Universidade Federal de Minas Gerais **Artificial Intelligence**
Instituto de Ciências Exatas **2014, 2nd semester**
Departamento de Ciência da Computação **Prof. Luiz Chaimowicz**

1st Assignment / Implementation

Introduction

Note: a small kit with input files and an output format verifier will be available soon.

In this list, we will work with a maze (Pac-Maze) to practice with the basics of some state space search algorithms (chapters 3 and 4 of textbook).

While in the original Pac-Man the goal is to collect all the pills and escape the ghosts that haunt you, in our Pac-Maze your goal is just to take the Pac-Man to single pill in the scenario - no ghosts to disrupt you. The setting is a two-dimensional world, represented by an matrix of characters. The pill is represented by `0` (zero), a wall by `#` and an empty space by `-`.

The coordinate system is based on the matrix representation in a program. The first coordinate is the row and the second is the column. The origin (0,0) is the top left character. In the example below, the world has 7 rows and 14 columns. The pill is in (1,6) – 2nd row, 7th column.

```
Origin (0,0) -> ##############
                #-----0--#---#
                #######-##-#-#
                #----------#-#
                #-##########-#
                #------------#
                ##############
```

Fig. 1 – Pac-Maze instance

Modeling

The state in Pac-Maze is simply the location of Pac-Man. For example, in Fig. 1 the goal state is (1,6).

An action in Pac-Maze corresponds to the 4 directions that Pac-Man can move: up, down, left and right. Assuming that Pac-Man starts in position (5,1) of Figure 1, the possible actions are right and up. By applying action 'right' in the initial state (5,1), state (5,2) is reached, as shown in Figure 2 (Pac-Man is represented as `P`):

```
##############
#-----0--#---#
#######-##-#-#
#----------#-#
#-##########-#
#-P----------#
##############
```

Fig. 2 – Status of Pac-Maze when action 'right' is executed from position (5,1) of Fig. 1

Input file

For the exercises, you must read Pac-Maze's world. The input file has the world size in its 1st line: `ma` rows and `n` columns, separated by a space. The next `m` lines contains `n` characters each (besides the line break). The characters are the wall, pill or empty space (according to the defined representation). For example, the contents of a file with the world of Fig. 1 is shown below.

```
7 14
##############
#-----0--#---#
#######-##-#-#
#----------#-#
#-##########-#
#------------#
##############
```

Fig. 3 – Input file for the example of Fig. 1

Exercise 1)

For a given Pac-Maze instance, function `successor(state)` receives a state and returns a list of pairs (action;reached state) for each action that can be performed in the state received as a parameter. The pair contains the action that can be performed in the state that was received as a parameter and the state achieved when performing that action. For example, for the state (5,1) in the world of Figure 1, `successor(5,1)` should return a list of two pairs (action;state) - note that action and states are separated by semicolons and state coordinates separated by commas:

`(right;5,2) (up;4,1)`

Implement the `sucessor` function.

Code evaluation:

- Write a script `avalia_sucessor.sh` that runs your code. Script `avalia_sucessor.sh` should receive the path of the input file with the description of the world and the coordinates of a state and call your program. Your code should print the pairs (action;state) on the screen for each state reached from the given state. Different pairs (action;state) must be separated by a space. Assuming that the file described in Figure 3 is `pacmaze.txt`, the execution of your script to query the successors of (5,1) and the expected output are as follows (the order of the pairs is not important):

```
avalia_sucessor.sh pacmaze.txt 5 1
(right;5,2) (up;4,1)
```

Exercise 2)

From the `successor` function, you will be able to build function `expand` and the search graph from an initial state. In the search graph, each node contains information about the state to which it refers, as well as information that will help searching for a solution.

Arcs of the search graph are the actions and the attributes of each node are:

- State: representation of the state that this node refers to (e.g.: (5,1))
- Parent: reference to the parent node
- Action: action applied to parent node to generate this one
- Path cost: cost of the path from initial state until this node. In Pac-Maze, each action (arc of the graph) has cost 1. Thus, cost of a node is cost of parent + 1.
- (optional) references to child nodes.

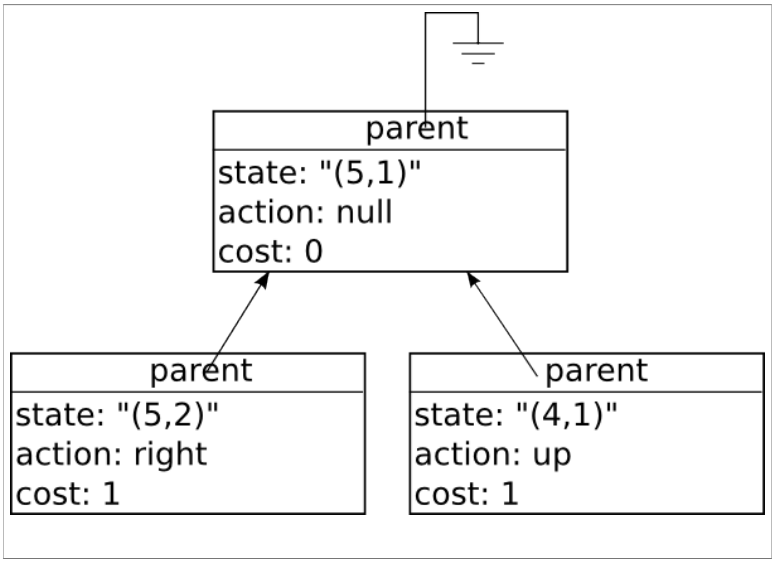

Fig. 4 – Pac-Maze node representation. Assuming that initial state of Fig. 1 is (5,1).

Function `expand` receives a node and returns a set of its child nodes, using `successor` function. For example, when receiving the root node of Fig. 3, the two inferior nodes are returned.

Implement the search graph node and the function `expand`.

Code evaluation:

- Write a script `avalia_expande.sh` that runs your code. The script `avalia_expande.sh` will receive the file with the world description, a state for query and its cost on the command line. Your script should call your program, which should create a node with this information, call function `expand` and print the desired result on the screen. Your code should print on the screen the quadruple (action;state;cost;parent.state) for each node of the set returned by the expansion. Different quadruples must be separated by a space. Assuming that the file described in Figure 3 is `pacmaze.txt`, the execution of your script for state (5,1) with cost 3 and the expected output are as follows (the order the quadruples is not important):

```
avalia_expande.sh pacmaze.txt 5 1 3
(right;5,2;4;5,1) (up;4,1;4;5,1)
```

(note that: items of a quadruple are separated by semicolons, states are row and column separated by a comma, different quadruples are separated by space, the states of the parents of all expanded nodes are the same and the cost of the child is the cost of parent + 1)

Exercise 3)

With function `expand`, you'll be able to implement the graph search procedure. In this procedure, a search space node belongs to set `explored` / closed or `fringe` / frontier / open. The `explored` set stores states that have already been expanded and the abstract data type for the `fringe` stores the ones that can be expanded next. The way the `fringe` is implemented leads to the different search algorithms (BFS, DFS, A *, etc.). Note the pseudo-code for general graph search (adapted from textbook):

```
function graph_search(fringe): //initially empty fringe
    explored ← {}
    fringe.insert(new Node(initial state))

    repeat:
        if fringe.empty():
            return "No solution"
        node ← fringe.remove()
        if (node.state is the goal):
            return solution(node)

        if node ∉ explored:
            add node to explored
            fringe.insert_list(expand(node))
```

Note that the node with the initial state has null parent and action, since it is the first. The progress of the search depends on how the fringe grows, which depends on which node is expanded. Besides, the 'explored' set prevents nodes from being expanded more than once (which would mean revisiting a state).

The `fringe` abstract data type must provide the following operations:

- `insert(Node n)`: adds node n to the fringe.
- `insert_list(lista_de_nodes)`: adds all nodes on the list to the fringe
- `remove()`: removes a node from the fringe and returns it

Function `solution` receives a node and returns the sequence of actions and states that lead from the initial state to the received node, that is, how the problem is solved.

Your task:

- Implement breadth-first search (BFS). To do this, implement `fringe` as a queue.
- Implement depth-first search (DFS). To do this, implement `fringe` as a stack.
- Implement A* algorithm. To do this, function `remove` of the `fronteira` should return the node with the lowest cost `f(node) = g(node) + h(node)`. In this cost function, `g(node)` is `node.cost` and `h(node)` is the heuristic distance from node to the goal. You choose the heuristic distance function.

Code evaluation:

- Write the scripts avalia_bfs.sh, avalia_dfs.sh, avalia_astar.sh that runs the code of each search algorithm. The scripts will receive the input file with the world description and the starting point coordinates and call your program, which will run the search algorithm from the node with the starting location of Pac-Man and print the result in screen. Your code should print on the screen sequence of actions leading from the initial state to the goal state (the location of the pill). Actions must be separated by a space. For example, assuming that the file described in Figure 3 is pacmaze.txt, and the starting point is (5,1), the execution of one of these scripts and the expected output are as follows:

```
avalia_bfs.sh pacmaze.txt 5 1
up up right right right right right right up up left
```

Exercise 4

Besides your programs, present the following items in a `.pdf` file:

- The solutions your search algorithms found for the input files of `kit_ex1.zip` (available soon), **except for DFS**.
- Quantitative analysis, comparing the search algorithms on the number of expanded states, solution cost and execution time. **Present comparative table and/or charts.**
- **Answer and explain:**
  - Is your A* heuristic admissible? Is it consistent?
  - Do breadth-first search and/or depth-first search always present optimal solutions for Pac-Maze?

Wrapping up, you must submit a `.zip` containing:

- A script `avalia_sucessor.sh`, according to Exercise 1;
- A script `avalia_expande.sh`, according to Exercise 2;
- The scripts `avalia_bfs.sh, avalia_dfs.sh, avalia_astar.sh`;
- A `.pdf` file with the answers of Exercise 4;
- All files of your source code.

WARNING: the shell scripts (`.sh`) and the `.pdf` file must be located at the root of your `.zip` file! That is, the content of your `.zip` must be:

```
compila.sh   (if you are using a compiled language)
avalia_sucessor.sh
avalia_expande.sh
avalia_bfs.sh
avalia_dfs.sh
avalia_astar.sh
respostas.pdf
[source code files] <<you may create subdirectories if needed
```

Content of the `.zip` file to be submitted.

Final remarks

- For A*, heuristics guarantee that fewer search space nodes will be expanded, compared to BFS e DFS. Even so, the way the boundary is implemented directly impacts the performance (execution time) of A*. A naive implementation, with a linear list, may take much more than BFS or DFS. Therefore it is advisable to use a smarter data structure.
- You can implement your code in your favorite programming language.
- As implementações podem ser feitas na sua linguagem de programação favorita. However, make sure that your code works in a GNU/Linux machine (e.g. `tigre.dcc.ufmg.br`). Do not print anything beyond requested, as this will result in error during the correction. For each exercise, in addition to the program, you must write a simple shell script (`.sh`) to run your program with the given inputs. If you use a compiled programming language, write a `compila.sh` script that compiles all your code. Instructions on writing the shell scripts can be found in the `auxiliar_shellscript.pdf` file.
- If you do not use GNU/Linux, be careful with character encoding, especially with line breaks. It's really important to test your code in a GNU/Linux environment, because it will be checked there. If you do not have access to the GNU/Linux computers of DCC, use a virtual machine to test your program. A list of virtual machines with preinstalled Ubuntu is available at: http://www.osboxes.org/ubuntu/.