

Assignment I

Introduction to Artificial IntelligenceInnpolois UniversitySpring semester 2020

Report by Alexander Krivonosov

Group 6

Table of Contents

[Disclamer 2](#_Toc34598894)

[Assumptions 2](#_Toc34598895)

[Algorithms explaination 3](#_Toc34598896)

[Random Search 3](#_Toc34598897)

[Backtracking Search 3](#_Toc34598898)

[Backtracking Search with Heuristic 4](#_Toc34598899)

[Algorithm comparison 5](#_Toc34598900)

[Statistical comparison 5](#_Toc34598901)

[Hard to solve and Impossible maps 8](#_Toc34598902)

[Extended field of view (EFV) 9](#_Toc34598903)

[Summary 10](#_Toc34598904)

# Disclamer

To run the program on a default input, perform following steps:

1. Run your prolog compiler (can be either swipl or gprolog)
2. Execute the command to compile knowledge database ([db]. or consult(db).)
3. Run main.

To run your own inputs, update input.pl

## Assumptions

A number of assumptions was made because of the lack of clarifications. Among them:

* While performing random search we become “blind”, meaning we can’t see what’s going on in adjacent cells.
* Backtracking and Heuristic algorithms don’t search for the best path, but for the first successful. Best path can simply be obtained by pressing “;” considerable number of times to backtrack the prolog search tree and find the shortest. “Cut” at the end of the base case marks it as a leaf node and that’s why algorithms can find better paths to the touchdown point. It just takes some time.
* The map is 10x10, not 20x20. Debugging was taking way too long.
* We do not delete human from a cell when we bump into him. Since we consider starting point to be inhabited by a human too, we can think of it as if the original human stayed in that cell and can get ball once again. But this can happen only in Random Search, since it is forbidden to visit already visited cells in other algorithms.
* We can have a touchdown point and an orc at (0, 0). But not a human (because it is already there).
* As already mentioned, in Backtracking and Heuristic algorithms we do not visit cells we have already visited.

And some others which will be described in the text.

# Algorithms explaination

## Random Search

The idea behind my Random Search is following. Function randomSearch runs randomMove 100 times. It compares the result of randomMove to current minimum path length and updates it in case new result is less and it is successful (reached the touchdown point).

randomMove is a function which performs random moves until it either loses the ball, bumps into an orc, or reaches the touchdown. So, one of my assumptions was that in Random Search we are blind and cannot see what’s happening in adjacent cells. This assumption was made after a direct question to a Dr. Prof. Prof. IEEE senior member Prof. Brown.

randomMove works as follows: first, we need to determine whether we are going to make a pass or make a move into one of adjacent cells. This is done with the use of a random number generator.

Probability of getting a pass is an implementation issue, because if we set this probability too high, then it will lose the ball too often and many maps, even the easy ones, will be unsolvable in 100 attempts. After some testing and experimenting (about testing – in the following sections) I figured out that 25% probability is more or less acceptable for easy maps. Hard maps are unsolvable for the Random Search anyway, but about this later.

After we generated this number, we check the state of current cell. If it is taken by the orc, we exit algorithm with FinalState set to orc, if it is a touchdown point, we exit with FinalState set to td. Same happens with an unsuccessful pass. FinalState is set to lost.

After that we follow one of two ways, depending on the random number we got: we either attempt a pass, or move to a random cell. If it is a pass, then we, again using the random number generator, choose the direction of the pass. After that we execute the toss function, which performs some kind of a ray tracking and returns either false if the pass is unsuccessful, or the coordinates of a new cell, otherwise. Then the simulatePath function performs necessary modifications with the ball’s path to proceed the algorithm without any mistakes. If, as in most cases, we are supposed to move to an adjacent cell, then we choose it randomly again, check if it doesn’t violate borders of the map and then simply go there. Then we run the recursive step with current state set to cont.

## Backtracking Search

Backtracking Search is the easiest one, due to the nature of the prolog execution algorithm. I just had to define the rules and it executed it itself. In the Backtracking Search the only issue was the order in which we check our predicates. First one is of course the base case. We check if we are in the touchdown point, and if so, return true. After that I had to firstly check if the touchdown point is in the field of view. Then we directly reach it in one move (this not necessarily applicable to a case with two-cells field of view, but about this later). Next, we check if we can pass in any direction, and if so, we pass. Next, if the adjacent cell is taken by a human, we also head there since it costs us nothing, but can expand our knowledge about the map. If none of the previous cases are applicable, we make a move to an adjacent cell.

## Backtracking Search with Heuristic

The idea of the heuristic I used for this algorithm is following: since we want to explore the map as fast as possible to determine where the touchdown is, we should prompt cells which open the biggest number of unknown cells first. So, we must maintain a data structure with all the discovered cells (not visited, but discovered). So, each time we prompt a cell we must understand how many new cells visiting it will open to us and choose one with the highest number. All this is done by the function chooseBest. I want to go through it step by step:

chooseBestRec(C, Adj, Discovered, CF, Max, CMin) :-

(

Adj = [],

listAdjacent(C, [], 0, AdjN),

colisions(AdjN, Discovered, 0, Coll),

countList(AdjN, AdjLen),

(

(AdjLen - Coll > Max), \+o(C) -> (CF = C);

(CF = CMin)

)

);

(

listAdjacent(C, [], 0, AdjN),

colisions(AdjN, Discovered, 0, Coll),

countList(AdjN, AdjLen),

(

(AdjLen - Coll > Max), \+o(C) -> CollN is AdjLen - Coll, (Adj = [CN|T], chooseBestRec(CN, T, Discovered, CF, CollN, C));

(Adj = [CN|T], chooseBestRec(CN, T, Discovered, CF, Max, CMin))

)

).

chooseBest(C, Discovered, CN) :-

listAdjacent(C, [], 0, Adj),

Adj = [H|T],

chooseBestRec(H, T, Discovered, CF, 0, C),

CN = CF.

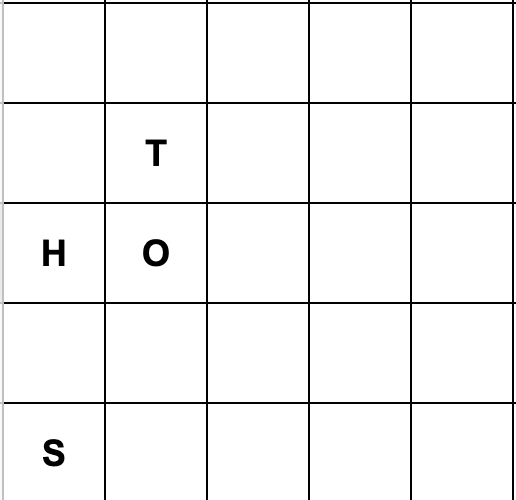
So, as you can see, this function is actually two functions, one of them is a recursive call. So, we start by listing all adjacent cells to our current cell. “Decapitate” this array and run recursion. Now let’s look through chooseBestRec. First pair of brackets is a base case – we will return to it later. In second brackets we do following: list all adjacent cells to currents cell (ones that are probably not discovered yet). Then find number of collisions between this list and list of already discovered cells (function collisions treats lists like sets and find number of elements that appear on both sets). Then, in case number of not discovered cells (computed by subtracting number of collisions from the length of list of adjacent cells) is greater than current and there is no orc, update current best cell. Then we run recursive case. In the base case we perform all the same computations, but not run the recursive case.

In all rest, this searching algorithm is a simple backtracking. The only difference is in the choice of the cell where we want to go. In backtracking it is determined simply by the order in which we prompt them. Here, this choice is a bit cleverer. It is not always an advantage, on the contrary, it sometimes gives even worse solutions in terms of the number of steps. All this will be discussed in the next section.

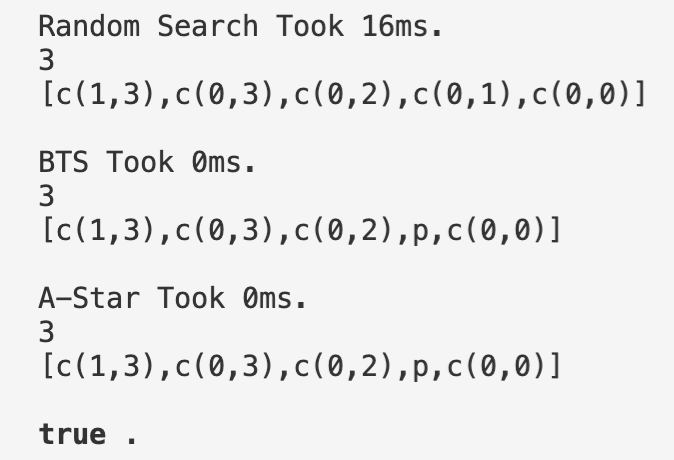
# Algorithm comparison

## Statistical comparison

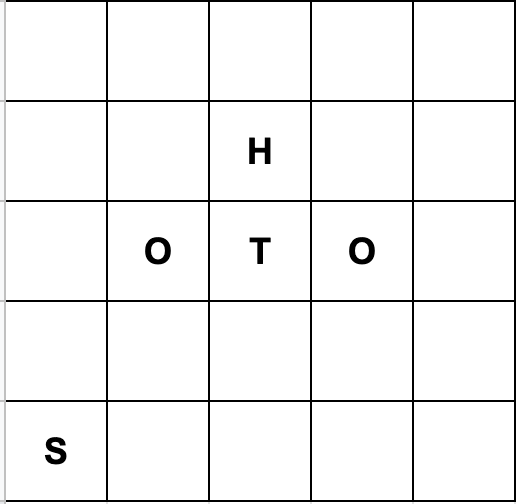
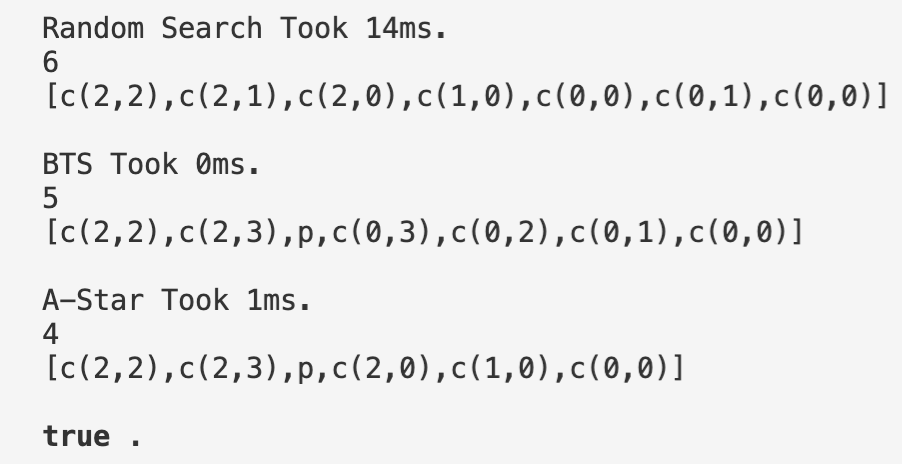
First, let’s consider relatively easy maps. For example, the default one:

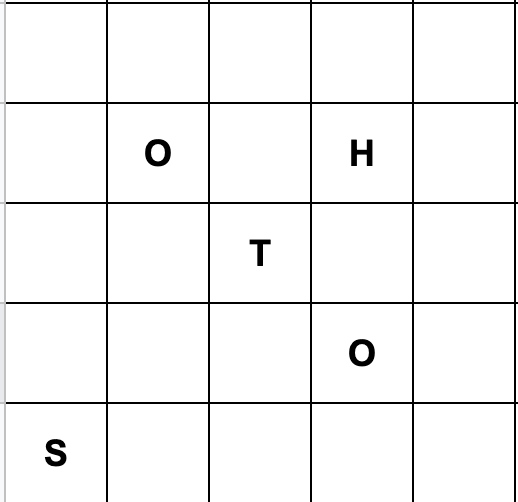
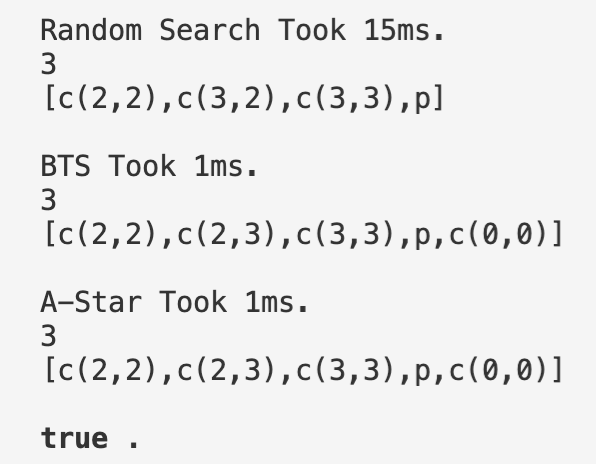


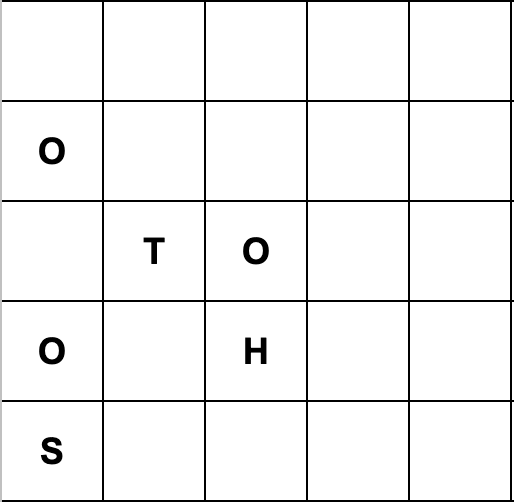
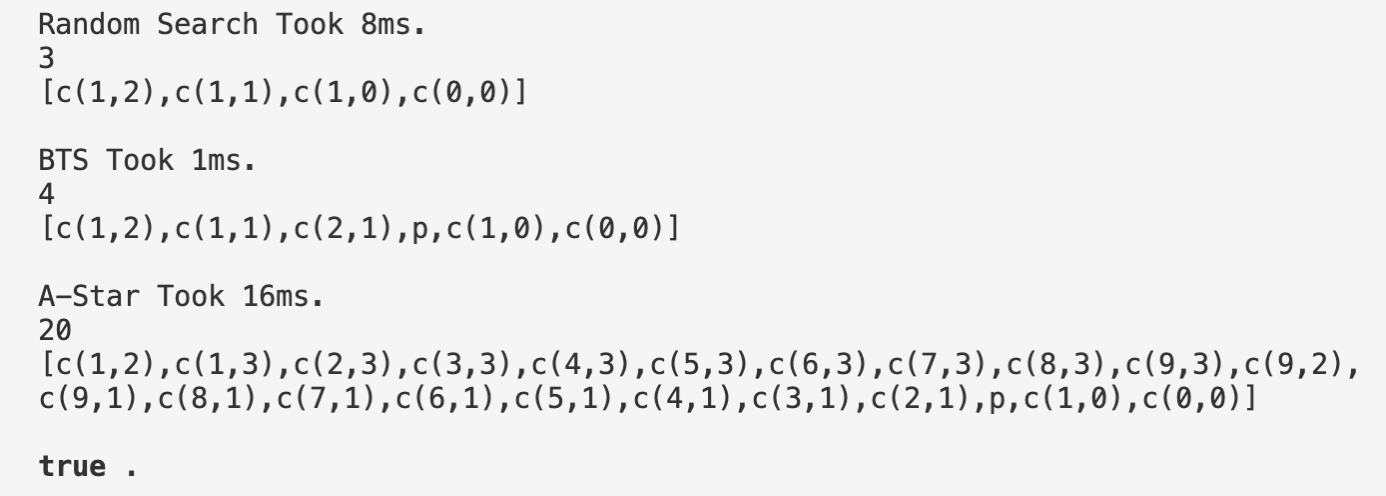
All three algorithms give us approximately the same result:



We can notice that Random Search took considerable time, whereas Backtrack and Heuristic were almost immediate. Let’s go through more examples:

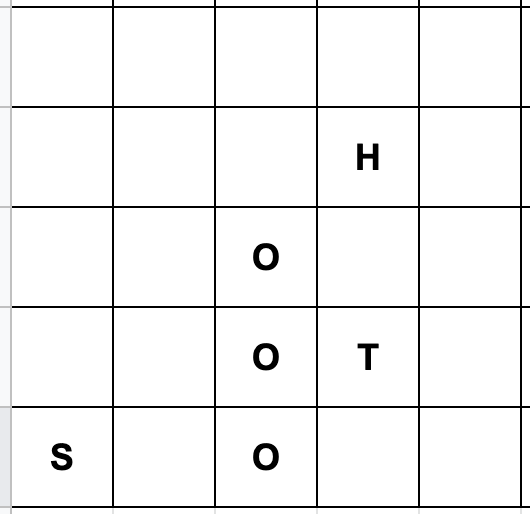
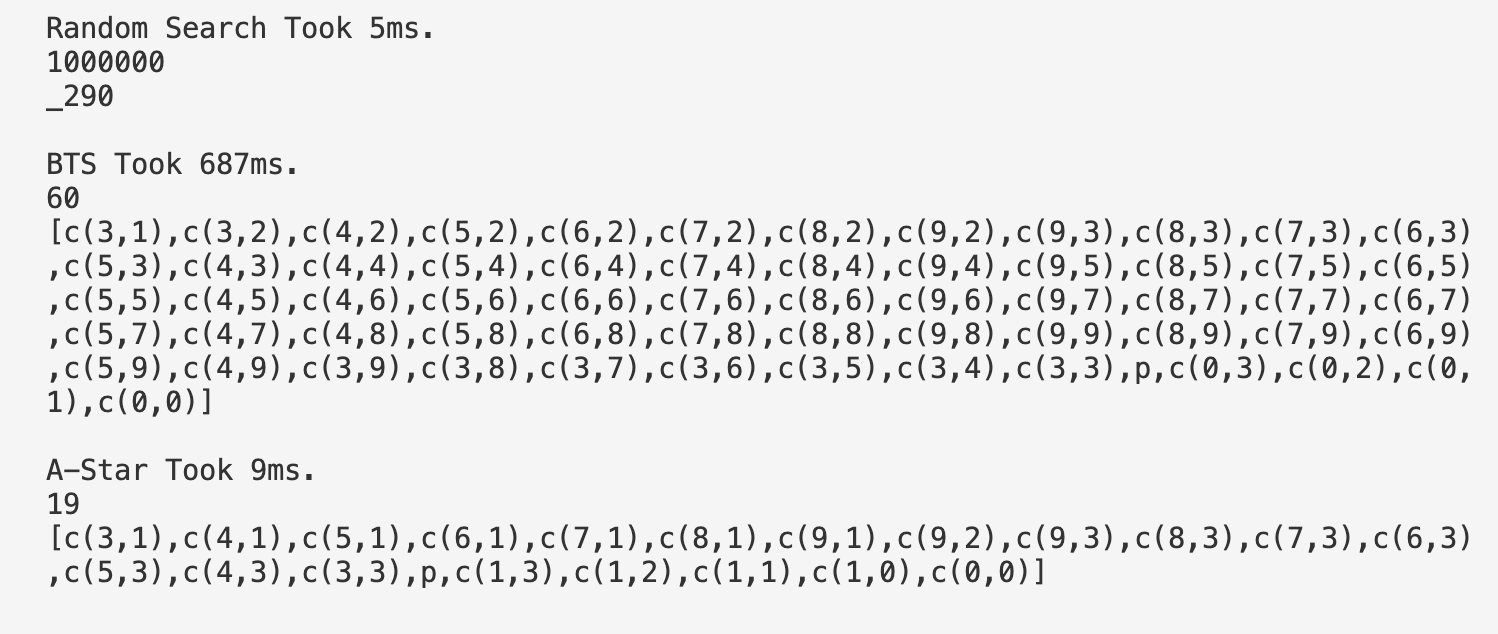
 

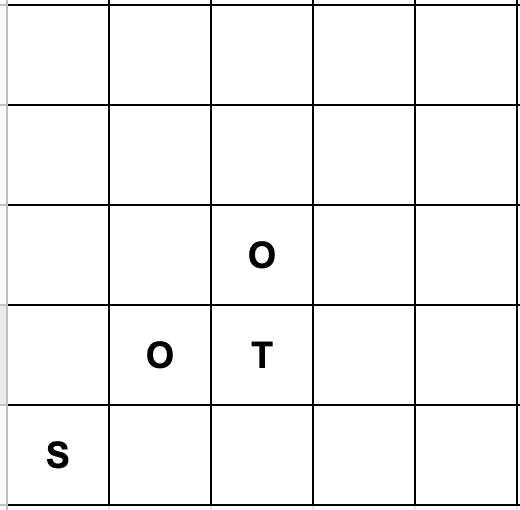
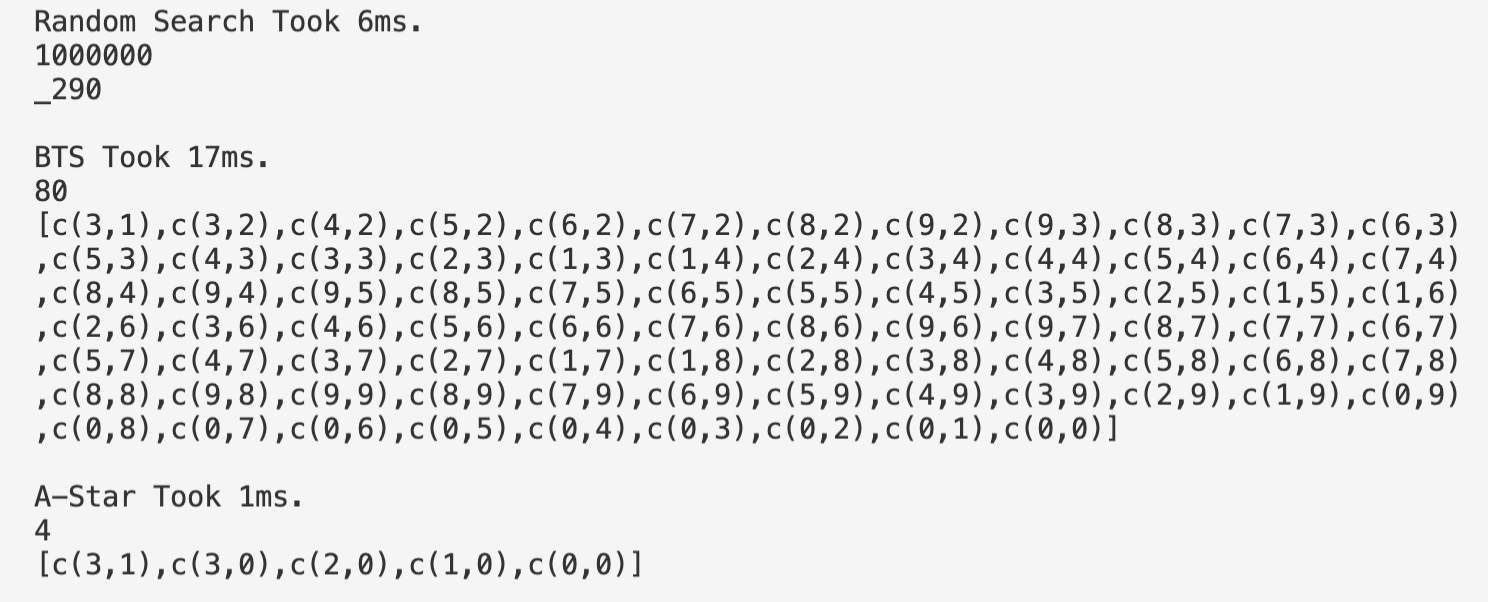
 

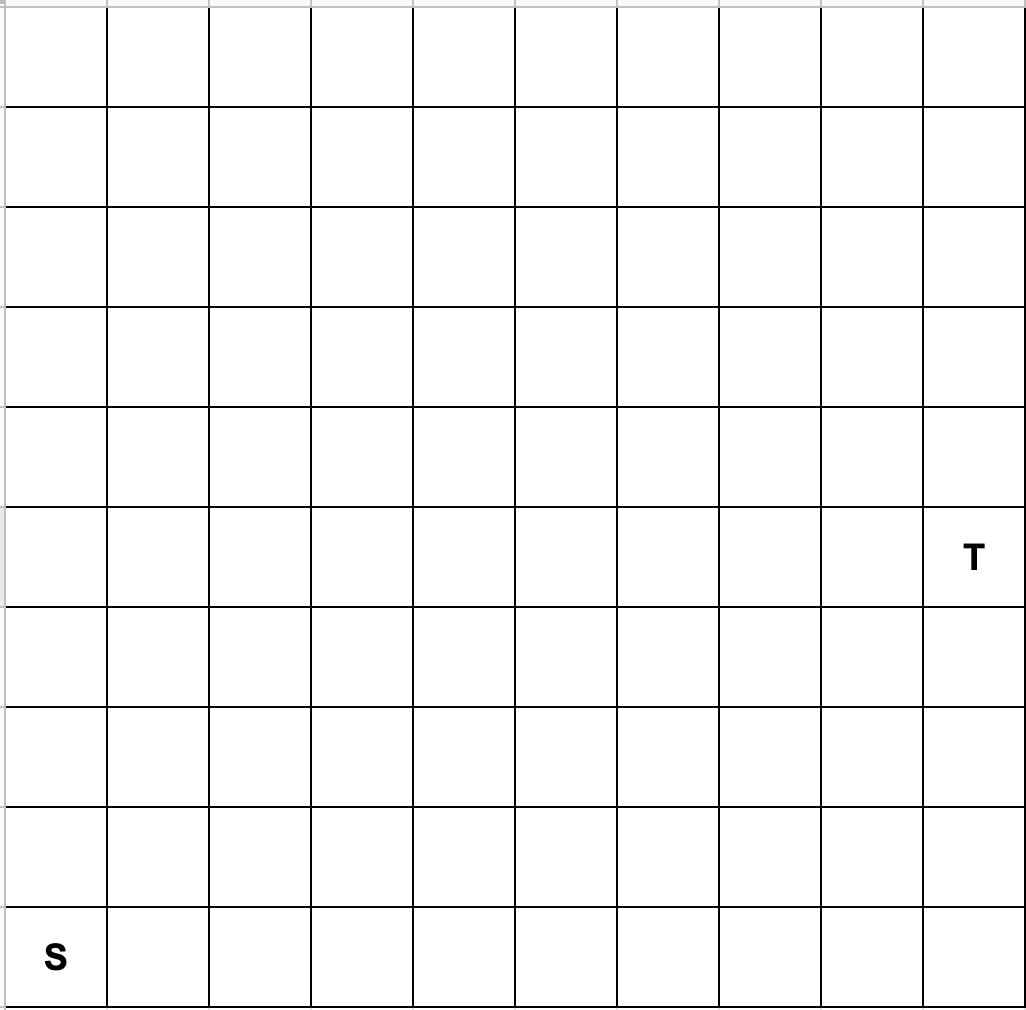
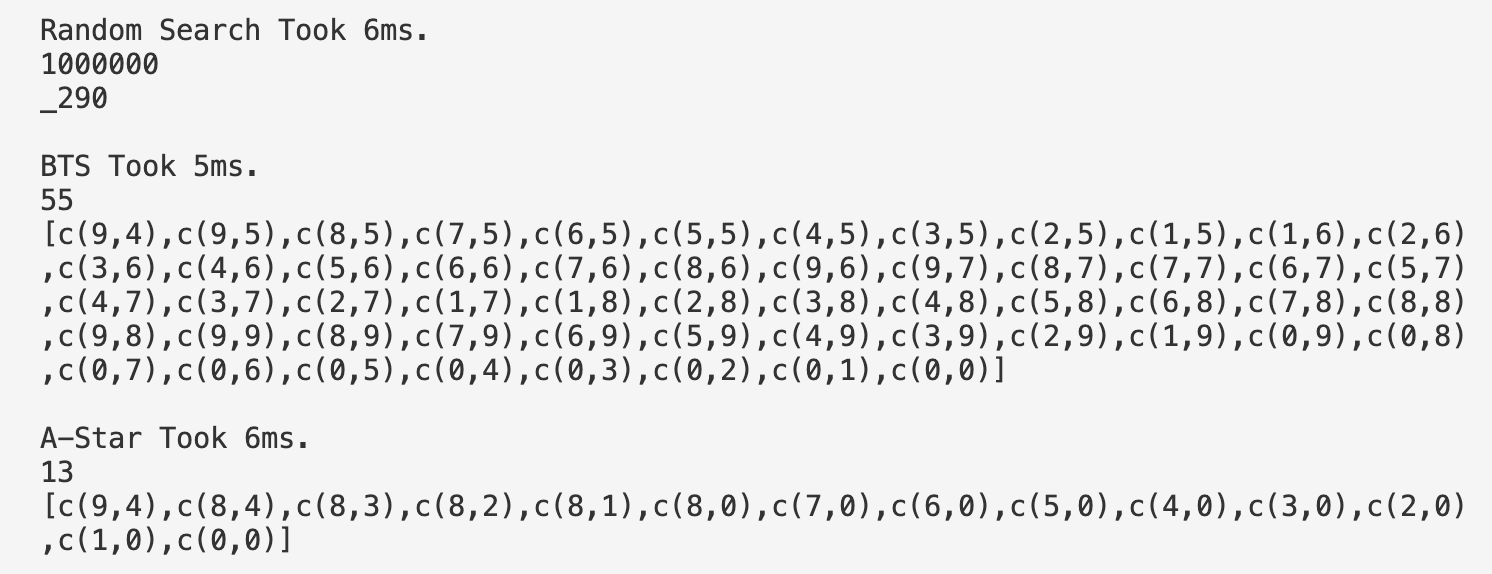
 

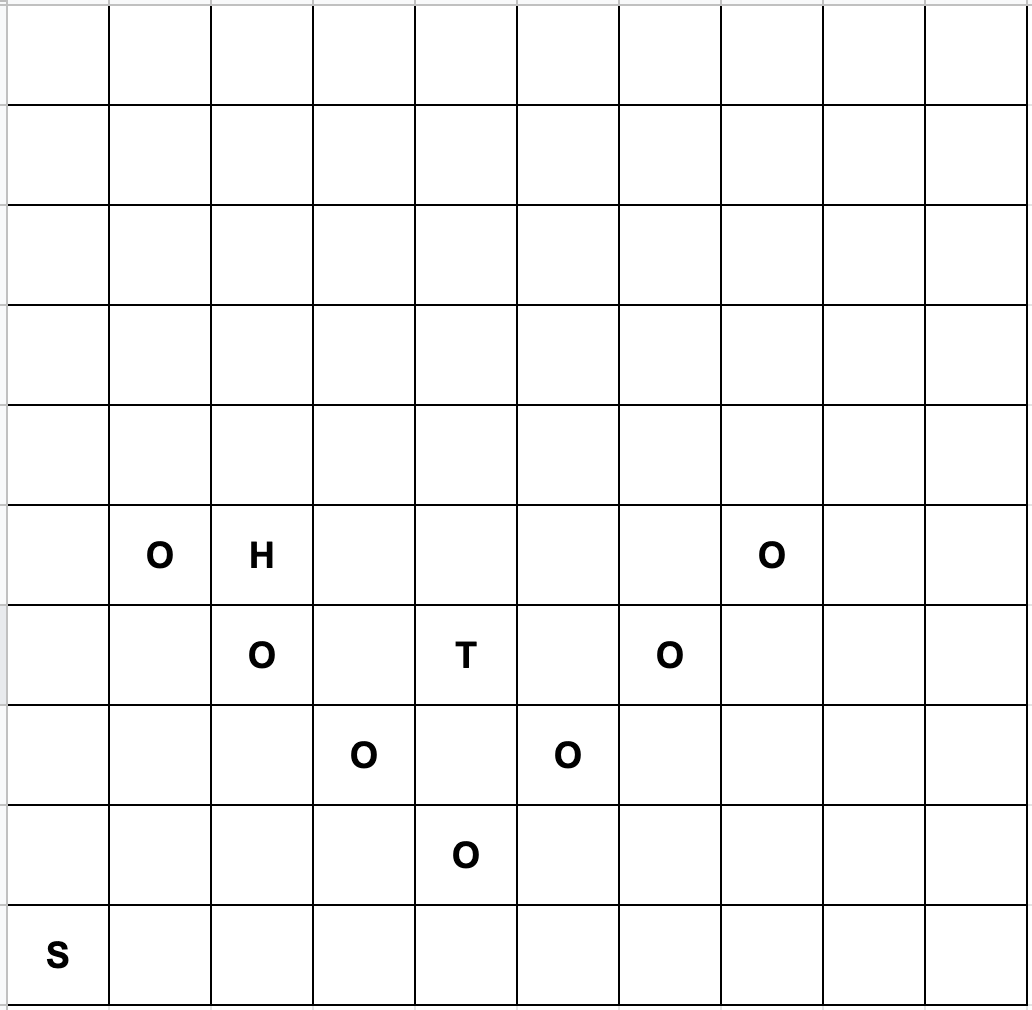
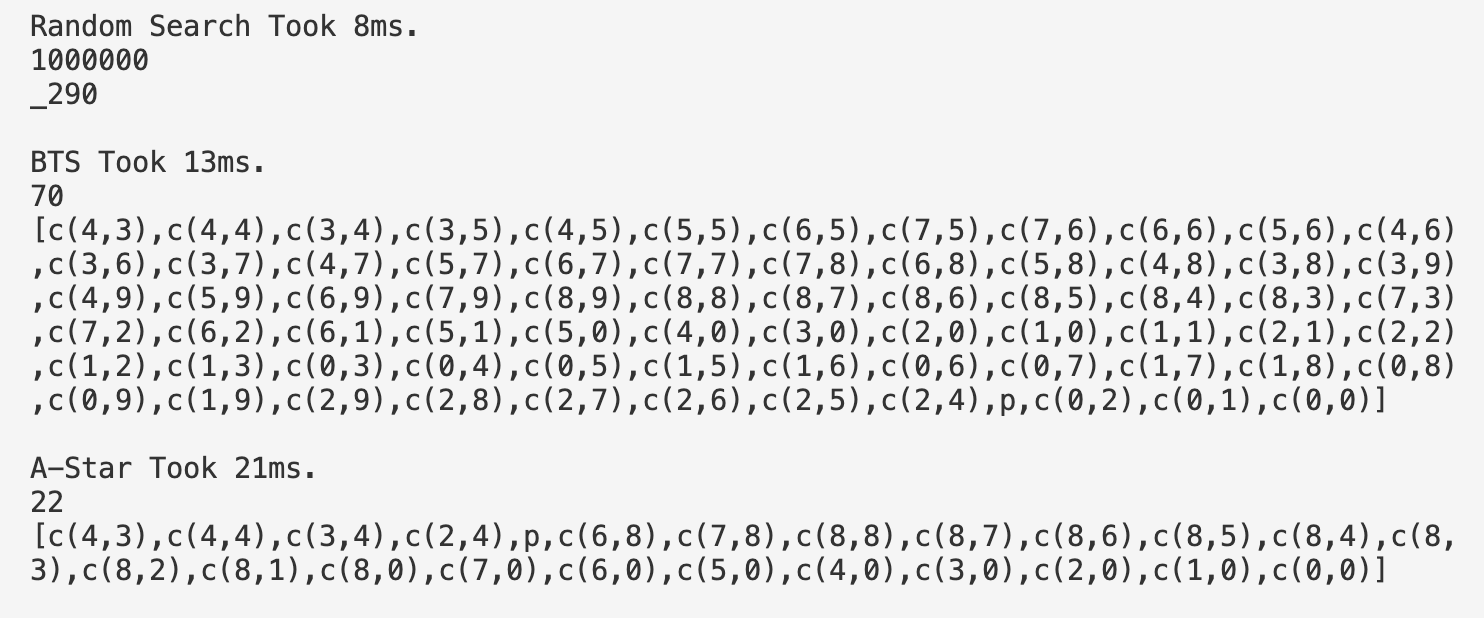
On the last map, as you can see, Heuristic (in code this function is called A-Star, although it’s not an A-Star, but nevermind) gave us unexpectedly poor result. The only reason for this is an order in which we star prompting cells around us. In this case simply going left after we received the pass was the best idea, although, according to our heuristic, going right would have given more information to us. This information turned out to be not so valuable.

But in general, The Heuristic algorithm performs much better:

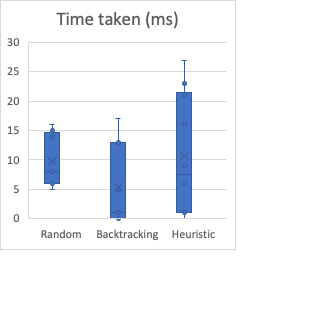
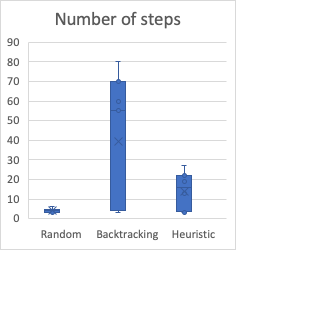
 

What we can clearly notice here is that Random Search already cannot handle these maps. 100 attempts is simply not enough for them. One more noticeable moment – Heuristic algorithm on average performs slower. Last example illustrates it best. Although the path is shorter, it took more time to compute then the Backtrack.

After a number of different tests, the statistic is following:

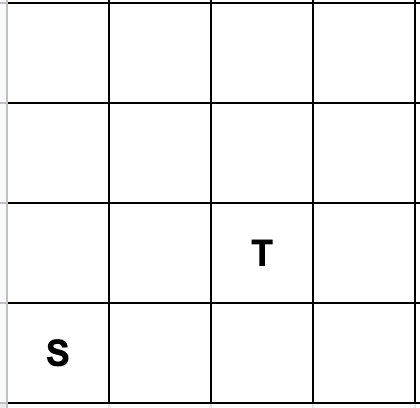
 

Data about Random Search is not really relevant, since it failed in the majority of the attempts and the data about these attempts was not considered. Also, we can see, that on average the number of steps taken by the Random Search in 4, which is, obviously, because of the pass rate which I set to 25%. The reason for poor time performance of the Heuristic algorithm is that it computes expensive chooseBest on each step.

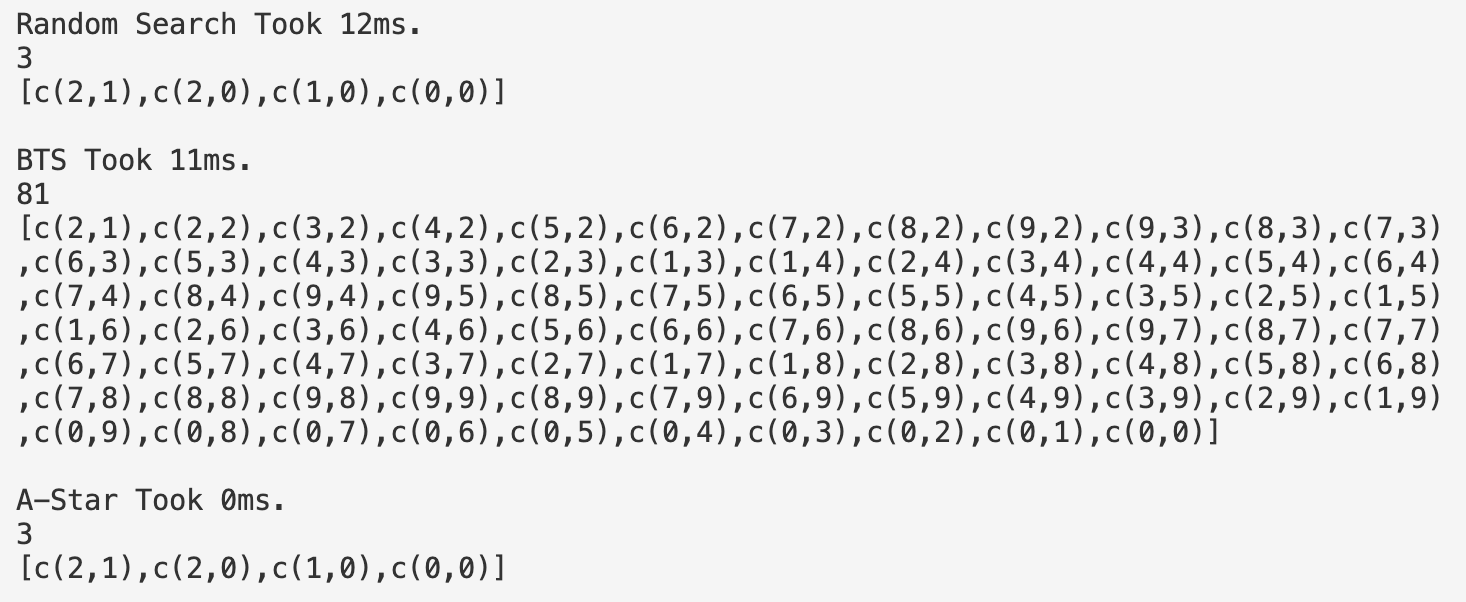
Maps considered above can already be called hard-to-solve and even impossible for the Random Search. Let’s take a look at some more such maps and try to figure out common pattern between them.

## Hard to solve and Impossible maps

One of the reasons for algorithms not being able to solve some maps was described above: order in which we prompt cells around us matters a lot. Following map can be a good example for this:



Looks very easy. And it is for Random and Heuristic, but not for the Backtracking:



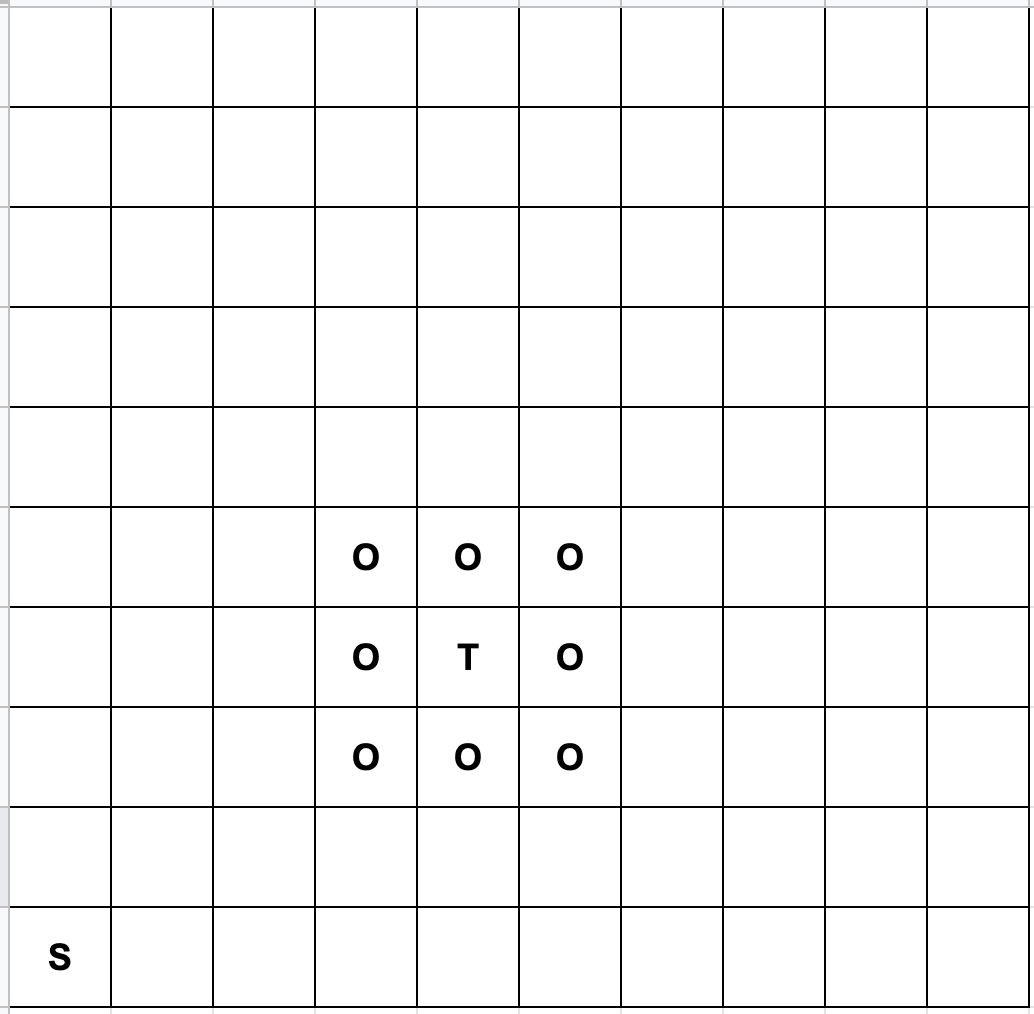
Default order for cell exploration in Backtracking is Up, Left, Right, Down. But if we change it to Right, Left, Up, Down the result will change dramatically:



And again, as you can see, Heuristic takes more time to execute then the Backtracking.

Another common reason for maps to be hard to solve is, of course, big number of orcs on the map. Only the fact that there are many orcs on the map doesn’t mean anything, but they can, again, confuse our algorithms by making them go to a wrong direction. Several examples were shown before.

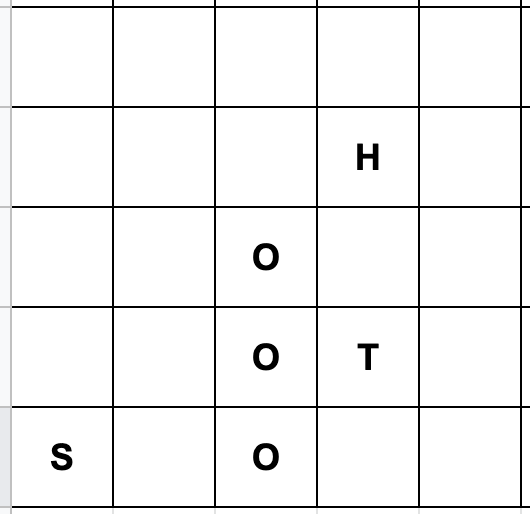
Impossible maps exist in case the touchdown point is unreachable, like this:



Backtracking will spend a long time attempting to find the path and will fail in the end. And, obviously, a map with no touchdown point will be unsolvable, as well as a map with an orc at c(0, 0).

## Extended field of view (EFV)

If we extend our field of view to two yards it can solve some maps more efficiently. For example, let’s consider the map we’ve already looked at.



Here it took backtracking 60 steps to find the touchdown point, but with an EFV it would take only 5. But, of course, it is obvious that EFV can help us to solve some maps more efficiently. More interesting question asked in the task description is whether it can harm our productivity.

There is no possibility that EFV will make a solvable map unsolvable because using backtracking we can always find the solution. And it won’t solve unsolvable maps, since they are UNSOLVABLE. The only thing that you can do to solve them is to violate the rules. Also, I couldn’t find any map where it could worsen our productivity. It only adds power to the heuristic. If our player could see, where the touchdown point is, but couldn’t reach it directly (for example, there is an orc between them) it still knows where the point it wants to reach is and can dedicate all its powers to exploration of the area around this point in order to find out how to get there. This is a very powerful tool to improve our algorithms.

# Summary

Now I would like to make a conclusion about results I have got. Random Search is clearly not capable of solving most of the maps. Passing is almost always unsuccessful and omitting this feature, I believe, would lead to even better results. The Backtracking algorithm is no surprise performs pretty good. It will necessarily find the path; the only issue is the time it will take. Heuristic algorithm was supposed to improve this value, but failed. On average it takes less attempts to solve the problem, cause the heuristic works pretty well (but not always), but at the same time it takes more time to execute each step, so two these factors balance each other. Of course, it depends on the tests you run and, as we could see, depending on the map, one of mentioned factors can dominate and affect the solution more then another. In conclusion, I want to say that I would prefer Heuristic algorithm to other two.