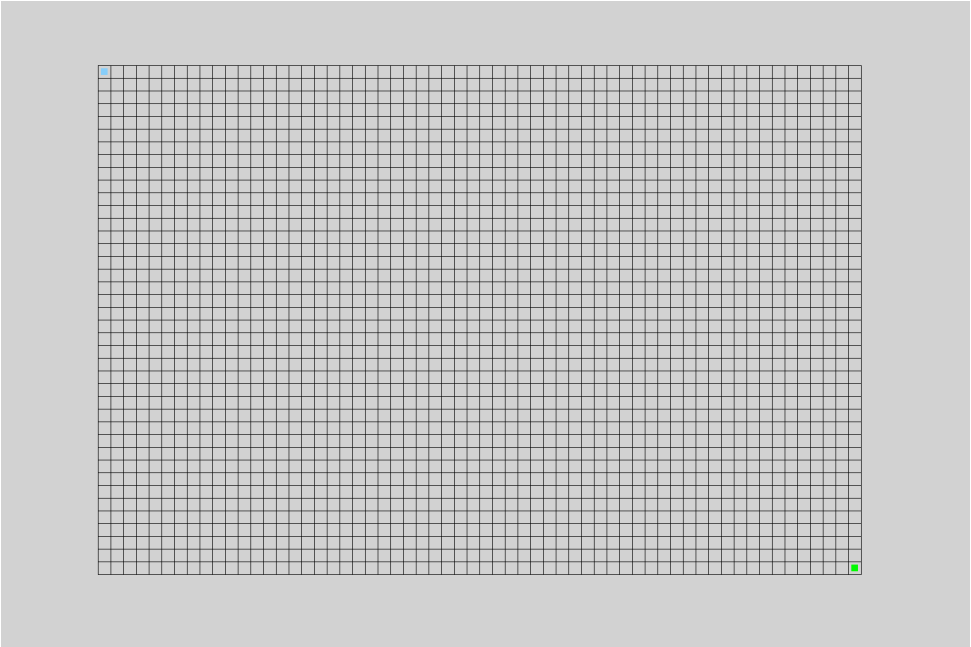


Figure 2: A 40×60 maze with the walls of all cells up. The maze entrance is the top-left cell, whereas the exit is the bottom-right cell.

An example of a simple algorithm that progressively creates paths among cells is:-

1. Choose a random cell to explore.
2. Choose a random wall of the selected cell.
3. Knock down the selected wall only if the neighbor cell that shares that wall does not have a path connecting them.
4. Repeat the previous steps until all cells are connected.

There are other algorithms that build mazes and is up to the team to choose the best approach to solve the problem.

3 Solving the Maze: pathfinding

To find a solution for the maze it is recommended to use an algorithm called **depth-first searching** (DFS) in conjunction with the **backtracking** approach to control and direct the search process.

This combined strategy might be described in high level as the following steps:

1. Store the entrance cell in a container.

2. While the container is not empty do:
   (a) Remove a cell from the container and check if it is the exit; if this is the case, the algorithm ends.
   (b) If the solution has not be found in the previous step, mark the current cell as (a candidate) part of the solution path. Find a way (a data structure?) to keep track of the positions that have already been tested so far to help the searching engine avoiding (testing) this position in the future.
   (c) Store in the container all the new possible single-step untested positions available (to be explored later).
   (d) If there is no untested cell in the previous step that you may go to from the current position it means we have reached a dead end; go back (backtrack) through all the cells that lead to the current position marking them as visited but not part of the solution until we get to a cell that has an untested neighbor that we can move to.

The one-million question is: "which container should I use?" Because we are relying on the backtracking approach, we may use a **stack** as the container. Therefore, each time we reach a dead end we must backtrack until the last cell where a decision has been made and try another (untested) direction.

4 Input

The input for the **MAGOS** is the rectangular maze dimensions (rows and columns, in that order), and the dimension of the output images (width and height, in that order). These arguments should be provided via command line arguments, as in

```
\$.\magos 40 60 1200 800
```

which creates a 40 rows by 60 columns maze that should be printed on an image with 1200 pixels in width by 800 pixels in height. Alternatively, you may omit part of the arguments and assume default values for them.

Of course, there are other input parameters such as the coordinates of the entrance and exit cells, the color of the drawing line, the color of the cells that are part of the solution, the color of the cells that have been visited but are no longer part of the solution (discarded cells), the color of the background, the color of the entrance and exit cells, the path where to output the files to, the names for the output files, etc.

All these parameters might be made fixed by the project, to make the input processing more simple. Alternatively, it is possible (although not required in this assignment) to create a configuration file in which all these parameters are defined. In that case, the program should receive as an input only the configuration file, since it should contain all the input data necessary to run the program.

By default, assume that the cell at the top most row, on the far left is the entrance; and the cell on the bottom most row, at the far right is the exit cell.

5 Output

Both the building and solving processes must be designed so that they might run in a step-wise fashion. This is a very important feature, since the output of the program is a series of PNG images depicting each step towards the solution in both processes. The ultimate goal here is to use an external program, for instance `ffmpeg`, to combine all the output images into two videos, one animating the building process, and another animating the chosen pathfinding algorithm. The resulting movies would enable the user to visualize and understand how both processes work.

Therefore, your program should output two sets of files, in two separate folders. The first folder should be named `builder`, while the second folder should be named `solver`. The files depicting the building process must all have the prefix '`building_`' followed by a sequential number, starting from 0 (zero)[1]. The sequential numbering should correspond to the step sequence taken during the building processes. The output of files to the `building` folder stops only when the building process is complete and the entire maze is ready. We consider a maze ready if there is a unique path between any two cells in the maze.

Similarly, the files depicting the solving process (running the maze) must all have the prefix '`solving_`', also followed by a sequential number, starting from 0 (zero). Each sequential file should correspond to the incremental advances done by the pathfinding algorithm, starting from the entrance and looking for the maze's exit. The program should stop only when the exit is reached by the algorithm. Therefore, the last output image should present the complete path found, from entrance to exit. Note that there shall always exist a solution, since the maze produced in the previous step (if done correctly) should have a unique path from any two cells, which includes the entrance and the exit.

6 The Implementation

In this section you will find some *suggestion* that might help the development process.[2]

6.1 Game loop design pattern

We suggest the Game Loop programming pattern. to organize the main loop in the program. In this pattern, there is a central class that should provide the following methods:

- constructor: class initialization and object instantiation.
- `initializer()`: sets ups all the objects parameters instantiated in the constructor. This method should validate the input arguments.
- `process_event()`: this method should process any input events, in case you plan to receive any input from the user (for instance, 'press enter to start the building process.').

[1] Don't forget to use zero padding so that all files have the same name length, as in 00001, 00002, 00003, etc.
[2] Most of the explanations, however, will be provided during the supporting classes.

- `update()`: This method should advance the current simulation (building or solving) a single step.
- `render()`: this is the method that sends the current snapshot of the maze to an output image file.
- `done()`: this method return true only if the simulation has finished. This might happen when the building and solving processes are completed or if an error happens during the execution.

6.2 Maze modeling

You may create a matrix of cells to represent the maze. The data we need to store in a cell are which walls are up, if a cell is part of the solution, if a cell was examined but is no longer part of the solution, and if a cell has not been visited yet. In other words, a cell may be either **visited** or **untested**. A visited cell, in turn, may be either **path** or **discarded** (no longer part of the path).

Instead of using a `struct` do model a cell, it is possible to store all these information in a `unsigned char` or a single byte. This can be accomplished if we encode the cell current status as a bit code, since most of the features of a cell are of the type on/off. For instance, the first bit on means that the top wall is up, the second bit would represent the right wall, the third bit the bottom wall, so on.

6.3 Maze building

To implement the building algorithm presented previously in Section 2, you may store each cell in a separate hash table of keys. A key is the unique numerical id of a cell, starting from zero, at the top left cell, and progressively advancing in a row-major order, until the bottom right cell.

Each cell belongs to a single hash table. Initially there are as many hash tables as there are cells. As the algorithm advances, a cell and a wall is picked up to be knocked down. As a result, a path is created connecting the chosen cell and its corresponding neighbor. Thus, the two corresponding hash tables must be united in a single hash table, meaning that the cells belong to the same set (there is a single path connecting them). This process is done by removing all the keys from the neighbor's hash table and inserting them into the chosen cell's hash table; the neighbor's hash table becomes empty and must be removed from the total list of hash tables.

This process is repeated until we only have a single hash table, with all keys inside it. This means there is a single path connecting any two cells in the maze. Consequently, the building process is completed.

7 Project Evaluation

You should hand in a complete program, without compiling errors, tested and fully documented. The assignment will be credit according to the following criteria:-

1. The program correctly reads and validates the input arguments (10 credits);

2. The program correctly builds a perfect maze, in which any two cells have a path connecting them (20 credits);
3. The program outputs a single image file for each and every step of the building algorithm (15 credits);
4. The program correctly solves the maze, showing the path from the entrance cell to the exit cell (20 credits);
5. The program outputs a single image file for each and every step of the solving algorithm, showing cells that are part of the path, cells that are no longer part of the path, and unvisited cells (15 credits);
6. The program is well organized in classes (subjective analysis) (20 credits);

The following items may *assign you work extra credits*, provided that you have completed all the regular items described before:-

- Support more than one algorithm to build a maze.
- Support more than one algorithm to solve the maze.

The following situations may *take credits out* of your assignment, if they happen during the evaluation process:-

- Compiling and/or run time errors (up to -20 credits)
- Missing code documentation in Doxygen style (up to -10 credits)
- Memory leak (up to -10 credits)
- Missing README file (up to -20 credits).

The README file (Markdown file format recommended here) should contain a brief description of the project, and how to run it. It also should describe possible errors, limitations, or issues found. Do not forget to include the author(s) name(s)!

Good Programming Practices

During the development process of your assignment, it is strongly recommend to use the following tools:-

- Doxygen: professional code documentation;
- Git: version control system;
- Valgrind: tracks memory leaks, among other features;
- gdb: debugging tool, and;
- Makefile: helps building and managing your programming projects.

Try to organize you code in several folders, such as `src` (for `.cpp` files), `include` (for header files `.h`, and `.inl`), `bin` (for `.o` and executable files) and `data` (for storing input files).

8 Authorship and Collaboration Policy

This is a pair assignment. However, you may choose to work alone. If you work as a pair, comment both members' names atop every code file, and try to balance evenly the workload. Only one of you should submit the program via Sigaa.

Any programming team may be called for an interview. The purpose of the interview is twofold: to confirm the authorship of the assignment and to identify the workload assign to each member. During the interview, any team member should be capable of explaining any piece of code, even if he or she has not written that particular piece of code. After the interview, the assignment's credits may be re-distributed to better reflect the true contribution of each team member.

The cooperation among students is strongly encouraged. It is accepted the open discussion of ideas or development strategies. Notice, however, that this type of interaction should not be understood as a free permission to copy and use somebody else's code. This is may be interpreted as plagiarism.

Any two (or more) programs deemed as plagiarism will automatically receive **zero** credits, regardless of the real authorship of the programs involved in the case. If your project uses a (small) piece of code from someone else's, please provide proper acknowledgment in the README file.

9 Work Submission

Only one team member should submit a single zip file containing the entire project. This should be done only via the proper link in the Sigaa's virtual class.

◀ The End ▶