Part 1. PEAS

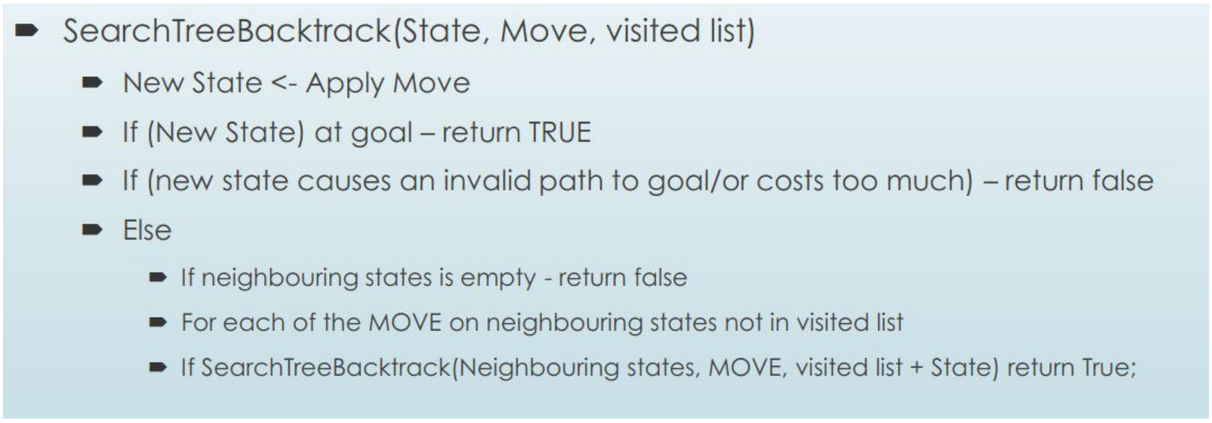
Performance Measure - the number of moves the actor makes to the home.
Environments - the board that is used in the game itself
Actuators - walking horizontally, vertically, diagonally, taking the mask/doctor. At each state the actor makes some movement, and also, if he has a mask/doctor, he can walk in the covid zone.
Sensors - a sensor to determine if the covid zone is close, a sensor to know where the house is, a sensor to check if there is a mask/doctor in our cell

Properties of the Environment:

- Partially observable - since we “perceive covid if you are standing next to the covid infected cells” (or variant 2)
- Single Agent - only one actor agent
- Deterministic - since we can determine the next state of the environment given the previous state and the action we are applying into it (there are no unforeseen actions here).
- Sequential - the agent decides where to go based on past steps
- Static - environment does not change as the agent is thinking about its actions
- Discrete - there is a fixed number of states
- Known - the agent have full knowledge of the rules of the environment

Part 2. Algorithms

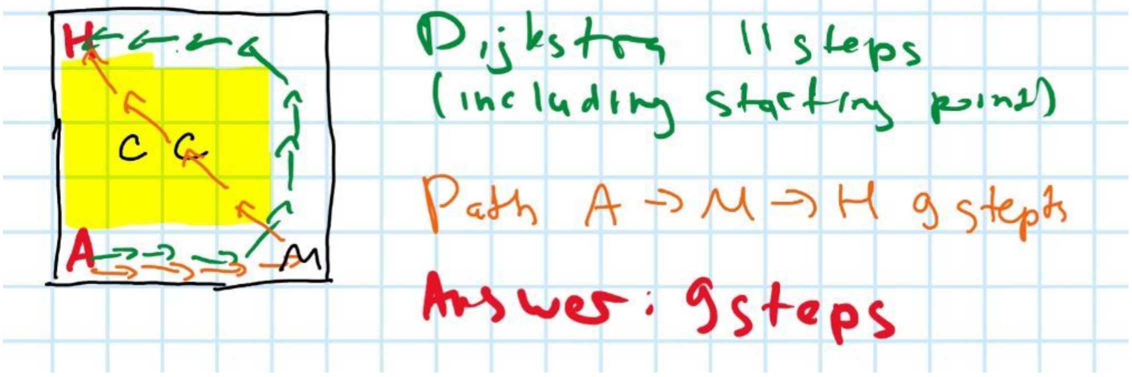
For this assignment I implemented 2 algorithms for searching the optimal path.
First one is Backtracking. It retrieves all possible steps using Prolog predicates, and finds the best solution from all possible paths. My implementation satisfy the logic of this pseudocode (taken from week4 lecture):



```
SearchTreeBacktrack(State, Move, visited list)
    New State <- Apply Move
    If (New State) at goal – return TRUE
    If (new state causes an invalid path to goal/or costs too much) – return false
    Else
        If neighbouring states is empty - return false
        For each of the MOVE on neighbouring states not in visited list
        If SearchTreeBacktrack(Neighbouring states, MOVE, visited list + State) return True;
```

Second one is the Dijkstra variation. The basic idea is to find how many steps you need to take to all the other possible points from the actor (in my case, these are all points that are not in the covid zone). For this reason, we have an array of costs, which stores the number of steps to a given point. There is also an array that stores the shortest path to a given point.
So, from each point not yet visited, we go to the neighboring ones and update arrays of costs and paths if necessary. All previously unvisited points are added to the queue. The recursion ends when the queue becomes empty. (The main function is called dijkstra)
Thus, we went through the entire map (without entering the covid zones) and found the shortest path to any point from the actor's point. However, we did not take into account the presence of the mask/doctor and the passage through the covid zone. So next we need to consider these two cases and identify the best one. It will be the answer.

Let us consider this case: (A - actor, M - mask, C - covid, H - home)



Here, the path to the house after our algorithm will be 11 steps. However, after our algorithm, we know how many steps it takes to reach the mask (in this case, 5), and from the mask to the house it is easy to calculate using this rule:

```
distance(FromX, FromY, ToX, ToY, Distance) :-
   Distance is max(abs(ToX - FromX), abs(ToY - FromY)).
```

After comparison, we understand that the path through the mask is shorter, so it will be the answer.

All in all, I have 2 files: backtracking.pl & dijkstra.py
To run program, you need to choose one of the 6 (e. g. `test5`) tests and run the following query:
?- test5, main.

If you want to run your own test, write the necessary data to the `test1` rule at the beginning of the file and run the following query:
?- test5, main.

The output will be an array of steps (path), and the number of steps including the starting point. (or `false`, if the path does not exist).

- *In my implementation, the map starts at index 1, not 0.*
- *I also didn't implement the doctor, since the mask and the doctor are the same thing. If you want to add both the mask and the doctor to the map, just add two mask*

Part 3. Statistical comparison

Dijkstra's algorithm is O(n * n), since it checks all the points, not just until we get home. So, in most simplest cases backtracking is faster (as you can see in tests 1-5), but if the case is not so easy, then Dijkstra's algorithm works faster (see test 6).

Backtracking

Test	Attempt 1, s	Attempt 2, s	Attempt 3, s	Attempt 4, s	Attempt 5, s
test1	0.002	0.002	0.002	0.003	0.003
test2	0.002	0.002	0.002	0.002	0.002
test3	0.003	0.004	0.005	0.003	0.004
test4	0.001	0.001	0.001	0.001	0.001
test5	0.002	0.003	0.002	0.002	0.002
test6	9.922	9.842	9.860	9.819	9.813

Dijkstra

Test	Attempt 1, s	Attempt 2, s	Attempt 3, s	Attempt 4, s	Attempt 5, s
test1	0.004	0.004	0.004	0.005	0.004
test2	0.088	0.102	0.099	0.120	0.086
test3	0.145	0.135	0.143	0.126	0.125
test4	0.002	0.002	0.002	0.002	0.002
test5	0.140	0.132	0.142	0.109	0.125
test6	0.112	0.099	0.094	0.099	0.097

Backtracking vs Dijkstra:
Let us calculate T-value for these maps. Let us choose significance level as 0.05 (p).

The *t*-value is 2.2965. The result is significant at *p* < .05.

This means that the mean value of the time calculation produced by Dijkstra is less than the one produced by Backtracking.

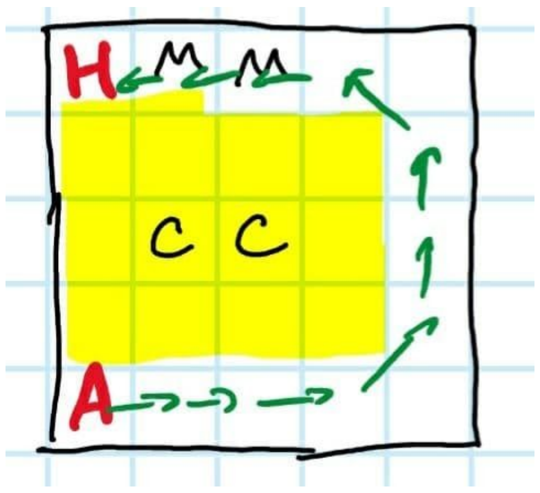
Note:
Since I didn't find any time changes in the two implementations of each algorithm, I decided to just compare Backtracking (v1) and Dijkstra (v1). I believe that the difference did not happen due to the fact that backtracking goes through all possible

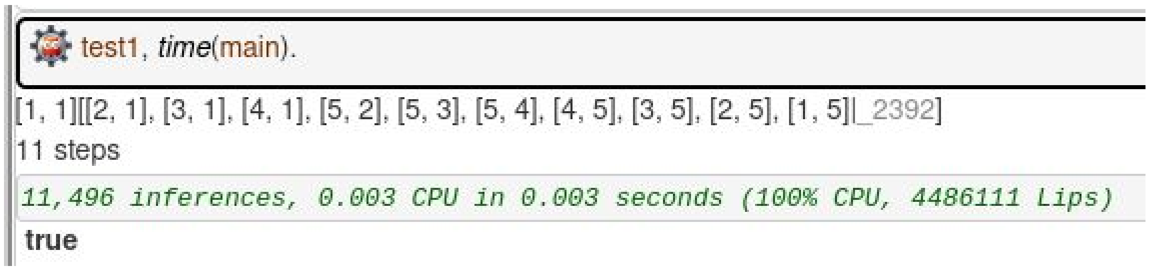
paths and it does not matter how far the covid is. If we talk about Dijkstra's algorithm, then there are no changes due to the fact that Dijkstra goes through the entire map and counts the cost for ***each*** *cell.*
In addition, from the tests, we can conclude that Dijkstra's algorithm works basically for about the same amount of time for all maps of a given size.

Part 4. Examples

Backtracking

1)

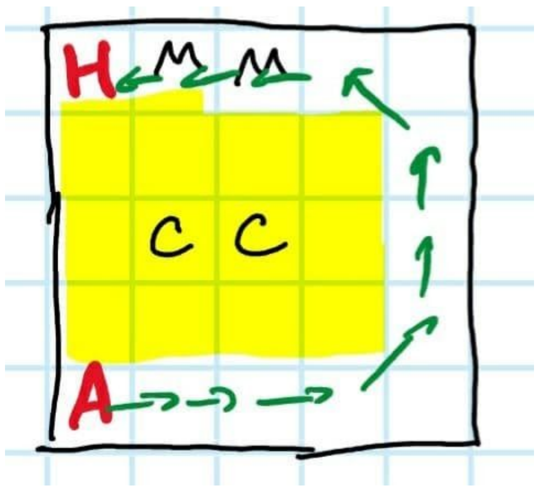


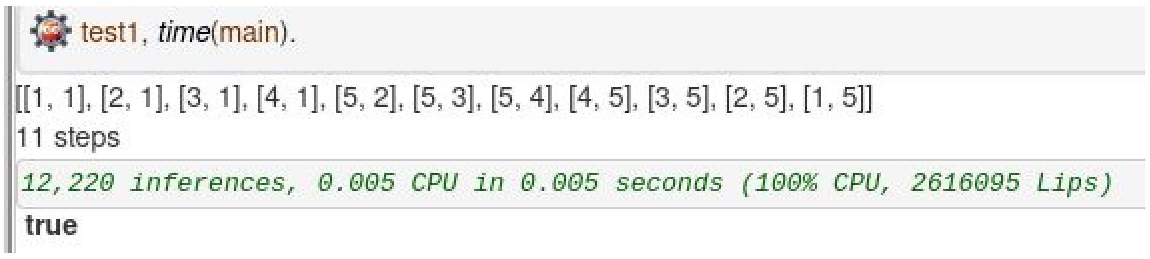


```
test1, time(main).
[1, 1][[2, 1], [3, 1], [4, 1], [5, 2], [5, 3], [5, 4], [4, 5], [3, 5], [2, 5], [1, 5]|_2392]
11 steps
11,496 inferences, 0.003 CPU in 0.003 seconds (100% CPU, 4486111 Lips)
true
```

Dijkstra

1)

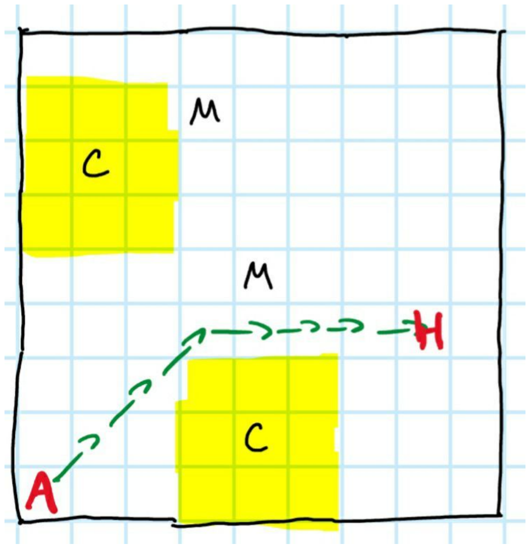


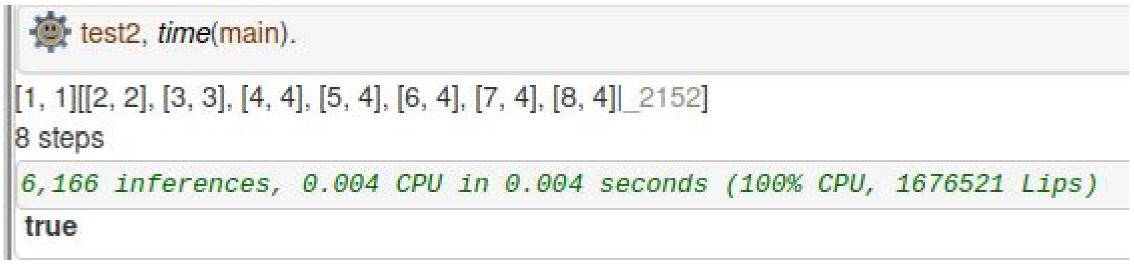


```
test1, time(main).
[[1, 1], [2, 1], [3, 1], [4, 1], [5, 2], [5, 3], [5, 4], [4, 5], [3, 5], [2, 5], [1, 5]]
11 steps
12,220 inferences, 0.005 CPU in 0.005 seconds (100% CPU, 2616095 Lips)
true
```

Backtracking

2)

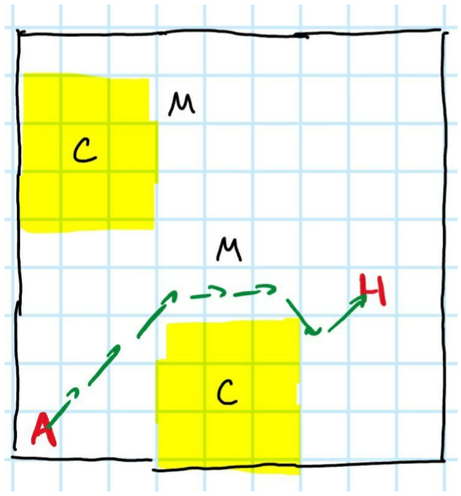


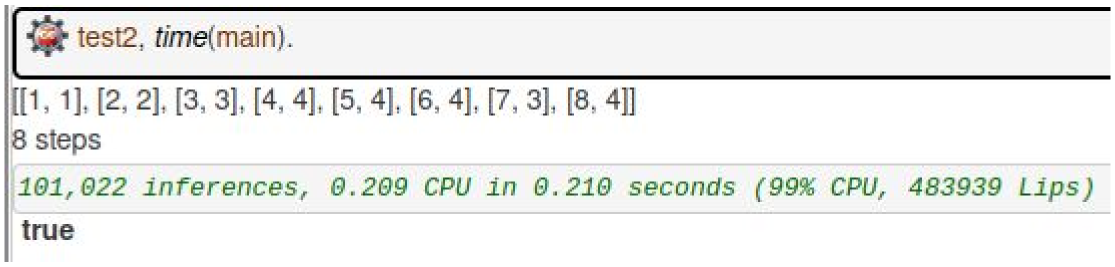


```
test2, time(main).
[1, 1][[2, 2], [3, 3], [4, 4], [5, 4], [6, 4], [7, 4], [8, 4]|_2152]
8 steps
6,166 inferences, 0.004 CPU in 0.004 seconds (100% CPU, 1676521 Lips)
true
```

Dijkstra

2)

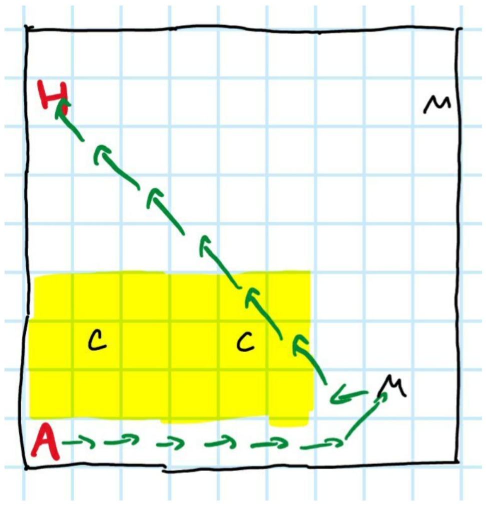


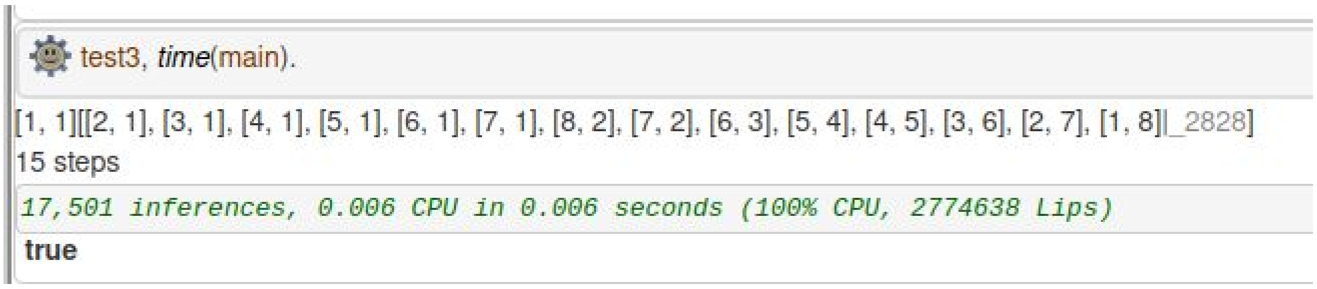


```
test2, time(main).
[[1, 1], [2, 2], [3, 3], [4, 4], [5, 4], [6, 4], [7, 3], [8, 4]]
8 steps
101,022 inferences, 0.209 CPU in 0.210 seconds (99% CPU, 483939 Lips)
true
```

Backtracking

3)

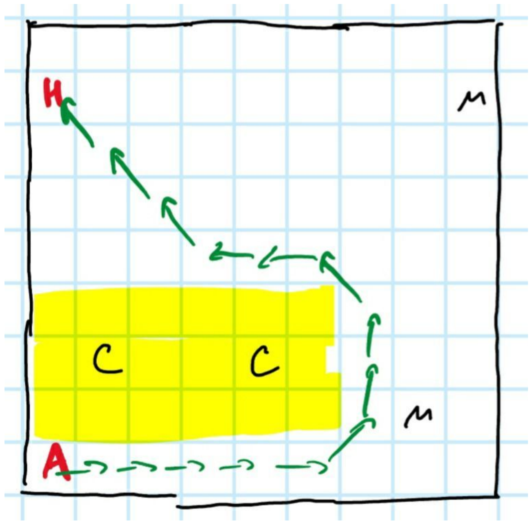


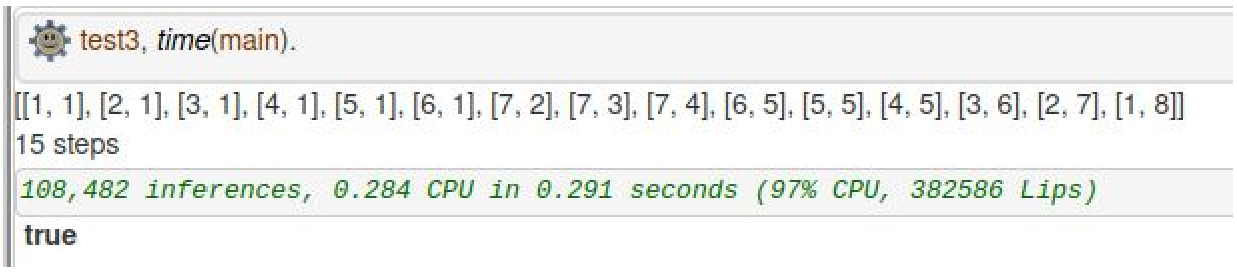


```
test3, time(main).
[1, 1][[2, 1], [3, 1], [4, 1], [5, 1], [6, 1], [7, 1], [8, 2], [7, 2], [6, 3], [5, 4], [4, 5], [3, 6], [2, 7], [1, 8]|_2828]
15 steps
17,501 inferences, 0.006 CPU in 0.006 seconds (100% CPU, 2774638 Lips)
true
```

Dijkstra

3)

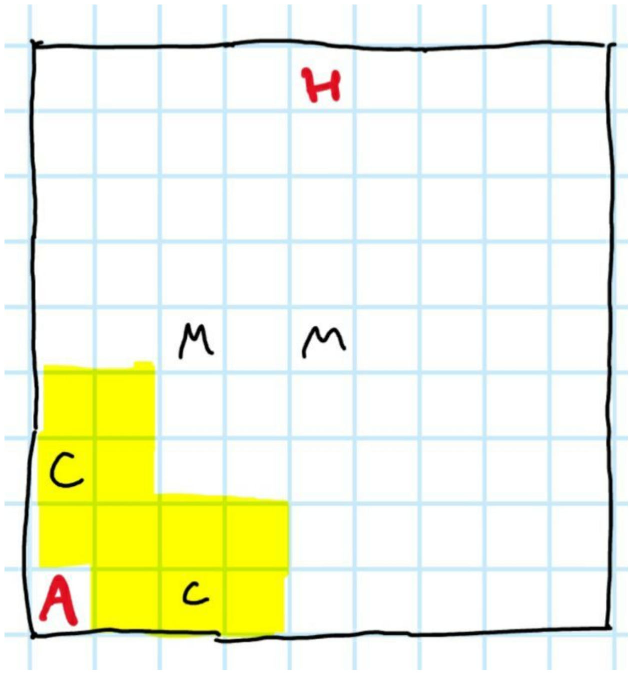


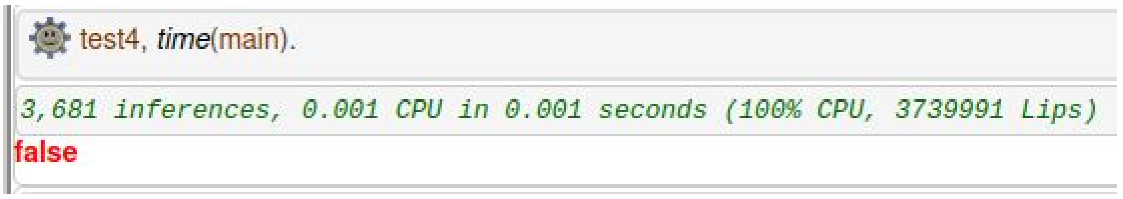


```
test3, time(main).
[[1, 1], [2, 1], [3, 1], [4, 1], [5, 1], [6, 1], [7, 2], [7, 3], [7, 4], [6, 5], [5, 5], [4, 5], [3, 6], [2, 7], [1, 8]]
15 steps
108,482 inferences, 0.284 CPU in 0.291 seconds (97% CPU, 382586 Lips)
true
```

Backtracking

4)

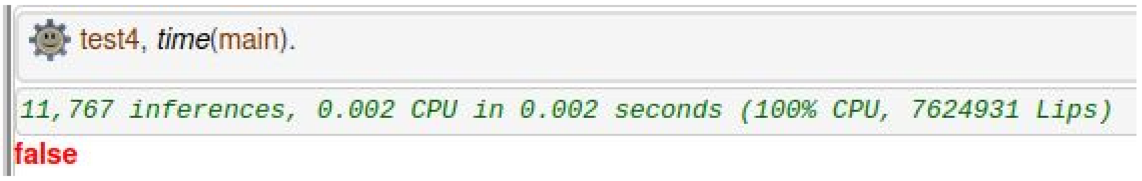




```
test4, time(main).
3,681 inferences, 0.001 CPU in 0.001 seconds (100% CPU, 3739991 Lips)
false
```

Dijkstra

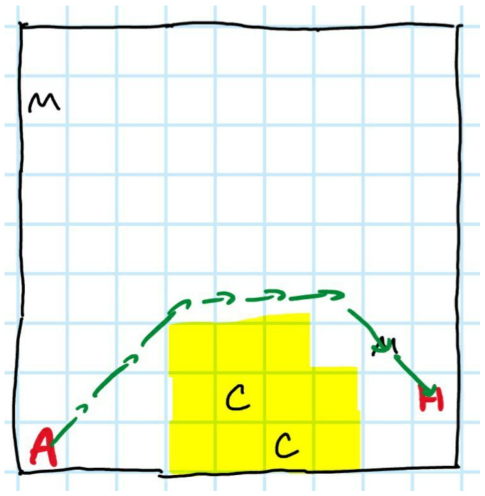
4)

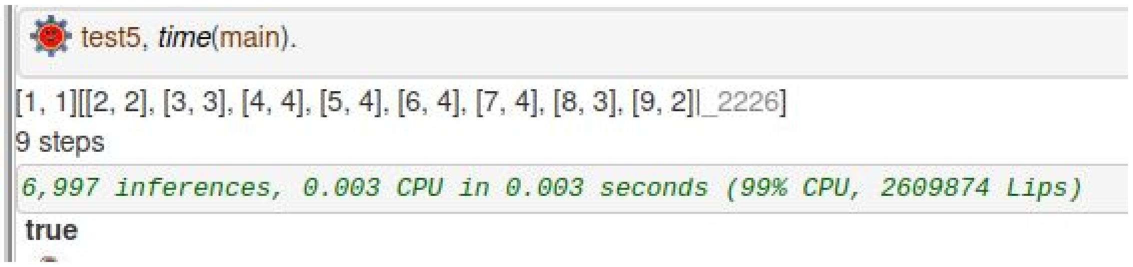


```
test4, time(main).
11,767 inferences, 0.002 CPU in 0.002 seconds (100% CPU, 7624931 Lips)
false
```

Backtracking

5)

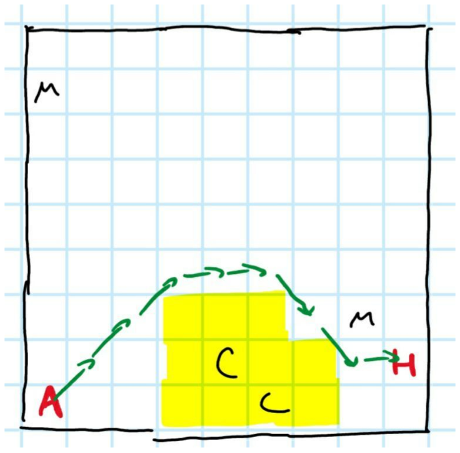


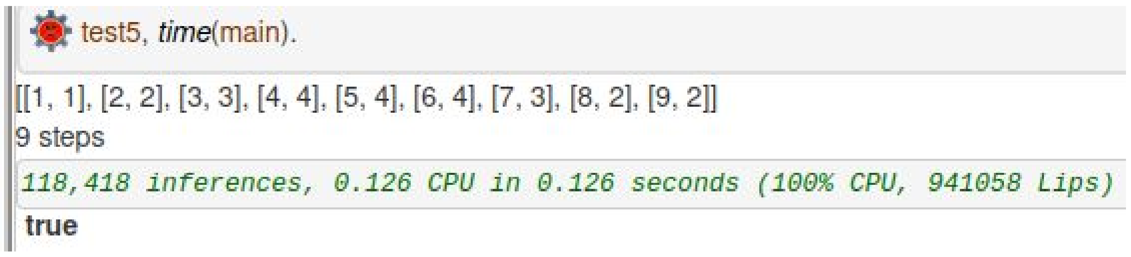


```
test5, time(main).
[1, 1][[2, 2], [3, 3], [4, 4], [5, 4], [6, 4], [7, 4], [8, 3], [9, 2]|_2226]
9 steps
6,997 inferences, 0.003 CPU in 0.003 seconds (99% CPU, 2609874 Lips)
true
```

Dijkstra

5)

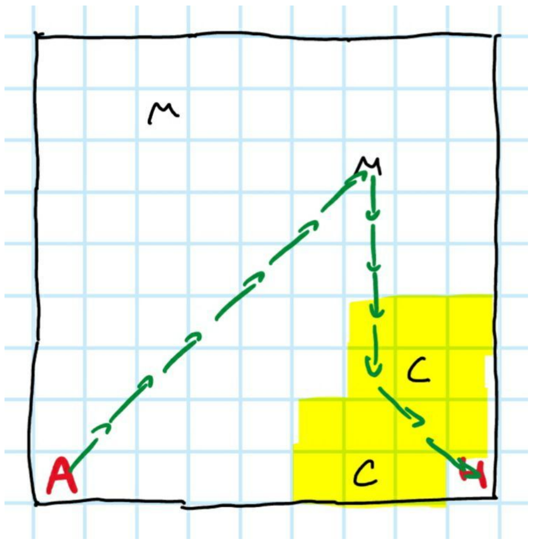


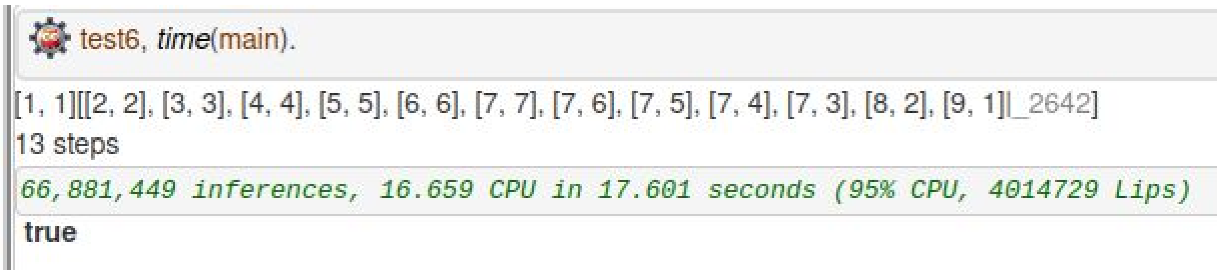


```
test5, time(main).
[[1, 1], [2, 2], [3, 3], [4, 4], [5, 4], [6, 4], [7, 3], [8, 2], [9, 2]]
9 steps
118,418 inferences, 0.126 CPU in 0.126 seconds (100% CPU, 941058 Lips)
true
```

Backtracking

6)

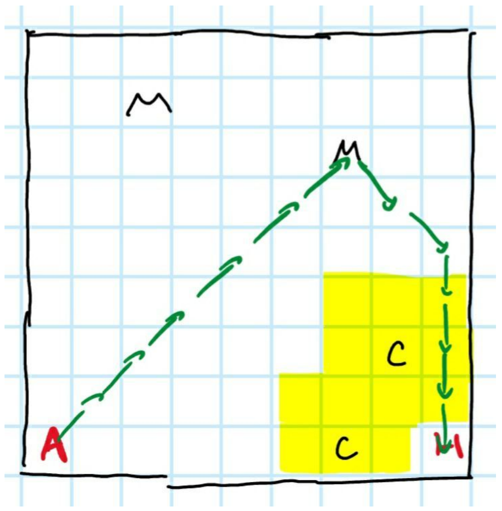


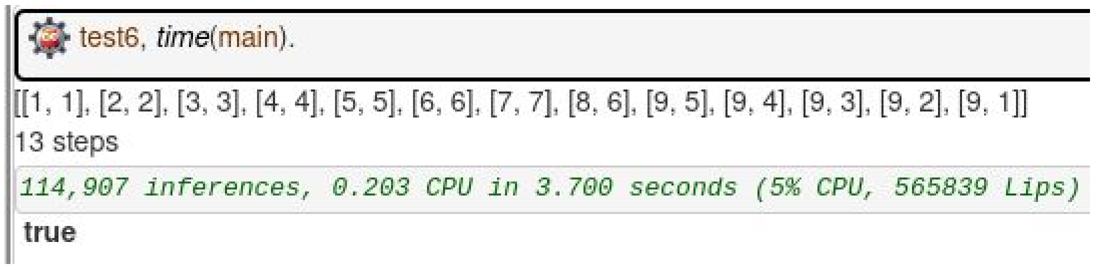


```
test6, time(main).
[1, 1][[2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [7, 6], [7, 5], [7, 4], [7, 3], [8, 2], [9, 1]|_2642]
13 steps
66,881,449 inferences, 16.659 CPU in 17.601 seconds (95% CPU, 4014729 Lips)
true
```

Dijkstra

6)





```
test6, time(main).
[[1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 6], [9, 5], [9, 4], [9, 3], [9, 2], [9, 1]]
13 steps
114,907 inferences, 0.203 CPU in 3.700 seconds (5% CPU, 565839 Lips)
true
```