This project focuses on some of the more advanced search algorithms and how we can use them to solve shortest path problems.

Problem description:

We have a strange doctor located at the southwest corner of an $n \times m$ table. He wishes to go on a journey and collect every possible potion on the table and then reach the northeast corner. Assuming he can only move one square (up, down, left, right) at each step, you are to calculate the minimum number of steps required for the doctor to reach his goal. Note that the table also contains the following items:

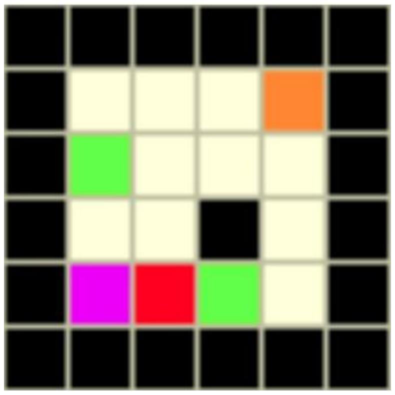
1- Black squares: He can't pass through these squares
2- Green squares: These squares represent the available potions' locations
3- Red squares: These squares contain a medicine each! if he passes through these squares, he will drink that medicine and a new doctor will enter the table from the northwest corner
4- Pink square: Starting point of the doctor, which is always the southwest corner
5- Orange square: The destination square, i.e., the northeast corner

Please take note that at each step, <u>only one</u> doctor can make a move.

Input:

In the first line, the dimensions of the table are given. Then in the second line you'll be given two inputs c, k which represent the number of potions and medicines respectively. Then in the next c lines you'll be given the coordinates of the potions (x, y) and on the next k lines you'll be given the coordinates of the medicines as well. You'll be given the number of black squares (d) in the upcoming line and then finally, d line of coordinates will follow.

This is the representative square for input1:



Output:

The minimum number of steps for the doctors to catch all the potions and reach the northeast corner.

Approach:

We need to simulate the environment in our program. We know our environment has the following specifications:

- Partially observable
- Deterministic
- Sequential
- Static
- Discrete
- Single-agent (or multi-agent)
- Known

The most important thing we need to take care of, is the difference between each state of the time while the doctors are moving. To do so, we represent the table at a specific time using a class named *Setup*. This class differs from its neighbors by the amount and the locations of its doctors and potions. We do not care for the medicines because we are not required to collect them. The initial state, is the respective Setup board built from the input. Any Setup board with its doctors all located on the northeast corner and its medicines all picked up, can be a nominee for our goal state. As mentioned, at each step, we can take a Setup board, move one of its doctors and then apply the necessary changes. We've implemented this approach using three different algorithms.

Algorithms:

i. BFS: In this algorithm, we simply build every Setup table possible until we reach our goal state. In each step, we take all of the remaining boards from the previous step, and move one doctor in each of them respectively. This will generate a whole new batch of Setups which we'll use in the next step. Because we need to find the minimum number of steps to reach our goal state, we can perform a BFS operation on these boards to reach the goal state from the initial board using the least number of actions.
ii. IDS: In this algorithm, we perform a limited-length DFS operation on our initial board, until we find a length and a path to satisfy our goal. We put

limitation on our DFS algorithm to find the satisfiable path faster and to avoid calculation of repetitive paths.

iii. A*: To perform the A* algorithm, we need a heuristic function. We also need the heuristics to be consistent for our algorithm to be optimal. The following function can be a good example: $heuristic(board) = \sum_{\text{doctor}}^{\text{doctors}} min\left(distance(doctor, goal), min\big(distance(doctor, medicines)\big)\right)$

we add the original cost to reach each board to its heuristic value and then we can multiply the result in some const α (to reach the goal faster, in expense of losing optimality).

$$h(board) = \big(heuristic(board) + cost(board)\big) \times \alpha$$

$$\alpha = \ 1\ or\ 2\ or\ \ldots$$

The heuristic function is consistent, and it is fairly easy to prove it. We just need to show that for two different neighbors A and B:

$$Cost(A\ to\ B) \geq heuristic(A) - heuristic(B)$$

We know $Cost(A\ to\ B) = 1$ because they are neighbors and only one doctor (resulting in one cost) can move at each step, so we only look for changes made by the said doctor (which we'll call Dr. Rogers from now on). Whether Dr. Rogers has moved in a direction closer or further from medicines or goal, doesn't matter because he can change the heuristic function by one at most, so the absolute value of $heuristic(A) - heuristic(B)$ is less than $Cost(A\ to\ B)$. Another perspective to look at this matter is that the cost implied by the heuristic function is far less then the actual cost. This is because we've used the minimum distance between objects in our function which may not hold true in the actual settings.

Pros vs Cons:

All of the above algorithms can be efficient to calculate the result. The A* algorithm solves the problem faster than the other two and is, generally, better to use. However, we need to be careful when we're working with heuristics. As mentioned, if we want our solution to be optimal, we cannot use const α have inconsistency in our heuristics. The BFS and IDS algorithms are very much alike.

The BFS algorithm tends to solve the problem quicklier, but the IDS algorithm can be implemented in a simpler way. BFS and IDS both generate optimal solutions.

These are the following results of the mentioned algorithms:

Input1	Minimum Cost	Processed States	Avg Execution Time
BFS	11	2170	0.103s
IDS	11	4015	0.091s
A*	11	955	0.060s
Weighted A*, α=1.5	11	955	0.054s
Weighted A*, α=33	11	955	0.054s

Result for input1

Input2	Minimum Cost	Processed States	Avg Execution Time
BFS	11	170289	12.163s
IDS	11	79886	3.725s
A*	11	7180	0.604s
Weighted A*, α=1.5	11	7180	0.607s
Weighted A*, α=33	11	7180	0.612s

Result for input2

Input3	Minimum Cost	Processed States	Avg Execution Time
BFS	19	332100	22.950s
IDS	19	258641	9.961s
A*	19	34620	2.746s
Weighted A*, α=1.5	19	34620	2.843s
Weighted A*, α=3	19	34620	2.813s

Result for input3

Note that the slight linear difference between BFS and IDS execution time is due to their implementation. Generally, the BFS algorithm should be a bit faster. However, because of the python *deepcopy* methods used in its implementation, it tends to solve the problems a bit slower. If one were to use the same methods in IDS algorithm, you wouldn't even get the results of the second and third input after more than five minutes! Similarly, if you implement the BFS algorithm by the duplication methods used for the IDS algorithm, you would get much better results.

Conclusion:

We've now learned to use search algorithms such as BFS, IDS, and A* to find the shortest path to satisfy the problem's conditions. The most significant output of this assignment was the noticeable time difference between A* algorithm and the other two. Because A* algorithm chooses the states based on a measured guess, it will most likely reach the goal state faster and thereby has a better execution time.