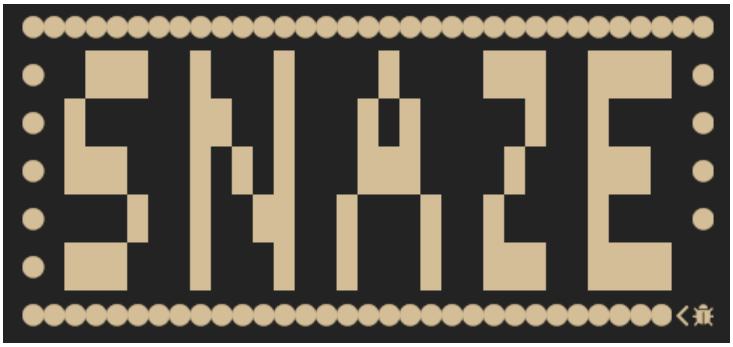
Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Programming Language I • DIM0120
◁ **Snaze**: A Snake Trapped in a Maze, Programming Project ▷
June 26, 2023



Contents

1 Introduction

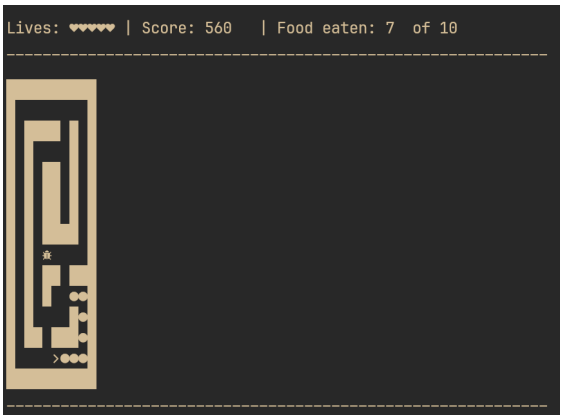
In this programming project your job is to develop a **simulation** of the classic Snake arcade video game with a twist: the snake is trapped inside a maze!

The **Snaze** game simulation loads the maze levels from an input text file, provided via command line arguments, and controls the snake movements. The main challenge in this programming project is to design a basic *artificial intelligence* (AI) engine that guides the snake to food pellets that pop out at random places inside the maze without running the snake into the maze's walls or itself.

To complete the task you have to master your skills in problem solving and system modelling, 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 the initial screen for the **Snaze** game, and Figure 1b shows a text-based representation of a game level being played, in which it is possible to identify the food pellet (an apple), and the snake moving Northwards.

(a) The opening screen of the **Snaze** game. The '*' represents the location where the snake spawns from.



(b) A screenshot of the **Snaze** in action, showing the snake, who has already eaten 3 apples, and the next apple she's trying to get to.

Figure 1: Text-based screens for the **Snaze** game.

2 The Gameplay

The game rules are:-

1. The snake has 5 lives (this may be changed via the command line interface).
2. The snake moves by extending its head 1 step in the direction it's moving and pulling its tail in.
3. If the snake eats a piece of food, its length grows by one and a new food pellet is randomly placed in a location the snake can get to.
4. There might be one or more levels, depending on the input file.
5. Everytime the snake crashes into itself or into a wall it looses one life; The snake **loses** the game simulation when all lives are spent.
6. The snake **wins** the game simulation if it eats all the food pellets in all levels.

3 The Problem

For each simulation iteration the program always do the following:-

1. Decides whether the snake keeps the *current direction* or turns either left or right to a *new direction*;
2. Move one step towards the direcion choosen in the previous step.

In order to control the snake the simulation must provide the decision required in Step 1. Thus, the decision-making process that happens in Step 1 is at the core of the game simulation.

In this programming project you should find **any sequence of directions** that leads the snake safely to the food pellet, if such a sequence exists. A valid solution will always make the snake reach the food pellet.

In case your AI engine does not find a valid solution, your simulation should fall back to a radomly controlled snake that, eventually, is going to crash. By randomly controlled snake we mean that your engine might suggest to the snake any direction that does not crash, unless there is no such a direction, in which case the AI engine just keeps the (doomed) current snake direction.

Your program should receive input information via the *command line interface* (CLI). The only essential information is the filename that contains the information about the game levels configuration. The output of your program should be the animation (i.e. a sequence of screens) of the snake following the moviments suggested by the game's AI engine.

4 Input

The input for the **Snaze** game simulation will always come from a file. The first line of the file will contain 2 non-zero positive integers: the number of rows and the number of columns of the level board "grid". If either of these values is zero or less, print an appropriate error message and end the program. These values will be separated by one or more white space characters. A level will always

be modeled as a rectangular grid. Assume that 100 is the limit for the number of rows or columns; thus, any input files with a board larger than 100×100 must be rejected.

The remaining lines of the file will contain the data representing the level. A level is composed by a set of characters that have the following meaning:-

- '#': represents a **maze wall**.
- '*': it is the snake's **spawn location**.
- '**.**': represents an **invisible wall**.

Each level must contain a single spawn location for the snake. If a level does not have one, your simulation should ignore that level. The invisible wall is useful to create non-rectangular levels. A location marked as 'invisible' cannot be reached by the snake nor it can receive a random food pellet.

Notice that an input file may have a sequence of one or more levels, each of them following the format described previously. Here we have and example of an input file with two levels. The first level of this example is show in Figure 1.

```
15 10
##########
#        #
# #### # #
# #    # #
# # ## # #
# # ## # #
# # ## # #
# # #### #
# #      #
# #*## ###
# # #    #
# #    # #
# ## ### #
#        #
##########
15 22
#####.....#####
#   #.....#   #
#   #########   #
#               #
#     ##### ######   #
#     ##### ##       #
#  *  ##### ## ###   #
#     ##### ## ###   #
#     #####    ###   #
#     ###########   #
####       #         #
....#      #    ###  #
.....#         ###   #
......#####    #     #
..........###########
```

5 Output

Your simulation should output a sequence of level representation (as in Figure 1), depicing the snake location inside the maze, as well as the walls and food pellet. This is done for each simulation iteration. Notice that the screen should also display the game status, i.e. the number of snake's lives, the score, the number of food pellets left to eat, and the current level being played.

Each time the snake crashes, your simulation should print a proper message and ask the user to hit enter to continue. If a level is clearead, the simulation should display a corresponding message and ask the user to hit enter to go the next level (if there is one).

In case the snake runs out of lives, the simulation should display a loosing message and quit the simulation. Similarly, if the snake clears all the levels, the simulation should display a winning message and also quit the simulation.

6 The Implementation

The next step towards a programming solution, after clearly understanding the problem, its input and output specifications, is to model the **Snaze** simulation entities.

Here is a list of classes that may be useful for modeling the problem:-

- `Snake` — represents the snake and its attributes. The game simulatiom must have only one snake.
- `Level` — represents a level of the game. It has dimensions and characters representing elements of the game. The game simulation may have one or more levels.
- `Player` — represents the AI engine. This class should store a sequence of direction to feed the simulation. The suggestion is to create at least two methods:
  - `bool find_solution()`: given the current snake location within the maze and the food pellet location, it returns `true` if it has found (ans stored) a sequence of directions leading to the food pellet, or `false` otherwise.
  - `Direction next_move()`: it returns a direction for the snake based on the sequence of directions found and stored by the AI engine or on a random policy (as described in Section 3).
- `SnakeGame` — the main entity that instantiates all other objects and manages the game execution. This class should provide methods, such as `initialize_game()`, `update()`, `process_events()`, `render()`, `game_over()`, that are called in the game loop. I recommend your organize your code architecture following the traditional Game Loop programming pattern.

6.1 Interface

The program may be called `snaze`, and should receive the input file through command line arguments. See below the snaze's CLI:

```
$ ./build/snaze
Usage: snaze [<options>] <input_level_file>
  Game simulation options:
    --help               Print this help text.
    --fps <num>          Number of frames (board) presented per second.
    --lives <num>        Number of lives the snake shall have. Default = 5.
    --food <num>         Number of food pellets for the entire simulation. Default = 10.
    --playertype <type>  Type of snake intelligence: random, backtracking. Default = backtracki
```

If you run the program without any arguments this (above) is what the program must print out.

6.2 The Backtracking

To produce a solution for the **Snaze** game simulation it is recommended to use the backtracking strategy, which might be described in high level as the following steps:

1. Starting from the initial snake position, store in a container all possible single-step locations the snake may occupy without crashing into a wall or into itself.
2. Remove a position from the container, check whether that position is the solution (i.e. the food pellet); if it's not the solution,
   (a) find a way (a data structure?) to keep track of the positions that have already been tested to help the AI engine avoiding (testing) this position in the future, and;
   (b) store in the container all the new possible untested positions the snake may reach departing from that location.
3. Keep doing this until the engine finds the food pellet or the container becomes empty, in which case it means there is no solution for the current snake-food configuration.

The one-million question is: "which container should I use?"

Don't forget to use some data structure to store *the sequence of directions* that lead to the location currently being tested in the Step 2. Remember, however, that whenever you reach a dead end, you must "undo" the directions leading to that dead end and resume the search towards another untested direction.

6.3 The bodyless snake: the Limitation of the Maze Solution for the Snaze Game Simulation

Consider the example below, in which we have a snake with body size $= 3$, i.e. a head and 2 body pieces. The current snake move direction is Northwards. The food pellet poped out at position $(4, 3)$, marked by a '`F`'. Assume that the entire maze is surround by invisble walls.

The (naïve) mouse-in-a-maze backtracking approach, in which we would only simulate the snake's head movement along the maze, would store in the container the positions $\langle(0,0),(0,1),(0,2),(1,2),(2,2)\rangle$, in this order, marking those positions as potential part of the solution. However, further down the backtracking algorithm inspects the position $(2, 1)$, which is currently occupied by the snake's tail.

According the the algorithm, the snake's reached a dead end. This means this path does not lead to the food pellet, therefore the whole path must be discarded and marked as visited but no part of the solution. Right?

```
    0   1   2   3
  +---+---+---+---+
0 |   |   |   |###|
  +---+---+---+---+
1 | V |###|   |###|
  +---+---+---+---+
2 | o | o |   |###|
  +---+---+---+---+
3 |###|   |###|###|
  +---+---+---+---+
4 |   |   |   | F |
  +---+---+---+---+

  V - Snake head going Northwards.
  o - Body of the Snake.
  F - The food pellet.
  # - Wall.
```

Wrong! In a real scenario, the snake body should come along with the head, unless our snake has been decapitated! In a correct simulation, the board should look like the board below (although, technically, the snake has not actually moved just yet, since we are inside the `find_solution` function call):

```
    0   1   2   3
  +---+---+---+---+
0 |   |   | o |###|
  +---+---+---+---+
1 |   |###| o |###|
  +---+---+---+---+
2 |   |   | > |###|
  +---+---+---+---+
3 |###|   |###|###|
  +---+---+---+---+
4 |   |   |   | F |
  +---+---+---+---+

  V - Snake head going Northwards.
  o - Body of the Snake.
  F - The food pellet.
  # - Wall.
```

Contrary to what had happened previously, in this scenario the snake has a clear path towards the food pellet! Therefore, the algorithms would additionally store in the container the positions $\langle(2,1),(3,1),(4,1),(4,2),(4,3)\rangle$, thus reaching the solution.

Considering the situation just descrined, the problems you need to solve are:

1. *How to keep track of the positions already visited by the snake?* In the second board, the algorithm should not consider going to position $(2, 0)$ since that would lead to a solution in which the snake would move in a circle endelessly. Besides, the position $(2, 0)$ has already been tested (i.e. occupied by the snake).
2. *How to make the snake's body come along with the head during the search for a solution?* The algorithm needs to make sure the "phantom" snake body does not block potential unexplored positions that might be part of the solution.

Any ideas? This is one of the big problems of this project you need to solve to get full score.

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. Correctly reads and validates a level input file (20 credits);
2. Considering the process of controlling the snake in a game simulation, we have 2 mutually excluding cases:
   (a) Only suggests random directions for the snake to follow (10 credits).
   (b) Correctly determines a set of directions (not necessarily the shortest path, although that would be highly desirable) that guides the snake to the food pellet; or make random suggestions in case there is no solution (50 credits);
3. Correctly display the various game states (running, level complete, game lost, game won, etc.) (30 credits);

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

- AI engine always suggests the **shortest set of directions** that lead to the food pellet.
- The program provides a nice graphical animation for the **Snaze** game simulation, which might be done with graphics library such as the SFML (`http://www.sfml-dev.org`).
- The simulation supports **more than one snake** running the maze simultaneously.

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 `author.md` file (up to -20 credits).

The `author.md` 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

Both members of the team should submit a single zip file containing the entire project or a link to the corresponding Git Hub Classroom generated repository. This should be done only via the proper link in the SIGAA's virtual class.

◀ The End ▶