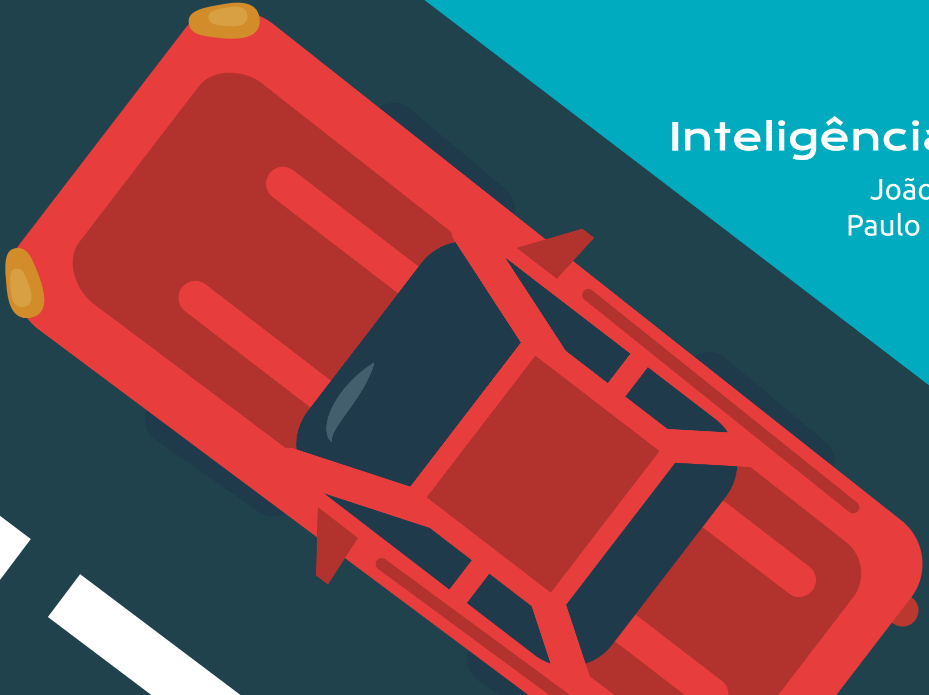

Agent for Rush Hour Game

Inteligência Artificial 2022/2023

João Monteiro - (102690) - joao.mont@ua.pt
Paulo Pinto - (103234) - paulojnpinto02@ua.pt

universidade de aveiro
theoria poiesis praxis

Introduction

In this report we will be explaining how the agent we developed to solve the Rush Hour game works. For this, we used the student.py archive given by the teachers and also replicated the player.py archive to start developing our agent algorithm.

Game Problem and Classes

This game is divided into levels pre-created by the teacher in the levels.txt archive. It is given as a string and follows the structure "a grid n" being 'a' the current level number, 'grid' a sequence of characters with the car's letters, empty spaces represented by 'o' and immovable obstacles represented by 'x', and, finally, ‘n’ represents the number of possible states for each different grid. The empty spaces and immovable obstacles only occupy one cell in the given grid, and the cars can occupy between two and three cells and can also be horizontal or vertical cars.

Our agent class is called Bot and initializes itself with the grid and cursor state for each of the levels to start solving the given problem.

Algorithm

The Bot class also contains the run function, that is called in student.py with three given arguments. The grid state, cursor state, and selected state.

Using these three elements, the first thing that our algorithm does is pass the initial grid state and grid size to the function make_board. This function will take the given grid state and split it to get the sequence of cars in the grid, and then, it will separate into a list with a number of elements equal to the size of the grid, that contains a list with the cars, empty spaces and obstacles of each row of the grid. For example, if the grid is 6x6, it has 36 elements, so, our function separates them into rows of 6 elements.

Having this grid, then, we separate the cars according to their orientation. Vertical cars will assume a letter in lower case, and horizontal cars will be upper case. Empty spaces remain with the letter ‘o’ and obstacles with ‘x’. As we found later that the letter ‘O’ can be associated with a vertical car, we decided to transform it into a ‘z’, and then, when analyzing the movement of the car to move, we back the letter to ‘O’. After we threat the board, the search function is called.

Search Function

In this function, first we initialize one queue with the board and a set with all possible boards we had (empty at the beginning of the search). Then, while the queue isn't empty and the algorithm doesn't find a winning path, it will remain searching.

The search starts with the first element of the queue and verifies if it is already has a solution for the problem (the red car 'A' at the end of its row). Then, we get all possible states according to the last board of the path, discard the repeated ones, and append every new state to the queue so the algorithm can visit them later and obtain the possible state that come from that state. If the car verifies the winning condition, it returns the path.

To obtain the states for the search function, we use the get_next_states function (well explained in the code comments). Here, we check the empty space around each car, and then we make copies of the board as if the car being processed was moved to each of those empty spaces.

Student.py Changes

This is the section where we pass each state of grid, cursor, and selected to the agent run function. Then, according to our algorithm, the agent will solve it and send the keys necessary to the movement of each individual piece and cursor position/state of selection to the student.py. Then, to always obtain the most recent request and dump the obsolete ones, we use a cycle to handle it.

Colleagues we’ve talked to:

- João Sousa - 103415
- Catarina Costa - 103696
- Tiago Carvalho - 104142
- Miguel Matos - 103341