U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

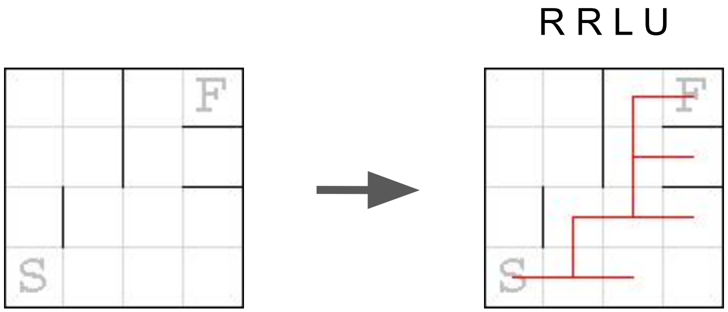
Robot Mazes

Artificial Intelligence 2021/2022

Group 20_1D
Adriano Soares, up201904873@fe.up.pt
Filipe Campos, up201905609@fe.up.pt
Francisco Cerqueira, up201905337@fe.up.pt

Specification of the Game

The problem consists in programming a robot with a minimum list of unitary movement commands that, when looped, lead it from Start to Finish. The board is composed with walls and if the robot bumps into one it cannot move. Here’s an example:



Problem Specification

State representation: list of instructions ∈ {‘U’,’D’,’L’,’R’,’E’}.

Initial state: [ ] (empty list).

Objective test: Loop through the list of instructions until it either reaches the final state or a cycle is found.

Operators:

Name	Preconditions	Effects	Cost
Left	The last element of the list mustn’t be the End operator.	Adds left operator to the end of the state list.	1
Right		Adds right operator to the end of the state list.	1
Up		Adds up operator to the end of the state list.	1
Down		Adds down operator to the end of the state list.	1
End	The current state must not contain cycles.	Adds end operator to the end of the state list.	0

Heuristics

1. Prioritize states in which the list of operators contains at least one of the directions that are fundamental to reach the final state. For instance, in a square matrix, if the robot is in the bottom left corner and the final state is in the top right corner, it should avoid visiting states that do not contain a single UP or RIGHT instructions. If the state does not contain one of these directions, it adds cost one to the heuristic for each direction needed.
2. Pre-compute the path with the lower number of direction changes between start and finish positions, obtain the direction changes made, group patterns with most elements and sum their length with the number of ungrouped direction changes. The heuristic value will be the absolute value of the difference between the current number of commands and the value computed previously.

Work Implemented

The project is being developed in Python3 with Visual Studio Code, using Pygame as the main graphical interface. We'll use a matrix based adjacency list to represent the maze and the obstacles, a list to represent the state and a search tree to keep track of the states for the search algorithms. The project repository contains the following appropriately named main directories: assets, doc and src.

```python
class RobotMazeState(State):
    def __init__(self, instructions): # {'U','D','L','R'}
        self.instructions = instructions

    def get_instructions(self):
        return self.instructions

    def is_simulation_state(self):
        return len(self.instructions) > 0 and self.instructions[-1] == 'E'

    def has_cycle(instructions):
        n = len(instructions) // 2
        return len(instructions) % 2 == 0 and all(instructions[i] == instructions[i+n] for i in range(n))

    def __str__(self):
        return '[' + ', '.join(i for i in self.get_instructions()) + ']'

    def __eq__(self, other):
        if isinstance(other, RobotMazeState):
            return self.get_instructions() == other.get_instructions()
        return False

    def __len__(self):
        return len(self.instructions)
```

```python
class SearchProblemSolver:
    def __init__(self, initial_state):
        self.initial_state = initial_state

    # Methods to be overwritten
    def cost(self, state):
        raise NotImplementedError()

    def heuristic(self, state):
        raise NotImplementedError()

    def operators(self, state):
        raise NotImplementedError()

    def is_final_state(self, state):
        raise NotImplementedError()
    ################################

    def breath_first_search(self, max_depth):
        return self.search_algorithm(max_depth, Queue(), False, False)

    def depth_first_search(self, max_depth):
        return self.search_algorithm(max_depth, LifoQueue(), False, False)

    def iterative_deepening_search(self, max_iterations):
        total_visited_nodes = 0
        for i in range(1, max_iterations + 1):
            (path, visited_nodes) = self.depth_first_search(i)
            total_visited_nodes += visited_nodes
            if len(path) > 0:
                break
        return (path, total_visited_nodes)

    def uniform_cost_search(self, max_depth):
        return self.search_algorithm(max_depth, PriorityQueue(), True, False)

    def greedy_search(self, max_depth):
        return self.search_algorithm(max_depth, PriorityQueue(), False, True)

    def A_star_search(self, max_depth):
        return self.search_algorithm(max_depth, PriorityQueue(), True, True)

    def search_algorithm(self, max_depth, queue, has_cost, has_heuristic):
        queue.put(StateWrapper(self.initial_state, 0, None))

        visited_nodes = 0
        while queue.qsize() > 0:
            state_wrapper = queue.get()

            visited_nodes += 1
            depth = state_wrapper.depth
            depth += 1
            if depth > max_depth:
                continue

            next_states = self.operators(state_wrapper.state)
            filter(lambda state: state != state_wrapper.parent, next_states) # Exludes the previous state

            for next_state in next_states:
                next_state_wrapper = StateWrapper(next_state, depth, state_wrapper)

                # Update total cost
                cost = self.cost(next_state)
                next_state_wrapper.total_cost = state_wrapper.total_cost + (cost if has_cost or has_heuristic else 1)

                # Update priority
                next_state_wrapper.priority = cost if has_cost else 1
                next_state_wrapper.priority += self.heuristic(next_state) if has_heuristic else 0

                # Check for final state
                if self.is_final_state(next_state):
                    print(next_state_wrapper.total_cost)
                    return (SearchProblemSolver.get_path(next_state_wrapper), visited_nodes)
                queue.put(next_state_wrapper)
        return ([], visited_nodes)
```

References

[1] Problem Definition, https://erich-friedman.github.io/puzzle/robot/

[2] Cycle In Maze (similar problem), https://codeforces.com/problemset/problem/769/C

[3] Adjacency List Temporal and Spatial analysis, https://en.wikipedia.org/wiki/Adjacency_list