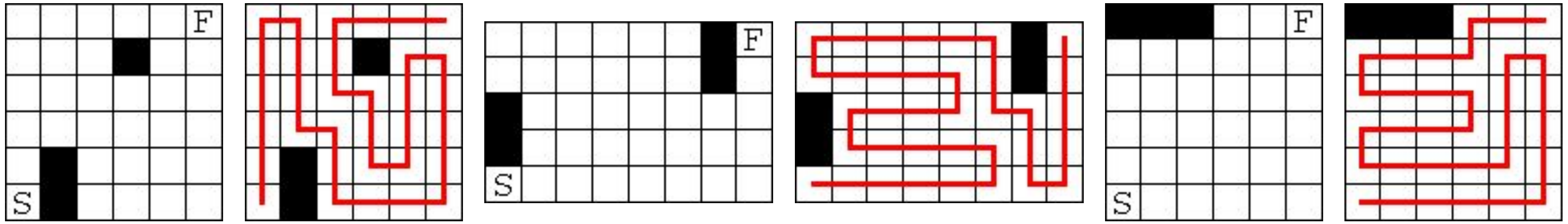
IART - 1st Assignment

Heuristic Search Methods for One Player Solitaire Games

Group 03_1A
Afonso Monteiro - up201907284@up.pt
Miguel Lopes - up201704590@up.pt
Vasco Alves - up201808031@up.pt

Work Specification

This assignment consists in the development of a program that will use heuristic search methods for a one player solitaire game, in our case the Unequal Length Mazes game. It consists of finding a path from the bottom left to the top right passing through every white square exactly once. The path must alternate between horizontal and vertical segments, and two consecutive segments can not be the same length. Each puzzle has a unique solution.



Figures 1-6: Examples of puzzles and their unique solution.

Problem Formulation

State Representation:

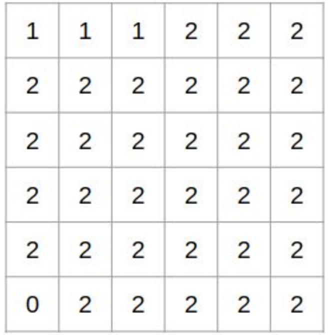
The board is represented through a Matrix with nxm dimension, where M[ni][mi]=k and k can be:

- **1** if there is a wall or if that square has been visited already
- **2** if that square can be visited (hasn't been visited yet)
- **0** represents the current position of the player

Additionally each game state holds the following variables:

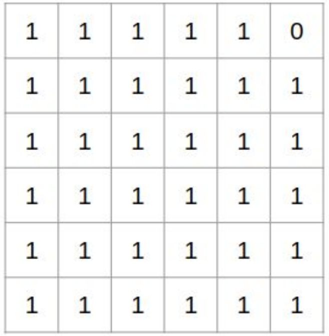
- **moveCount**, a variable that indicates the number of moves made so far in the current direction.
- **previousMoveCount**, a variable that indicates the number of moves made in the previous direction and that limits the amount of moves that can be made in the current direction.
- **previousDirection**, a variable that indicates the previous direction (horizontal or vertical)
- **P=(x,y)**, a tuple with the position of the player to improve efficiency

Initial State



1	1	1	2	2	2
2	2	2	2	2	2
2	2	2	2	2	2
2	2	2	2	2	2
2	2	2	2	2	2
0	2	2	2	2	2

Final State (Solution)



1	1	1	1	1	0
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1

Problem Formulation

Operators:

- MU(state), move up
- MD(state), move down
- MR(state), move right
- ML(state), move left

Evaluation Functions and Heuristics:

- Number of squares left to finish
- The Manhattan distance from the current square to the objective square

H = ManhattanDist(state, objective) + Count(not_visited) in state

Operator	Precondition	Effect
MU(state)	0 < y <= n - 1 and state.board[y-1][x] != 1 and ((state.previousDirection!=Up and state.moveCount!=state.previousMoveCount) or (state.previousDirection==Up))	newState.board[y-1][x]=0 and newState.board[y][x]=1 and P=(x,y-1) and newState.previousDirection=Up and (newState.moveCount+=1 or (newState.moveCount=1 and newState.previousMoveCount=state.moveCount))
MD(state)	0 <= y < n - 1 and state.board[y+1][x] != 1 and ((state.previousDirection!=Down and state.moveCount!=state.previousMoveCount) or (state.previousDirection==Down))	newState.board[y-1][x]=0 and newState.board[y][x]=1 and P=(x,y-1) and newState.previousDirection=Up and (newState.moveCount+=1 or (newState.moveCount=1 and newState.previousMoveCount=state.moveCount))
ML(state)	0 < x <= m - 1 and state.board[y][x-1] != 1 and ((state.previousDirection!=Left and state.moveCount!=state.previousMoveCount) or (state.previousDirection==Left))	newState.board[y][x-1]=0 and newState.board[y][x]=1 and P=(x-1, y) and newState.previousDirection=Left and (newState.moveCount+=1 or (newState.moveCount=1 and newState.previousMoveCount=state.moveCount))
MR(state)	0 <= x < m - 1 and state.board[y][x+1] != 1 and ((state.previousDirection!=Right and state.moveCount!=state.previousMoveCount) or (state.previousDirection==Right))	newState.board[y][x+1]=0 and newState.board[y][x]=1 and P=(x+1, y) and newState.previousDirection=Right and (newState.moveCount+=1 or (newState.moveCount=1 and newState.previousMoveCount=state.moveCount

Implementation

The chosen language to codify is Python

The data structures used include:

- Nodes that hold the states, connected in a tree-like structure
- A state class that has the board, the location of the player and the information about the previous move
- The board is represented using a list of lists
- The location is represented using a tuple of coordinates
- The previous move is stored using a string

What has been implemented:

- A State class
- A Node class (whose nodes will be consisted of State objects)
- The objective function
- A player mode
- A algorithm solving mode that uses the implemented algorithms (see the next slide)

Implemented algorithms

We decided to implement the following algorithms:

- BFS (Breadth-first search);
- DFS (Depth-first search);
- Greedy based algorithm.

The purpose behind this decision centers around the fact that the movements are all uniform from a cost perspective, which means rendering algorithms such as the Uniform Cost Algorithm and the A-Star Algorithm (both cost-dependent) are, respectively, equal to the BFS and Greedy algorithms.

Result analysis

With a problem this complex, it is natural that the non-heuristic based algorithms (BFS and DFS) would take such a long time to reach the goal of this maze, having to tunnel through numerous state nodes.
On the other hand, the Greedy algorithm is just as efficient as its heuristic function. We tested three heuristic functions: minimizing the Manhattan distance, minimizing the number of spaces left to go through and the minimum of the sum of both the Manhattan distance and the number of spaces left to go through. We concluded that this last option, Greedy with min(MD + NFSL) heuristic, obtains clearly better results, and so we ran that algorithm on multiple boards in order to analyse its efficiency.

Algorithm	Time (6x6 board)	Worst-case space complexity
BFS	556.216	O(|V|)
DFS	901.142	O(|V|)
Greedy - Manhattan Distance (MD)	236.411	O(|V|)
Greedy - Number of Free Spaces Left (NFSL)	20.013	O(|V|)
Greedy - MD + NFSL	0.690	O(|V|)

Greedy Algorithm Analysis

Further comparison of the greedy algorithms with just one of the heuristics defined makes us believe that the best individual heuristic function is not the manhattan distance, but the one that minimizes the number of spaces left.

Board Format	Board Size	Empty Count	Time
6x6	36	33	0.284118079000109
		33	2.1037659270000404
		33	4.947820024000066
		33	3.7441650349999236
		35	1060.4494458370002
		33	0.3491299780002919
5x8	40	36	7.2125845750001645
		36	0.09519288699993922
		36	6.092445683999813
		36	67.91092045300002
7x7	49	45	669.4189071149999
		45	151.57660297199982
		45	10371.732638866

Conclusions

In this project we were able to grasp the concepts of heuristic search methods for one player solitaire games.

Making use of various algorithms, we experienced the advantages and disadvantages of them, and we come to the conclusion that having heuristics improves the efficiency by a wide margin.

This knowledge is the first step in the study of intelligent agents and one of many on our formation as engineers.