IA - First Practical Assignment

Group 14_1C
Mário Travassos - up201905871@edu.fe.up.pt
Rita Mendes - up201907877@edu.fe.up.pt
Tiago Rodrigues - up201907021@edu.fe.up.pt

April 24, 2022

Introduction

The assigned project consists of:

- The development a solitaire game, called Exactly One Mazes
- The development of several algorithms that can solve the game at different levels of speed and effectiveness, using different heuristics and search methods as basis of work.

To accompany this, the algorithms will be analysed in terms of:

- Speed of execution
- Number of nodes visited

A graphical user interface built with pygame, mostly keyboard based, for the game will be developed as well. The player can navigate this interface via use of the arrow keys. The return key will be used to select an option. Backspace will allow the user to navigate to the previous state.

Problem Definition

Details of the puzzle are as follows, according to [Fri]:

- The board has several L shapes placed inside it
- The initial position is on the lower left corner of the board
- The final position is positioned on the top right corner of the board.
- The objective of the puzzle is to find a path from the initial position to the final position, which goes through every single L shape, albeit only once per shape
- At each state, it is only possible to move vertically or horizontally to a cell adjacent to the current one

Although the puzzle does not demand this, the solution obtained by the player (and our solving algorithms) will ideally also be the shortest possible path to the solution, or paths, provided that more than one exists.

The Problem as a Search Problem

According to [Reia], we can define the problem as a search problem the following way:

- A set of states, S. In this case, our set of states are all the possible board states. Each board can be represented as a matrix (list of lists), where a 0 indicates free space, a positive integer i represents a tile belonging to the L shape L_i, and a -1 represents a visited tile. An auxiliary data structure can be a stack of moves, to which items are added (or popped from) as the program runs.
- An Initial state $s \in S$. In this case, it is the blank board, without any -1 tiles. A representation of it may be found in the source code
- A Transition Relation T. In our representation of this problem, (x, y) represents the position of the player - (0, 0) is the starting position, and (n, n) the final position. The set of possible transitions is as follows:
  - Name: MoveUp. Pre-Conditions: $y < n,\ visited(x, y+1) = False$. Post-Conditions: $(x, y) = (x, y+1)$. Cost: 1.
  - Name: MoveDown. Pre-Conditions: $y > 0, visited(x, y-1) = False$. Post-Conditions: $(x, y) = (x, y-1)$. Cost: 1.
  - Name: MoveLeft. Pre-Conditions: $x > 0, visited(x-1, y) = False$. Post-Conditions: $(x, y) = (x-1, y)$. Cost:3 1.
  - Name: MoveRight. Pre-Conditions $x < n, visited(x+1, y) = False$. Post-Conditions: $(x, y) = (x+1, y)$. Cost: 1.
- A Set of Final, Objective states $O \in S$. In this case, it is the set of boards with a path from the lower left corner to the top right one, passing through each L shape exactly once. Also, as a way to check if every L shape has been passed at least once, the auxiliary set should include every integer.

Implemented Features

The language we chose for our assignment was Python, due to its clean syntax, simplicity in writing algorithms, and extensive collection of libraries we might want to use to aid us in developing our project. So far, we have implemented:

- Board class, with auxiliary functions and data structures to:
  - Save state
  - Print current state
  - Visit/Unvisit board cells
  - Track number of shapes visited
  - Change state
- Backtracking algorithm
- Began work on a generic node abstraction of the Board Class, to allow us to use a board as a generic node in any algorithm.

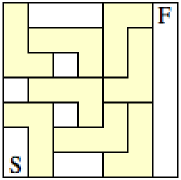
To this end, we used a few data structures:

- List to represent the puzzle board
- List to represent the visited tiles
- Set which holds the visited shapes at each state

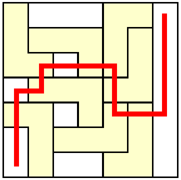
Although we are not using these in any algorithm yet, we intend to test a few heuristics, among which the Manhattan and Euclidian distances from the starting cell to the final cell, and the number of shapes visited.

Approach Taken

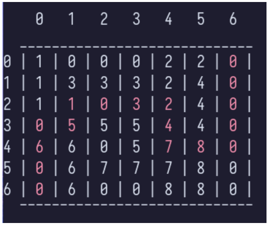
- The interaction between the different algorithms and the Board was made using a BoardNode abstraction layer. The algorithms only traverse a tree of board nodes, and each state is generated as the children of the previous one. With a human player, a similar approach was taken, making use of the BoardNode class.
- The following heuristics were used to evaluate the board. Note that the objective is to minimize the value of each:
  - Euclidean distance, as a bad example of a heuristic
  - Manhattan distance, as a better but still not ideal example
  - Number of L's yet to visit
- Pygame was used to develop the GUI and give a more pleasing experience to the user



Images depicting the original board



Solution to the board



```
    0   1   2   3   4   5   6

  -----------------------------
0 | 1 | 0 | 0 | 0 | 2 | 2 | 0 |
1 | 1 | 3 | 3 | 3 | 2 | 4 | 0 |
2 | 1 | 1 | 0 | 3 | 2 | 4 | 0 |
3 | 0 | 5 | 5 | 5 | 4 | 4 | 0 |
4 | 6 | 6 | 0 | 5 | 7 | 8 | 0 |
5 | 0 | 6 | 7 | 7 | 7 | 8 | 0 |
6 | 0 | 6 | 0 | 0 | 8 | 8 | 0 |
  -----------------------------
```

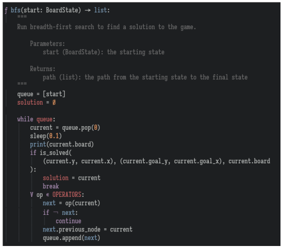
CLI representation of the solved Board

Implemented Algorithms

We implemented the almost all the algorithms that appeared in [Reib], with the notable exception of Bidirectional Search. These include:

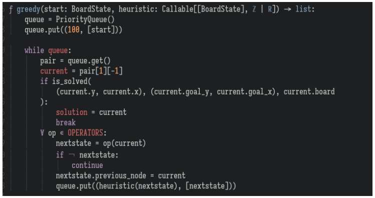
- Blind Search Algorithms like Depth-First Search, Breadth-First Search, Iterative Deepening and Uniform Cost Search
- As well as the two Heuristic Methods, Greedy Search and A* Search.

All of these methods work as intended, and use only the BoardNode as a way to interact with the boards. Also, they are responsible for printing the progress of the board on each instance.



```
ƒ bfs(start: BoardState) → list:
    """
    Run breadth-first search to find a solution to the game.

        Parameters:
            start (BoardState): the starting state

        Returns:
            path (list): the path from the starting state to the final state
    """
    queue = [start]
    solution = ∅

    while queue:
        current = queue.pop(0)
        sleep(0.1)
        print(current.board)
        if is_solved(
            (current.y, current.x), (current.goal_y, current.goal_x), current.board
        ):
            solution = current
            break
        ∀ op ∊ OPERATORS:
            next = op(current)
            if ¬ next:
                continue
            next.previous_node = current
            queue.append(next)
```

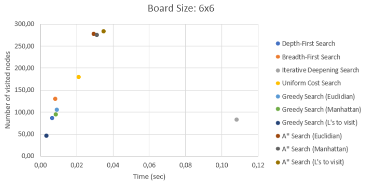
Breadth First Search Implementation



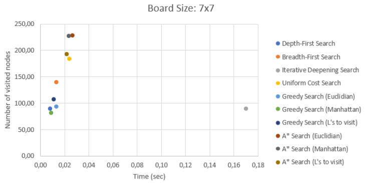
```
ƒ greedy(start: BoardState, heuristic: Callable[[BoardState], ℤ | ℝ]) → list:
    queue = PriorityQueue()
    queue.put((100, [start]))

    while queue:
        pair = queue.get()
        current = pair[1][-1]
        if is_solved(
            (current.y, current.x), (current.goal_y, current.goal_x), current.board
        ):
            solution = current
            break
        ∀ op ∊ OPERATORS:
            nextstate = op(current)
            if ¬ nextstate:
                continue
            nextstate.previous_node = current
            queue.put((heuristic(nextstate), [nextstate]))
```

Greedy Search Implementation

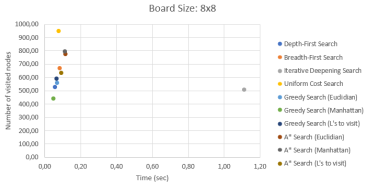
Results



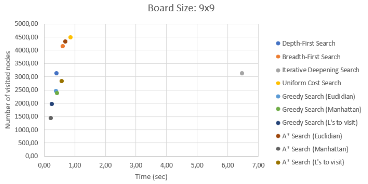
Average results with an easy board



Average results with a medium board



Average results with a hard board



Average results with an "extreme" board

Final Remarks

From this project it is possible to conclude that:

- Choosing the correct heuristic is key for the performance of informed searches. Minimizing the number of L's visited and Euclidean Search were the most consistent ones
- Greedy search proved to be more efficient than the A* algorithm, due to ignoring the cost to get to a certain point
- Iterative deepening, which in this situation should be a good option, turns out to be painfully slow
- Blind methods don't differ so much from informed searches as expected (in some cases they are even faster)
- Sometimes Depth-First Search gives out amazing results (solving with 22 moves in one situation), but this is heavily dependant on the branch it takes, and can be extremely slow on others

References

[Fri] Erich Friedman. *Exactly One Mazes*. URL: https://erich-friedman.github.io/puzzle/exactly1/.

[Reia] Luís Paulo Reis. *Lecture 2a: Search Problems*. URL: https://moodle.up.pt/pluginfile.php/196454/mod_resource/content/0/IART_Lecture2a_Search.pdf.

[Reib] Luís Paulo Reis. *Lecture 2b: Solving Search Problems*. URL: https://moodle.up.pt/pluginfile.php/196455/mod_resource/content/0/IART_Lecture2b_SolvingSearch.pdf.