Intro to AI - Assignment I

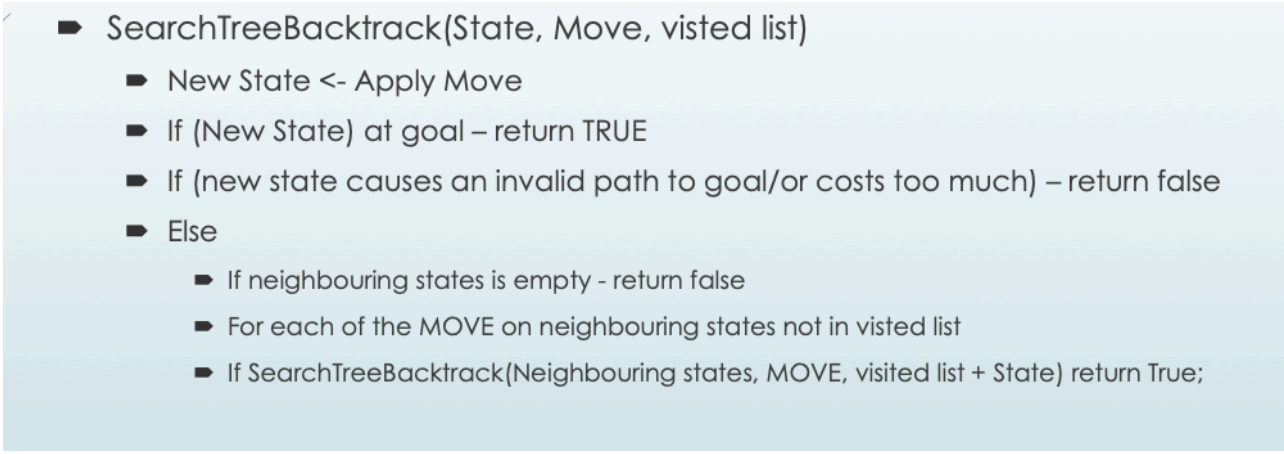
Search algorithms implementation and analysis

Alla Chepurova BS18-01 - March 2020

Part 1

For this assignment I implemented 3 algorithms for searching optimal path in maps representing the orc rugby game.

First one is the backtracking algorithm. It retrieve all possible paths using prolog predicates, find the best solution from all possible paths. My backtracking implementation satisfy the logic of this pseudocode (source - Intro to AI course, lecture 4):



Also I implemented random search algorithm as a second one. It runs 100 times and in each time it try throwing a ball or going in a random direction until there comes a touchdown or an obstacle. In first case that means that algorithm succeeds, in the second one it tries to generate path one more time (no more than 100). Despite the fact that it is a pretty simple implementation it always succeed on the relatively simple maps.

The third one algorithm is an upgrade of the random algorithm, but it uses some sort of heuristics (in files I call it "greedy"). On each steps it goes whether on (1) the cell where touchdown is situated, or (2) the random cell or the cell with human, or (3) if the random cell (2) was unsafe it goes to the safe cell with priority - north, east, south, west. The decision are making with priority (1) -> (2) -> (3). So, on each step the algorithm goes on locally optimal cell, but doesn't stay always on the same rout, as there is a an element of randomness in it. So it could be considered as a combination of exploring and exploiting. As in case of the pure random algorithm there is no guarantee that it find the solution at all. But it succeed in the majority of times, despite the number of steps it makes is usually more than one of random search.

How to run code. To run the particular algorithm there is a line with a query at the very start of every code file. To run algorithm on particular map you need to substitute the predicates in the space commented for maps.

I designed 10 maps (6 solvable and simple, 2 solvable and hard, 2 unsolvable) to show the

correctness of algorithm and variety of possible paths on relatively small 5x5 map. For the possible ones the code outputs the best (in case of greedy the only one) path that can be found with corresponding amount of steps, if the algorithm doesn't succeed in finding the path it outputs the message «algorithm doesn't find the path». Also in each case there is a time which algorithm takes.

Examples:

```
time(run(Steps, Moves, 1000)).
```

<-Query

Two possible outputs. First - algorithm doesn't succeed; second - algorithm found a path.

```
Algorithm didn't find a path
34,533 inferences, 0.009 CPU in 0.009 seconds (100% CPU, 3776319 Lips)
true
```

```
504 inferences, 0.001 CPU in 0.001 seconds (99% CPU, 478372 Lips)
Moves = [east, east, west, north, north, north, south, east, west, south, south, north, north, east, east, east, north, south, north, north, west],
Steps = 21
```

All of 10 designed maps and corresponding predicates for them are at the end of the document in appendix. As I said, these 10 maps includes different types of routes (very short, significantly long or there is no path at all) to demonstrate the possible behaviour of searches in different cases to use this information in further analysis.

To compare and contrast algorithms I used the Two Sample Student T-Test. As a metrics I used the amount of steps produced by algorithms. I compared search methods pairwise and for computing the value of t-statistics I wrote python script (applied with other files), which used T-test formula and produces needed values for each pair of algorithms (to ensure you can run it).
Null hypothesis for pair of algorithms is «there is no significant difference in the mean value of steps produced by two algorithms on different maps». Alternative hypothesis is «The mean value of first one is more (less) than second one». [The formulas - https://en.wikipedia.org/wiki/Student%27s_t-test]

The results of comparing (the number of steps are represented under each of 10 maps in the appendix - they were used as datasets):

Backtracking vs greedy -> t_statistics = -0.35432006645470315, $\alpha = 0.05$ (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of number of steps produced by backtracking search is less than the one produced by greedy search.

Backtrack vs random -> t_statistics = -0.49481378036005336, $\alpha = 0.05$ (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of number of steps produced by backtracking search is less than the one produced by random search.

Random vs greedy -> t_statistics = -0.2760979369252341, $\alpha = 0.05$ (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of number of steps produced by random search is less than the one produced by greedy search.

Conclusions.
Evaluating searches by the amount of steps we got that in this metrics the best one is backtracking, then random, then greedy. But it is important to note that for the last two (difficult) maps random search is not able to provide any solution (more than 100 launches showed this). While calculating the t_statistics based on the number of steps I just didn't include this cases for random search.

But it would be **interesting to look** at the results of different metrics - the number of attempts to run search to get **any** result. For this aim I compared the number of attempts of the greedy and random searches (there is no sense to do it with backtracking as it always provides solution if there is any in first attempt).

The table with numbers of attempts for all the designed maps except the impossible ones (for generally solvable maps, but impossible in case of random algorithm I just substitute relatively large number - 100):

	Numbers of attempts - greedy	Numbers of attempts - random
1st map	1	1
2nd map	1	3
3th map	1	1
4th map	1	3
5th map	1	8
6th map	5	6
9th map	45	100
10th map	1	100

Greedy vs random (number of attempts) -> t_statistics = -0.03614641347724217, $\alpha = 0.05$ (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of number of attempts produced by greedy search is less than the one produced by random search.

That means, that despite the fact that the mean number of steps generated by random search is less than the greedy search one, greedy search is more stable and able to provide at least one solution more frequently.

Part 2

Improvement of greedy algorithm.

Adding the ability to see on two cells further produces next results:

	Numbers of steps - greedy	Numbers of steps - greedy improved
1st map	18	5
2nd map	10	9
3th map	98	2
4th map	12	3
5th map	28	8
6th map	13	6
9th map	32	29
10th map	29	41

Improved g**reedy vs greedy ->** t_statistics = -0.03614641347724217, $\alpha = 0.05$ (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of number of steps produced by improved greedy search is less than the one produced by greedy search.
Looking at the table and gotten result we can see that there is indeed significant difference in step numbers of solution due to this improvement!

Improvement of backtracking algorithm.

As this algorithms generate the best path always - with or without ability to see further, there is no sense to compare amounts of steps. For this aim I compared the time which backtracking and backtracking with improvement demands.

Adding the ability to see on two cells further produces next results:

	Time - backtracking	Time - backtracking with improvement
1st map	0.661	0.775
2nd map	0.585	0.710
3th map	0.559	0.697
4th map	0.166	0.195
5th map	0.049	0.054
6th map	0.747	0.945
7th map	0.205	0.225
8th map	0.01	0.01
9th map	0.025	0.029
10th map	0.008	0.010

Backtracking vs improved backtracking (time) -> t_statistics = -1.0116606592892639, α = 0.05 (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of time in seconds of backtracking search is less than the one of improved backtracking search.
It could seem strange as we added «improvement" in our algorithm. But there is a simple explanation - as backtracking checks every possible path, on each step algorithm with «improvement» checks on one and additionally on two cells further than simple backtracking which adds extra complexity and time demands, making it worse in this case.

Random vs improved random (number of attempts).
Despite that it is a «silly" algorithm I decided to add the heuristics of seeing on two cells further and look at the results in number of attempts (no sense to compare time and step size as they fully depend on randomness).
The results running programs on all designed maps except the impossible ones (100 means not produced the map at all after 100 launches):

	Numbers of attempts - random	Numbers of attempts - random improved
1st map	1	1
2nd map	3	1
3th map	1	1
4th map	3	1
5th map	23	3

	Numbers of attempts - random	Numbers of attempts - random improved
6th map	1	1
9th map	100	37
10th map	100	80

Random vs improved random (number of attempts) -> t_statistics = -0.01908048638878364, $\alpha = 0.05$ (p-value), $\Phi(\alpha = 0.05) = 0.519939$, so $\Phi(\alpha) >$ t_statistics and that means that the mean value of attempts of improved random search is less than the one of random search.

Also it is interesting to note that random search with improvements solves maps which were unsolvable for simple random algorithm before (hard ones - 9th and 10th).

Conclusions.
Ability to see on two cells further improved the random and greedy search algorithms, but in case of backtracking it just added extra complexity and make algorithm works slower.
Also this ability helps to solve hard maps for random algorithm.
However, it doesn't help in solving generally impossible maps for any algorithm (it is logical - there is no valid path in these maps at all).

Part 3

The hard maps are the 9th and 10th ones in appendix. I consider them to be difficult due to the fact that backtracking and greedy algorithms generate longer paths on it than on simple ones (in case of greedy significantly longer paths!), and random (with no improvements) doesn't find the path at all. Ability to see on 2 cells further significantly changes only random search making these maps solvable, but for other two algorithms it still generate large paths.

The impossible maps are the 7th and 8th ones. The arrangement on these maps do not allow them to be solvable at all, so none of algorithms are able to solve these ones. And there is indeed a common feature of these maps - the orcs divide the space of map on two parts - one with start, another with touchdown. So in this case there is no way to come to final stage (touchdown point) and finish the path, so all searches fails.

Appendix - maps

Orc(2,0)				
		H(1, 2)		
H(0,0)				TD(0,4)

Simple map.
Predicates for the map:
orc(2,0).
human(1,2).
touchdown(0,4).
Backtracking
Moves = [east, east, east, east],
Steps = 4
Random
Moves = [east, north, east, east, east, south],
Steps = 6
Greedy
Moves = [east, east, west, west, east, north, south, north, north, west, north, east, west, north, east, east, east, east, west, east, west, south],
Steps = 22

H(3,0)			TD(3,3)	
			Orc(2,3)	
H(0,0)				

Simple map.
Predicates for the map:
orc(2,3).
human(3,0).
touchdown(3,3).
Backtracking
Moves = [(pass_north,3,0), east, east, east],
Steps = 4
Random
Moves = [north, south, east, north, north, south, north, east, north, east],
Steps = 10
Greedy
Moves = [north, south, north, north, north, north, east, west, east, west, east, east, south, east],
Steps = 14

TD(4,0)				
H(3,0)				
	Orc(2,1)			
H(0,0)				

Orc(2,0)		Orc(2,2)		
H(0,0)		H(0,2)		TD(0,4)

Simple map.
Predicates for the map:
orc(2,1).
human(3,0).
touchdown(4,0).
Backtracking
Moves = [(pass_north,3,0), north],
Steps = 2
Random
Moves = [north, (pass_north,3,0), north],
Steps = 3
Greedy
Moves = [north, south, north, north, north, north],
Steps = 6

TD(4,0)				
Orc(3,0)				
Orc(2,0)			Orc(2,3)	H(2,4)
Orc(1,0)				
H(0,0)				

Simple map.
Predicates for the map:
orc(3,0).
orc(2,0).
orc(1,0).
orc(2,3).
touchdown(4,0).
human(2,4).
Backtracking
Moves = [east, north, north, north, north, west],
Steps = 6
Random
Moves = [east, west, east, east, (pass_south_west,2,4), south, west, west, north, south, west, north, north, north, west],
Steps = 15
Greedy
Moves = [east, north, north, north, north, east, east, east, south, west, west, south, west, north, north, west],
Steps = 16

Simple map.
Predicates for the map:
orc(2,2).
orc(2,0).
human(0,2).
touchdown(0,4).
Backtracking
Moves = [(pass_east,0,2), east, east],
Steps = 3
Random
Moves = [north, east, east, east, east, south],
Steps = 6
Greedy
Moves = [north, east, east, east, south, east],
Steps = 6

				Hu(3,4)
H(0,0)		Orc(0,2)	TD(0,3)	

Simple map.
Predicates for the map:
human(3,4).
orc(0,2).
touchdown(0,3).
Backtracking
Moves = [east, north, east, east, south],
Steps = 5
Random
Moves = [north, east, north, east, east, south, south],
Steps = 7
Greedy
Moves = [north, east, south, north, east, east, south],
Steps = 7

			Orc(4,3)	TD(4,4)
			Orc(3,3)	Orc(3,4)
H(2,0)				
H(0,0)				

Impossible (unsolvable) map.
Predicates for the map:
touchdown(4,4).
orc(3,4).
orc(3,3).
orc(4,3).
human(2,0).

Outputs for all algorithms - Algorithm didn't find a path

		TD(3,2)		
Orc(1,0)				
H(0,0)	Orc(0,1)		H(0,3)	

Impossible (unsolvable) map.
Predicates for the map:
touchdown(3,2).
orc(1,0).
orc(0,1).
human(0,3).

Outputs for all algorithms - Algorithm didn't find a path

H(4,0)		Orc(4,2)	TD(4,3)	
		Orc(3,2)	Orc(3,3)	
Orc(2,0)				
		Orc(1,2)		
H(0,0)			Orc(0,3)	

	Orc(3,1)			
TD(2,0)	Orc(2,1)	Orc(2,2)		
Orc(1,0)	H(1,1)			
H(0,0)				

Hard map.
Predicates for the map:
orc(2,0).
human(4,0).
orc(4,2).
orc(3,2).
orc(1,2).
orc(3,3).
touchdown(4,3).
orc(0,3).
Backtracking
Moves = [east, north, north, east, east, east, north, north, west],
Steps = 9
Random
Didn't find in 100 tests
Greedy
Moves = [east, east, west, east, west, north, south, north, north, east, east, south, north, west, east, west, east, east, west, east, north, north, west],
Steps = 23

Hard map.
Predicates for the map:
orc(1,0).
touchdown(2,0).
human(1,1).
orc(2,1).
orc(2,2).
orc(3,1).
Backtracking
Moves = [(pass_north_east,1,1), east, east, north, north, north, west, west, west, south, south],
Steps = 11
Random
Didn't find in 100 tests
Greedy
Moves = [east, north, east, south, east, west, west, east, north, west, east, west, east, west, south, north, south, north, east, west, south, west, east, north, east, east, east, north, north, north, west, east, west, west, west, west, south, south],
Steps = 38