Introduction to AI: Assignment 1

Due on March 10, 2020 at 11:59pm

Dr. Joseph Brown

Artem Bakhanov (B18-03)

Contents

1 Introduction

1.1 General information

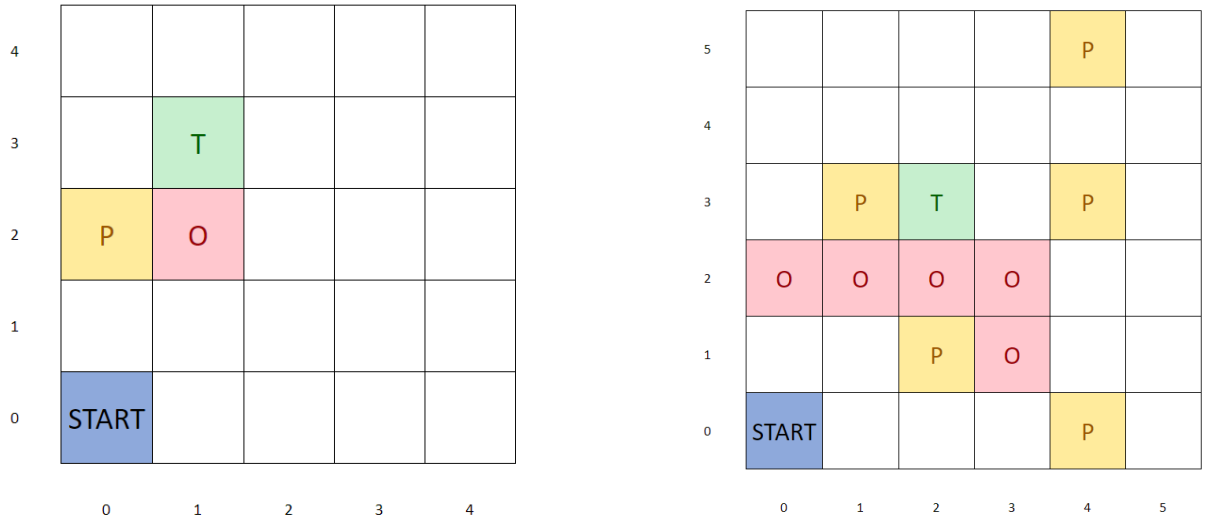
The assignment is solved by me, Artem Bakhanov, a student of Innopolis University. If you have any question regarding any part of this document and other provided materials, you can contact me via email: a.bahanov@innopolis.university.
In this assignment I used SWI Prolog of version 8.0.3-1. It can be easily downloaded from the official website. I highly recommend you to use VCS-Prolog extension for Visual Studio Code.

1.2 Tests

All the tests are created by me (except for the first one, which was taken from the assignment text). You can find them in the directory called "tests". They are simple prolog files with predefined predicates. Note that I use predicates `player(+Position)`, `orc(+Position)`, and `touchdown(+Position)` for defining players', orcs' and touchdowns positions respectively, where `Position` is a position predicate `p(X, Y)` with X and Y coordinates. X-axis is a horizontal axis directed to the right, Y-axis is a vertical one directed upward (please see figures 1a and 1b).

Listing 1: Test 1

```
player(p(0, 2)).
orc(p(1, 2)).
touchdown(p(1, 3)).
max_axis(4, 4).
```



(a) Test 1 (b) Test 2

Figure 1: Examples of tests

Here and throughout I mark players (humans) as P (yellow), orcs as O (red), touchdowns as T (green). The start is always situated at $(0, 0)$. By the conventions, player at the start cell is implicitly specified in the input files.

1.3 Running the code

To run the program you need to execute `main.pl` file and run the following query: `?- start(Number).` where `Number` is 0, 1, or 2, which stands for Backtracking Search, Random Search, and new Search, respectively.

The result will be printed to the standard output of the interpreter.

2 Algorithms

2.1 Backtracking Algorithm

I implemented the backtracking algorithm using the Prolog Tree. The algorithm finds the first solution and returns it. There is no randomness in the implementation, so the algorithm tries to traverse the map spirally. In the backtracking search, the algorithm tries all the possible movements, including ball passes (if it was not performed earlier).
There are several assumptions I made during coding the algorithm.

1. There is no need to return to a cell where the ball was before (i.e. duplicated move).
2. Since the ball cannot return to any old position, there is no need to move players. All the positions are static predicates.
3. There is no need to pass a ball over a cell where it was before. For example, if player 1 passes a ball from $(2, 1)$ to player 1 located in $(2, 5)$ after player 1 came to $(2, 1)$ from $(2, 3)$, then there is no reason no make such a pass because it could be made earlier.
4. Handoff is a move but with 0 cost.
5. The start position of the ball is not saved while performing the backtracking search since the ball cannot enter $(0, 0)$ twice and exit from there because of forbidden duplicated moves.

A very important metric of the algorithm is its time complexity. Since the algorithm is exponentially complex its execution time can be high on the big maps (e.g. 10*10). On big maps with "unusual" field configuration, the algorithm needs a lot of time. For instance, the algorithm does not work on test 7 faster than 5 minutes (I did not run the algorithm for a longer time). Note that the problem with the test is just a `move` predicate order. The algorithm goes in this order: it tries to pass the ball from upward clockwise, then it tries to move a player from upward clockwise.

2.2 Random Search

The random search is very similar to the backtracking search except for the implementation of `move` predicate. I create new predicate `select_random_move(...)`, which literally selects only one move at a time (I used cut to avoid backtracking behaviour). The second difference is that the random search performs exactly 100 moves in one round. I simulate 100000 rounds and select the best solution. I used the same assumptions as I used for backtracking search.
The time complexity of the algorithm is always constant since the algorithm performs the same amount of work for any input (100 moves for each round).

2.3 Improved Backtracking Search

The third algorithm I implemented was actually an experiment that I liked and want to present. In this algorithm I used Prolog Tree but the algorithm does not terminate there but goes further. It selects the best solution dynamically (i.e. while performing). While implementing this algorithm I used several assumptions:

1. There is no need to continue searching if the current path is already longer than the found one.
2. Sometimes it is better to traverse not spirally (like in test 7), but randomly for getting the solution. So special predicate was implemented for random move selection.

Also I used all the assumptions from the Backtracking Algorithm.

3 Advanced Team

3.1 Backtracking Algorithm

There is no improvement of the algorithm since it works like trial and error methods. It **tries**, if no success, it rolls back and try another way.

3.2 Random Search

Random search will not get any improvement too, since it takes purely random moves, which does not depend on the environment.

3.3 Improved Backtracking Search

The situation is the same for this search as for the backtracking search. Since I do not use any heuristic function, extended visibility does not help at all.

3.4 Conclusion

There is no difference when it comes to distance of visible objects, therefore there is no improvement on the algorithms' time complexity. This methods will not make unsolvable map solvable and vice versa. It is important to mention that for some algorithms extended visibility might help to improve its performance. The very obvious examples of such algorithms are A* (heuristic function) and Tabu (Tabu lists can be created without visiting certain cells) searches.

4 Hard Maps

There are different hard or impossible maps for different algorithms. Let us talk about backtracking algorithm.

4.1 Backtracking Algorithm

In case of this algorithm, it is hard to find a solution when the map forces player to start freely move from the center of the map. Example of such a map is input 7 that is shown on the picture below. As you can see, we can treat the start of the field at $(4, 0)$ where the algorithm literally starts from. Since the order of movements are from up to left (clockwise), the algorithm will stuck in the $X = [5, 10]$ area before it goes to the left.

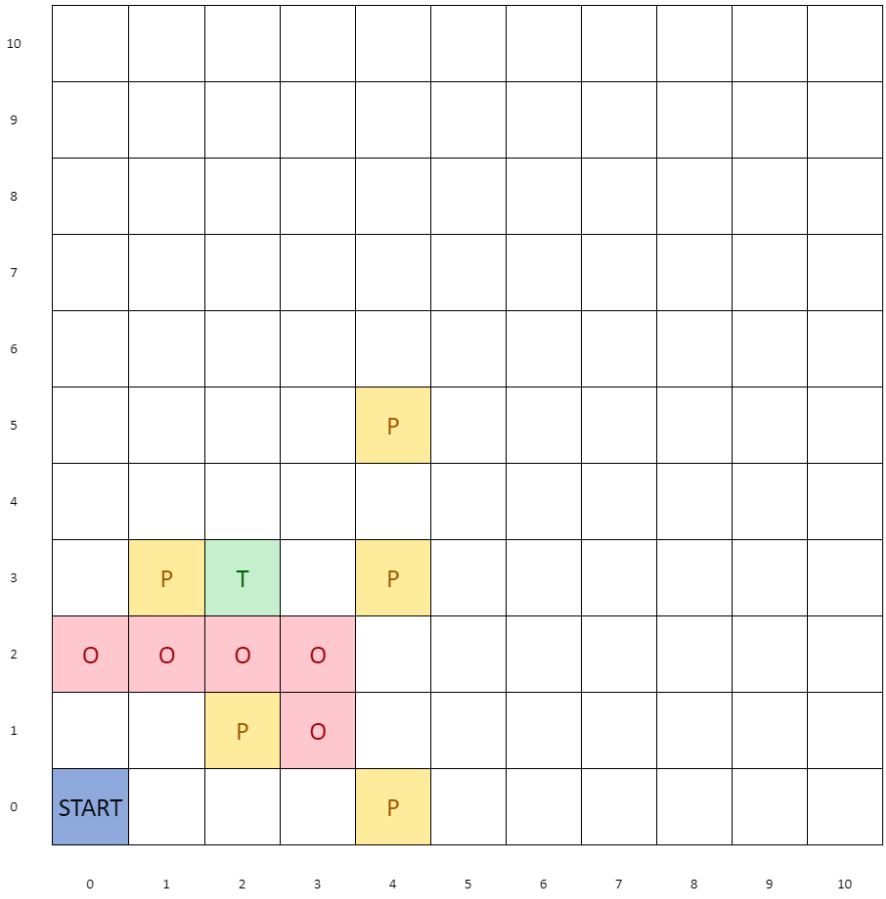


Figure 2: Test 7 - impossible one