Introduction to Artificial Intelligence. Assignment I: Humans vs Orcs Rugby.

Murashko Alecsey,
Group 6

March 9, 2020

Contents

1 Assignment Assumptions and Testing

1.1 Assignment Assumptions

During the implementation of the Assignment, I have made few assumptions according to the assignment description.

1. The field is 20×20 squares.
2. The player currently holding the ball is called Runner.
3. Runner can move one square up, down, left or right.
4. Runner can not go beyond the field's boundaries.
5. If Runner tries to step in the square occupied by another player, then the step will not be counted, whereas will be printed out. Moreover, the step performance mechanism is depicted in Figure 1.
6. The random search takes 100 steps to perform, and if it does not find the solution - it terminates.
7. The Implementation may not be 100% correct and complete, but the places where implementation is not complete are emphasized in the text.
8. The algorithm's Implementation may cause side effects in the instance of the program running. Thus, it is recommended to restart the program after one attempt to solve the problem was performed.
9. Code Examples provided here differ from the actual code itself for clearness and shortness.

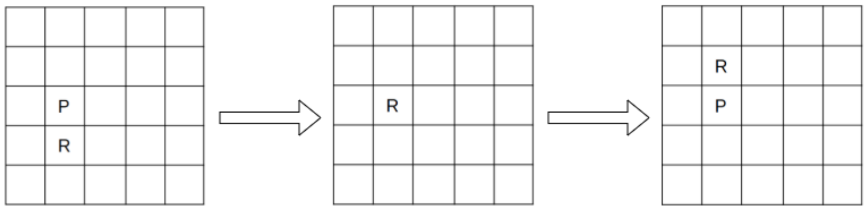
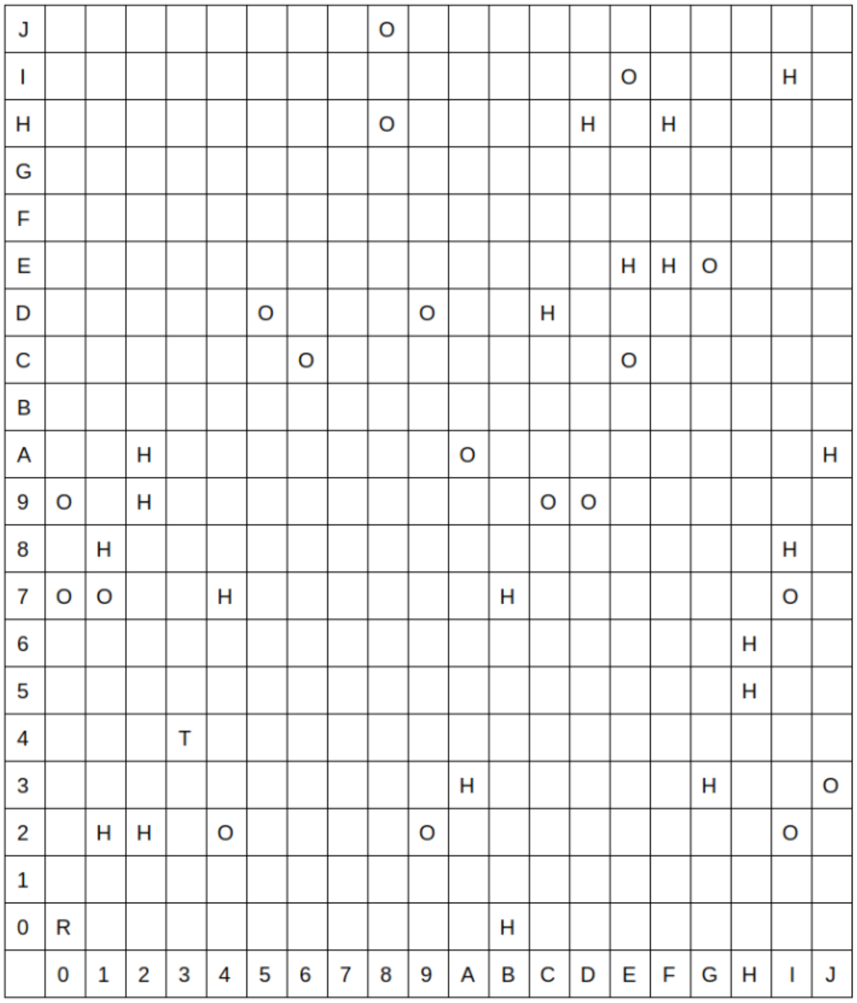


Figure 1: Positions of players by performing $move(1, 2) \rightarrow move(1, 3)$.

1.2 Testing

The Report itself shows the results of how algorithms perform on the map depicted in Figure 2. Statistical analysis was performed on the test cases provided with the report in the folder *Test_Input*.



J									O											
I															O				H	
H									O					H		H				
G																				
F																				
E															H	H	O			
D						O				O			H							
C							O								O					
B																				
A			H								O									H
9	O		H										O	O						
8		H																	H	
7	O	O			H							H							O	
6																		H		
5																		H		
4				T																
3											H						H			O
2		H	H		O					O									O	
1																				
0	R											H								
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J

Figure 2: Test Cases Map

The output of the programs is done as array of objects in the form $p(x, y)$ or $m(x, y)$, where $p(x, y)$ stands for "The ball was passed(thrown) to position x,y" and $m(x, y)$ stands for "Runner moved to position x,y".

2 Random Search

2.1 Basic Idea

The Random Search Algorithm is implemented in recursive manner. All the predicates are recursively calling each other with different arguments. For instance,

$$solve\_game\_random(Pos\_x, Pos\_y, Steps, Nsteps, Max\_steps)$$

is the main function – function where the recursion begins. It may call other functions, such as

$$throw\_up\_right(Pos\_x, Pos\_y, Steps, Nsteps, Max\_steps)$$

or

$$move\_up(Pos\_x, Pos\_y, Steps, Nsteps, Max\_steps)$$

which will determine, whether the Action can be performed, and if so – will change the variables $Pos\_x, Pos\_y, Steps, Nsteps, Max\_steps$ accordingly, and call

$$solve\_game\_random(Pos\_x, Pos\_y, Steps, Nsteps, Max\_steps)$$

with new values of variables $Pos\_x, Pos\_y, Steps, Nsteps, Max\_steps$. The variables, taken by all the functions play different roles:

- Pos_x and Pos_y describe current position.
- $Steps$ is the array of already performed steps.
- $Nsteps$ is the number of performed steps.
- Max_steps is the amount of steps that can be made in current state.

2.2 Code Examples

Let us take a look at examples of the code. Listing 1 shows how recursion base cases are implemented.

```
solve_game_random(_,_,_,_,0):-
    format('Was not able to solve!).

solve_game_random(Pos_x, Pos_y, Steps, Nsteps, Max_steps) :-
(
    t(Pos_x, Pos_y),
        format('ANSWER:~w~nNumSteps: ~w~n',[Steps,Nsteps])
    ).
```

Listing 1: Recursion Base Cases.

The main idea of the algorithm is to compute two random numbers - *Action* and *Direction*. If $Action = 1$, then move is performed – $Direction \in [0,3]$, if $Action = 2$, ball throw is performed and $Direction \in [0,7]$. The part of the code responsible for it is shown in Listing 2.

```
random_between(1,2, Action),
Action is 1,
random(0,4,Direction),
(
    Direction is 0 ->
        move_up(Pos_x, Pos_y, Steps, Nsteps, Max_steps);
    Direction is 1 ->
        move_right(Pos_x, Pos_y, Steps, Nsteps, Max_steps);
    % ------------------------------ // ------------------------------
    solve_game_random(Pos_x, Pos_y, Steps, Nsteps, Max_steps)
)
```

Listing 2: Recursion Move Selection.

2.3 Testing

The Algorithm was tested on different test cases, but for now I will show the result of testing on map depicted in Figure 2. After trying to run the algorithm several times, seeing messages of the form:

```
Was not able to solve!
% 4,273 inferences, 0.001 CPU in 0.001 seconds (99% CPU, 3644455
     Lips)
```

I finally got the answer:

```
###### ANSWER: ######
[m(1,0),m(0,0),p(11,0),m(11,1),m(10,1),m(10,0),m(9,0),m(8,0),
    m(7,0),m(6,0),m(5,0),m(4,0),m(3,0),m(3,1),m(3,2),
    m(3,3),m(3,4)]
With Number of Steps: 16
#####################

% 1,857 inferences, 0.003 CPU in 0.003 seconds (99% CPU, 663188
    Lips)
```

2.4 2 Yards Vision

As far as my algrorithm does not depend on how far my agent can see, new ability will not affect the performance of my agent. So, there is no map which will be unsolvable with this new abolity or which is solvable with it, whereas it was unsolvable without it.

2.5 Hard and Impossible to Solve Maps

As the algorithm performs random search it is possible for it to solve any solvable map. The only hard situation which can happen in reality is the situation in which the shortest path takes more than 100 steps, but this issue can be solved by performing the following change in code:

```
% From:
solve() :- time(solve_game_random(0, 0, [], 0, 100)).

% To:
solve() :- time(solve_game_random(0, 0, [], 0, 400)).
```

2.6 Known bugs and issues to solve

The Random Search algorithm has two bugs.

- It can sometimes perform throwing a ball through the player to another player. This issue can be solved by adding additional verification in all the *throw_∗* functions(As it is in backtracking and greedy algorithms), but it requires to rewrite vast amount of existing code.
- Dirty program architecture, due to its Imperative, but not Declarative nature.
- The Program generates side effects, so you a required to restart a program whenever you are trying to find the solution, using $solve()$.

2.7 Conclusion

The Random Search algorithm is the most unreliable search algorithm, which is easy to code(at least, in imperative programming way). Can be used in different applications as temporary solution for the pathfinding problem.

3 Backtracking

3.1 Basic Idea

Basic Idea of Backtracking implementation is to create knowledge base with rules, which will be able to automatically find the solution, using prolog tree. An example of implementation of sudoku solver, using this approach can be found Here.

3.2 Code Examples

The codebase for the Backtracking implementation consists of two files – *General_Base.pl* and *backtracking.pl*. First file contains common knowledge base for backtracking and greedy search, as well as different libraries used. Second file contains rule $satisfy(Curr, Moves, Passed, Visited)$, which checks whether it is possible to go from $Curr$ to the Tauchdown, by following path in $Moves$. Variables $Passed$ and $Visited$ are used by backtracking's logic and for debug. The main predicate is shown in Listig 3.

```
satisfy(Curr, _, _, _) :-
    Curr = [X0,Y0],
    t(X0,Y0).

satisfy(Curr, Moves, Passed, Visited):- (
  % Obtain X_0, X_1, Y_0, Y_1 from Curr and Moves:
    Curr = [X0,Y0],
    Moves = [[X1,Y1]|R],

    % Check whether current position is available:
    not(o(X0,Y0)),

    length(Moves, X), X #< 50, % Len(Moves) < 50:

    % Check two cases:
    % If |(X0 + Y0)| - |(X1 + Y1)| = 1 => move()
    % Otherwise => pass()
    (
        (
            1 #= abs(X1-X0) + abs(Y1-Y0),
            move(Curr, [X1,Y1]),
            append(Visited, [[X1,Y1]], Vis),
            satisfy([X1,Y1], R, Passed, Vis)
        ); (
            Passed #= 0,
            pass(Curr, [X1,Y1]),
            append(Visited, [[X1,Y1]], Vis),
            satisfy([X1,Y1], R, 1, Vis)
        )
    )
).
```

Listing 3: Main Backtracking Predicate.

As you can see from the code listing, this time main predicate is written fully declarative, as well as all the other predicates.

3.3 Testing

The backtracking algorithm finds the solution for the map depicted in Figure 2:

```
% 1,952,491 inferences, 0.106 CPU in 0.106 seconds (100% CPU,
    18366271 Lips)
Answer: [p(2,2),m(2,3),m(2,4),m(3,4)], with the number of Steps:
     4
true.
```

3.4 2 Yards Vision

As with the Random search algorithm my backtracking does not depend on the radius of vision of my agent. Thus, there is no map which will be unsolvable with this new abolity or which is solvable with it, whereas it was unsolvable without it.

3.5 Hard and Impossible to solve maps

The only limitation of the algorithm is the size of the map. The algorithm works well and finds the solution in reasonable amount of time only for small maps, or in cases when tauchdown is in somewhat favorable position[1].

3.6 Conclusion

Backtracking algorithm always finds one of the optimal solutions, but requires very big amount of time and computational resources. Can be used to solve constraint satisfaction problems or search problems in small search space.

[1]There is no need to perform big amount of steps to get to the tauchdown.

4 Greedy Algorithm

4.1 Basic Idea

The basic idea of the Greedy Algorithm is to use a heuristic function to decide on each step where to go. I've used manhattan heuristic to create my heuristic function. The implementation of the greedy algorithm is imperative and uses recursion, so it looks pretty like the implementation on Python or any other imperative language.

4.2 Code Examples

```
greedy(Curr, Moves, _,_):-
  Curr = [X,Y],
    t(X,Y),
    parse([0,0],Moves, 0, []), !.

greedy(Curr, Moves, Passed, Queue):-
  expand_nodes(Curr, Q1, Passed),
    append(Queue, Q1, Q2),
    minNode(Q2, MinNode, NewQueue),
    MinNode = [_, NextPos],
    (
        (
            (
                (
                    (
                        Curr = [Xc,Yc],
                        NextPos = [Xn, Yn],
                        1 #= abs(Xc-Xn) + abs(Yc - Yn),
                        NewPassed = Passed
                    );(
                        Passed #= 0,
                        pass(Curr, NextPos),
                        NewPassed #= 1
                    )
                )
            ),
            append(Moves, [NextPos], NewMoves),
            greedy(NextPos, NewMoves, NewPassed, NewQueue)
        ); (
            greedy(NextPos, Moves, Passed, NewQueue)
        )
    ).
```

Listing 4: Main Greedy Predicate.

My implementation uses queue, which is implemented as a list with one additional function: $minNode(Queue, MinNode, NewQueue)$, which takes $Queue$, find min node in it, deletes it from the $Queue$, and stores new queue and minimal node in $NewQueue$ and $MinNode$.

Another important predicate here is $expand\_nodes(Curr, Queue, Passed)$, which looks for squares reachable from current position and adds them to the $Queue$ with corresponding heuristic values.

The main predicate of the program is shown in Listing 4. What it does is expands positions reachable from the current one, searches for the best one to take[2] and calls $greedy(...)$ with this minimal node.

4.3 Testing

The greedy algorithm finds the solution for the map depicted in Figure 2:

```
Answer: [p(2,2),m(2,3),m(2,4),m(3,4)], with the number of Steps:
    4
% 63,655 inferences, 0.010 CPU in 0.017 seconds (55% CPU,
   6670889 Lips)
true .
```

4.4 2 Yards Vision

As with the Random search and backtracking algorithms, my greedy search does not depend on the radius of vision of my agent. Thus, there is no map which will be unsolvable with this new abolity or which is solvable with it, whereas it was unsolvable without it.

4.5 Hard or impossible to solve maps

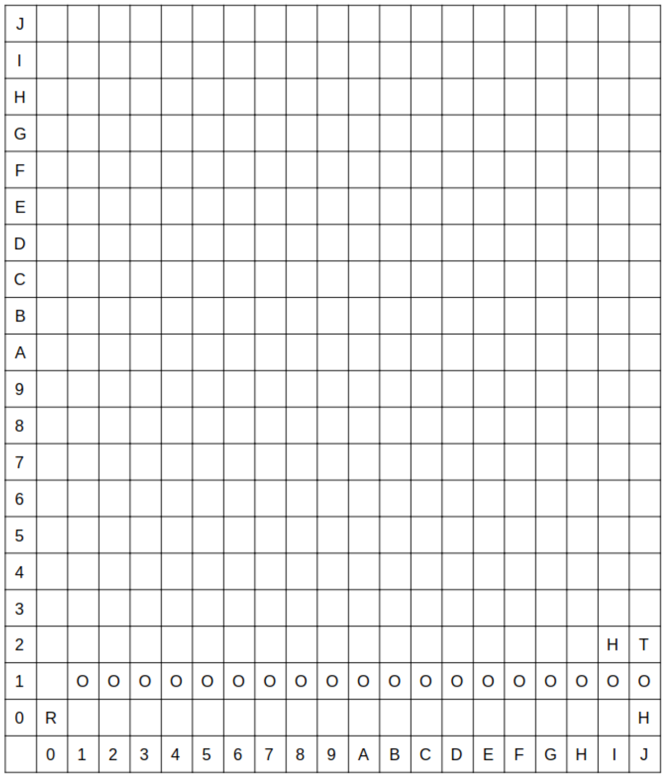
For this algorithm, it is hard to solve maps where the agent is required to accomplish the move, which will have bigger cost than others. In this case, the algorithm will be able to solve the problem[3], but the output will contain some redundant information. Fortunately, such maps are practically rare, but an example of such a map is depicted in Figure 3.

4.6 Conclusion

Greedy algorithm is easy to implement, but it can not help you to surely find the shortest path. This algorithm, due to it's simplicity, can be used

[2]With the minimum heuristic value.

[3]Because the Queue was used in the process of selecting the node.



J																				
I																				
H																				
G																				
F																				
E																				
D																				
C																				
B																				
A																				
9																				
8																				
7																				
6																				
5																				
4																				
3																				
2																			H	T
1		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
0	R																			H
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J

Figure 3: Example of hard to solve map for Greedy Algorithm.

in practical applications, where you do not need to surely find the most optimal path(artifficial enemies in computer games).

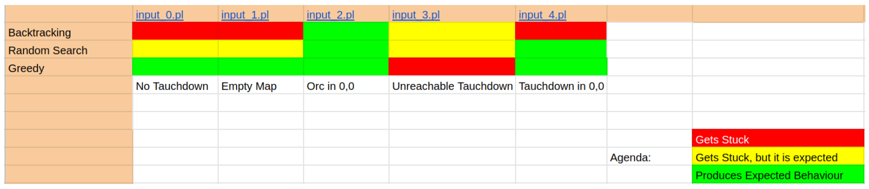
5 Simple Statistical Analysis of the Search Algorithms

This section shows the results of performance of different algorithms runned on 30 different test cases.

5.1 Corner Cases

I've tested my algorithms on different corner cases and obtainde the results shown in Figure 4.

As you can see from the Figure 4, the most stable program is Greedy



	input_0.pl	input_1.pl	input_2.pl	input_3.pl	input_4.pl		
Backtracking	🟥	🟥	🟩	🟨	🟥		
Random Search	🟨	🟨	🟩	🟨	🟩		
Greedy	🟩	🟩	🟩	🟥	🟩		
	No Tauchdown	Empty Map	Orc in 0,0	Unreachable Tauchdown	Tauchdown in 0,0		
							🟥 Gets Stuck
						Agenda:	🟨 Gets Stuck, but it is expected
							🟩 Produces Expected Behaviour

Figure 4: Corner Cases Statistics

Search. Backtracking gets stuck, because it starts to encounter all the possible paths and this takes vast amount of time.

5.2 Dense and Sparse Maps

The following table shows the results of testing different algorithms on maps with big amount of objects. More complete information can be found here.

	input_0.pl	input_1.pl	input_2.pl	input_3.pl
Backtracting Time	14.545		0.350	6.368
Backtracting Length	8		4	7
Random Time		0.001	0.001	0.001
Random Length		17	18	23
Greedy Time	0.027	0.07	0.022	0.24
Greedy Length	10	9	4	11

Google sheets also contain results of tests on maps with 4 touchdowns and on maps with small amount of objects.

5.3 Conclusion

As you can see from this google sheet, backtracking is not capabale of solving any probkem, because it takes him big amount of time, to accomplish the task. Greedy always finds the solution, but this solution is not always optimal, and random search algorithms finds the solution, if you are lucky enough.

A How to Run All the Algorithms

When you unpack your archive, you will see the following structure:

```
root
 |__Automated map creation
 |   |__src
 |       |__Main.java
 |       |__MapCreator.java
 |__Implementation
     |__Backtracking
     |   |__backtracking.pl
     |__Greedy Search
     |   |__Greedy Search.pl
     |__Random_Search
     |   |__random_search.pl
     |__Test_Input
```

backtracking.pl, *GreedySearch.pl* and *random_search.pl* - should be executed in the following form:

```
swipl [Algorithm] [Input]
```

Example:

```
swipl swipl Greedy\ Search.pl ../Test_Input/Dense_1Td/
input_2.pl
```

When you are in SWI-Prolog Terminal, to get the solution, just type:

```
?- solve().
```