Homework AI

Volintiru Mihai Catalin - 343C3

TASK 1

A* heuristic explanation

In order to find an admissible heuristic for my algorithm, I concluded that taking all the colors and counting only those that are not on the right face and dividing the result by 8 is heuristic because:

- For one move, there are only 8 colors misplaced, so the result of the heuristic will be 8 / 8 = 1 <= 1, so far so good
- For 2 moves, there is a total of 36 possible combinations of 2 moves for the cube. For all these values, the number of misplaced cubes varies between 0, 8 and 14 (I tried them by hand and counted them 😊). So, the maximum number of colors misplaces is 14, divided by 8 is 14 / 8 <= 2, which is good, keeping in mind that the real cost is 2 and the result of the heuristic is less than two, so no overshooting here
- For 3 or more moves it's simpler. Suppose that all the colors of the cube are misplaced, that sums it up to 24 misplaced colors. Dividing it by 8 will result in 24 / 8 = 3 <= N, which is again correct, for N that goes from 3 to any number of moves.

The reason I divide it by 8 is because I "misplace" 8 colors from a face after one move. There are 4 colors that are moved around, but they remain on the same face. For example, from the goal state, the R move will misplace 8 colors, but the colors on the left face will remain on the same face (in this case, all the red colors will remain on the red face).

A* vs Bidirectional BFS

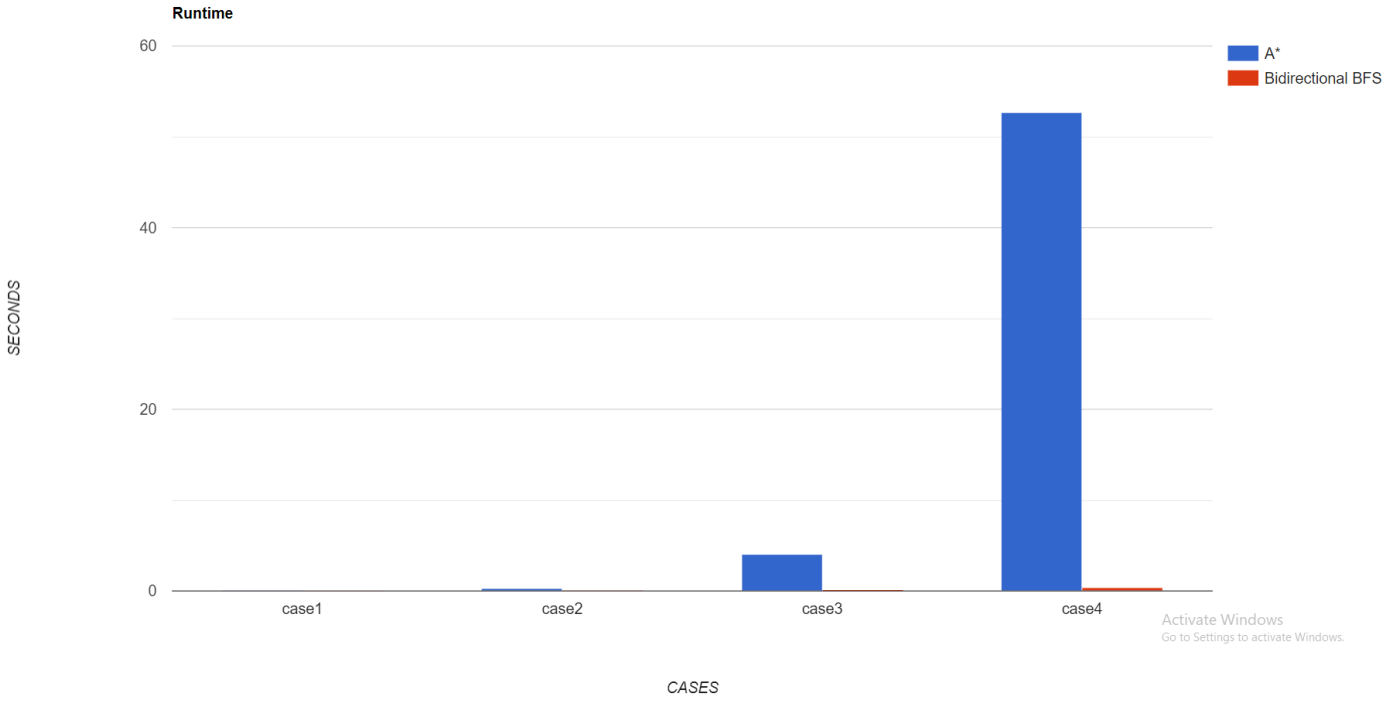
Bidirectional BFS is by far the best algorithm to solve the Pocket cube problem, whether we talk about time complexity or space complexity.

I would like to add that the bar charts that I made have values that are quite hard to see, so I decided to list the values obtained by me down below.

Runtime

We will first compare the values for the **runtimes**. Because some of the values are too small, the graph is not so helpful, so I will list them here:

- case1 is solved in 19.4 ms by A* and 6.99 ms by Bidirectional BFS
- case1 is solved in 249 ms by A* and 39.4 ms by Bidirectional BFS
- case1 is solved in 4.03 s by A* and 126 ms by Bidirectional BFS
- case1 is solved in 52.7 s by A* and 379 ms by Bidirectional BFS



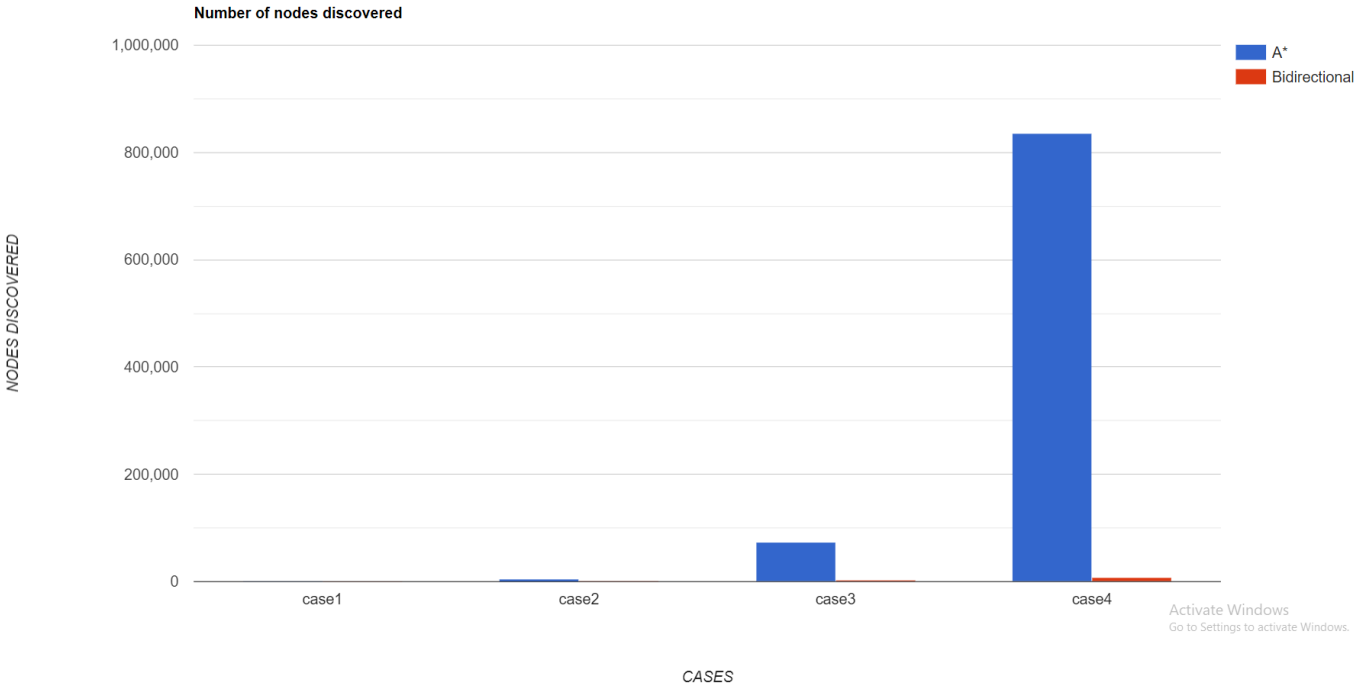
Time complexity of the A* algorithm, even with a good heuristic will be $\mathbf{O(b^d)}$, while the complexity of Bidirectional BFS will be $\mathbf{O(b^{d/2} + b^{d/2})}$, where b is the branching factor and d is the depth of the closest solution.

The branching factor is 6, because from a given state, we can go into maximum 6 new states (for every move available), and d is 14, because Pocket Cube can always be solved in maximum 14 moves.

Number of nodes discovered

For every algorithm, the values of the discovered nodes are as it follows:

- For case1 A* discovers 374 nodes while Bidirectional BFS discovers 110
- For case1 A* discovers 5187 nodes while Bidirectional BFS discovers 782
- For case1 A* discovers 73309 nodes while Bidirectional BFS discovers 2719
- For case1 A* discovers 836285 nodes while Bidirectional BFS discovers 8303



Bidirectional BFS is much faster than A* and much more efficient with the **space complexity**, given the fact that in all the cases, Bidirectional BFS algorithm **discovers a number of nodes** smaller than the latter algorithm does. This is because A* is forced to go through more states based on the heuristic score (which can be pretty slow), while Bidirectional BFS searches for a path from both ends, discovering a solution much faster, thus decreasing the number of discovered states.

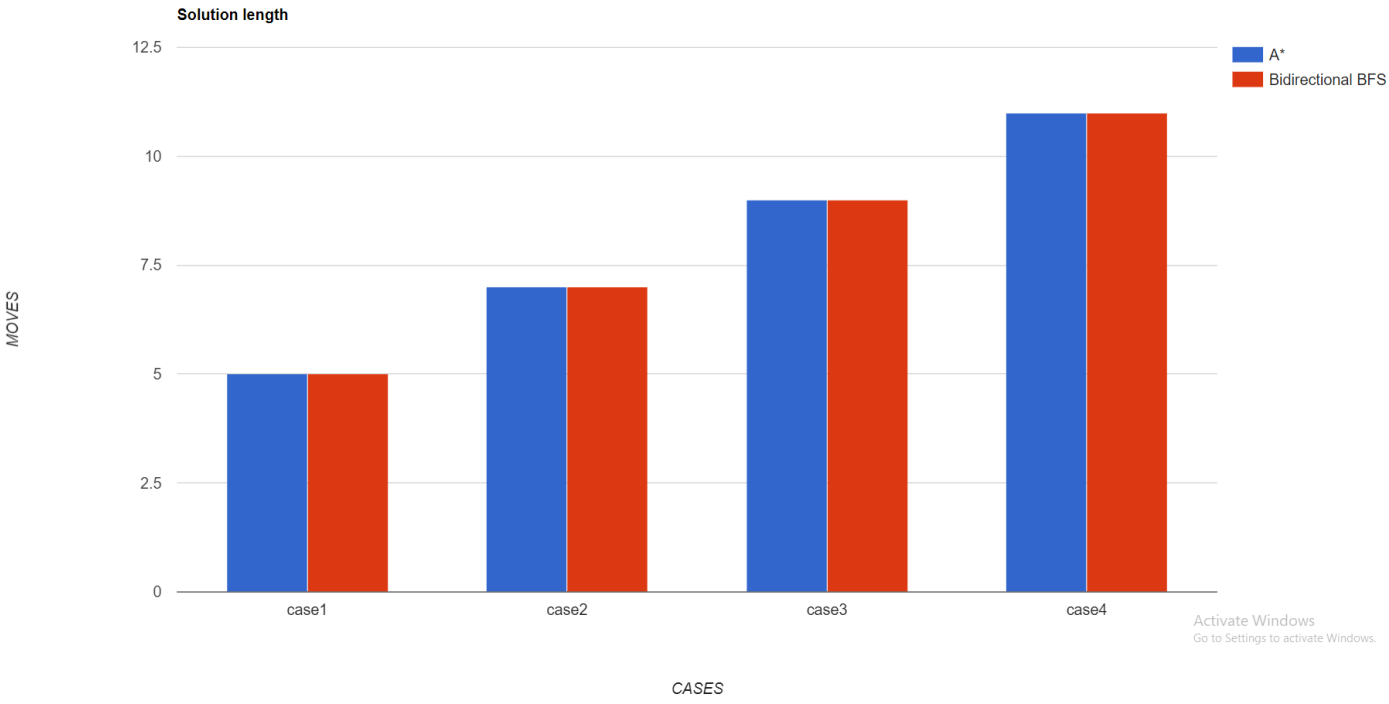
The results can't be ignored, for the last case the second algorithm discovers a value of nodes that is **100** times smaller than the first algorithm does.

Length of the solution

Both algorithms discover a **solution** with the same **length**, given the same scrambled cube, so we can say that both algorithms are equal, for this case. The solution is also optimal because:

- for A* we have an admissible and monotonic heuristic function

- for Bidirectional BFS, the costs of the paths are uniform (in my case, I did not bother to add a cost function for the nodes; I add the new states in a queue and that is it, so we can say that all the paths have the same cost), thus returning the optimal solution



How can we observe the fact that the solutions are optimal? We can look at the number of moves it takes to scramble a cube. From here, we can deduce that it takes the same amount of moves to solve the cube (the moves are the reverse of the moves used to scramble a cube, starting with the last move to the first one).

TASK 2

Monte Carlo Tree Search

In order to implement the algorithm, I used the implementation from the laboratory. The heuristic used in A* gave me the **distance** from the current state to the goal state, so in order to be sure that the algorithm will return a **reward**, I took the maximum value that can be offered by the heuristic function and subtracted from it the value of the heuristic for a given state. In this manner, for a state that has the value for the heuristic 3, the reward will

be 0, a very low reward. For the goal state, the reward will be 3, the highest possible reward.

Second heuristic for MCTS

For the second heuristic, I chose a heuristic that is not admissible, to see a different way of approaching the problem.

The heuristic is the result of the Manhattan Distance between where a corner is and the actual correct position where the corner is supposed to be. This heuristic is not admissible because it does not divide the result by 8, leading to often overshooting the correct distance from the current state to the goal state.

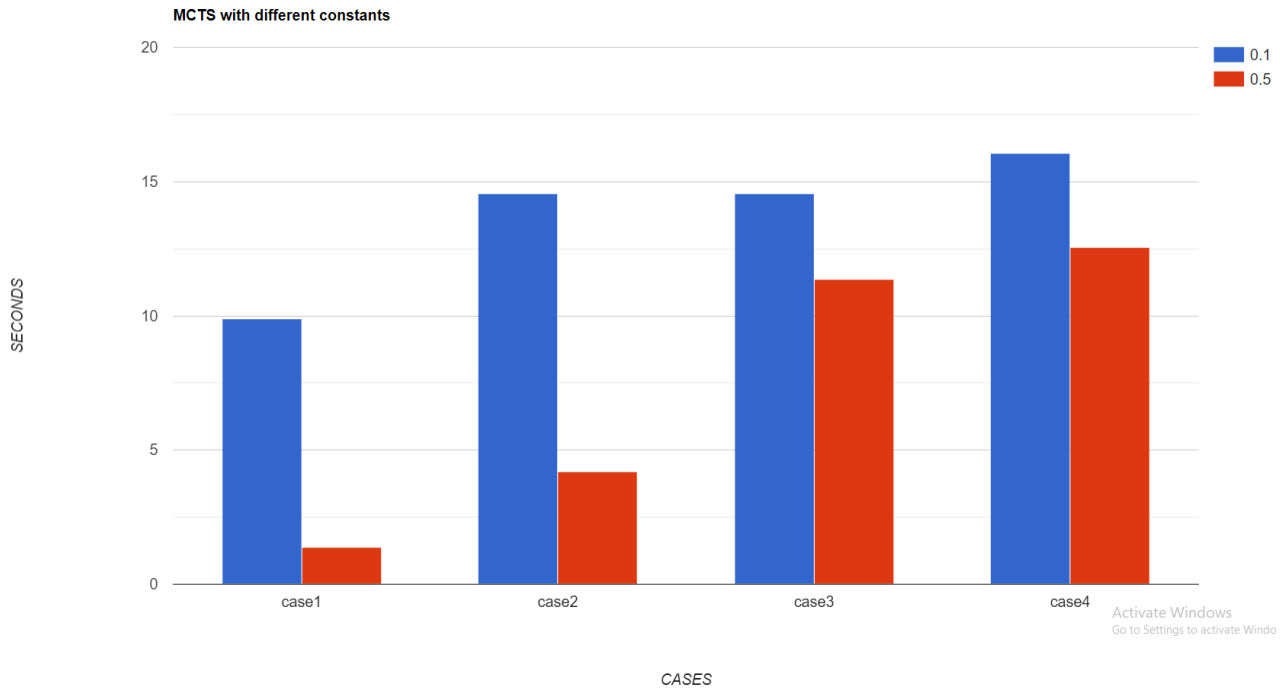
To make it a **reward**, I took the maximum value that can be offered by the heuristic function and subtracted from it the value of the heuristic for a given state. In this manner, for a state that has the value for the heuristic 24 (all the corners are 3 moves away from their correct position, though I don't really know if it is possible), the result will be 0, the lowest reward. For the goal state, the reward will be 3, the highest possible reward.

MCTS best options

In order to be able to compare this algorithm, I ran all the possible combinations (values of c being 0,5 or 0,1, values of budget being 1000, 5000, 10000, 20000, and both heuristics). It was a pretty long endeavor, given the fact that the algorithm did not always return a solution, especially for the last test case, because there are a lot of states to search for and the random choice function can't simply find the correct path. For the second heuristic, I chose a non-admissible one.

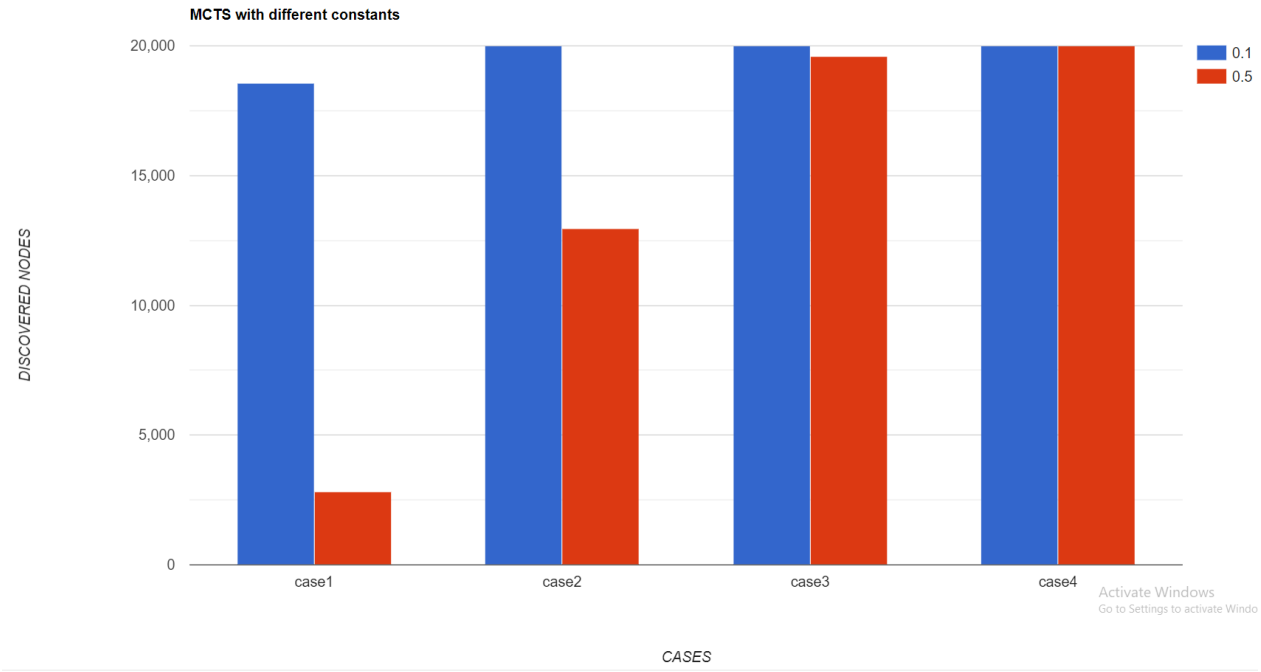
While comparing all the combinations, the best option I found is using the admissible heuristic I implemented for A* algorithm, the constant equal to 0,5, and the budget 20000.

First, let's compare the algorithm with different constants, but the same heuristic (the admissible heuristic) and the same budget (20000)



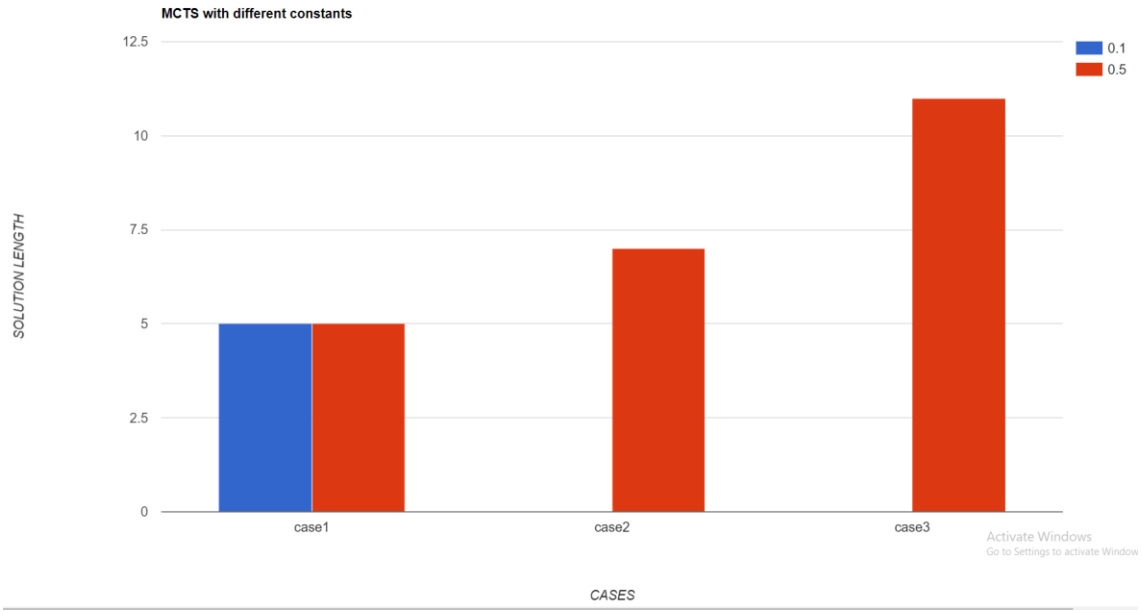
We can see that with the constant equal to 0,5, the algorithm is much faster compared to constant equal to 0,1.

Now let's compare the number of nodes discovered.



Overall, with c = 0,5, the number of discovered states is smaller. Since both of the cases could not provide an answer to case 4, they discovered as much as the budget allowed them, so they are equal in this case.

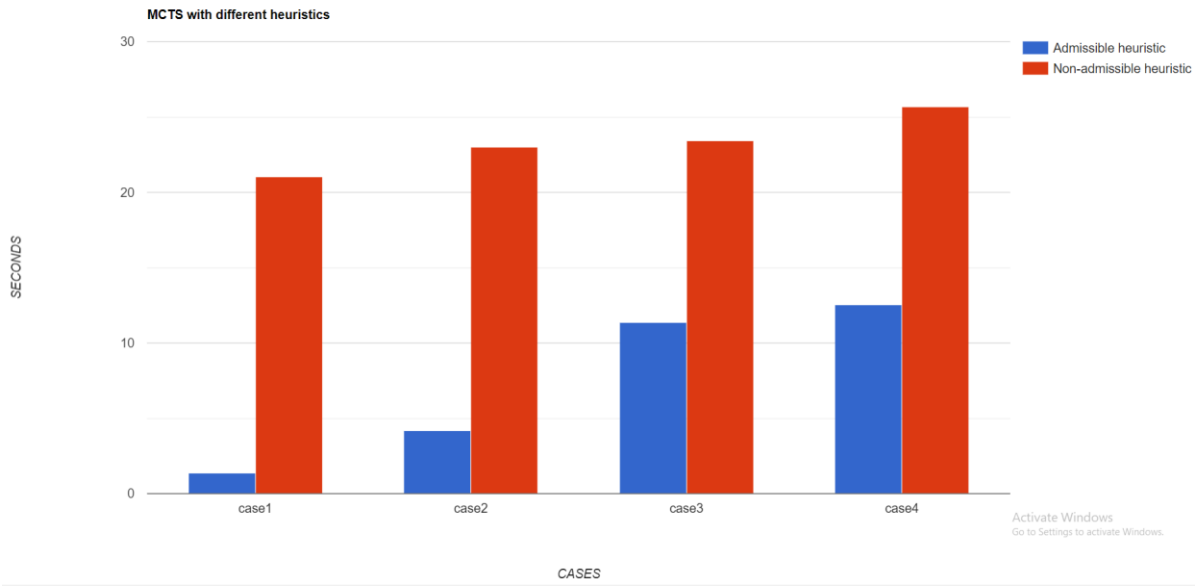
Now, let's compare the length of the solution.



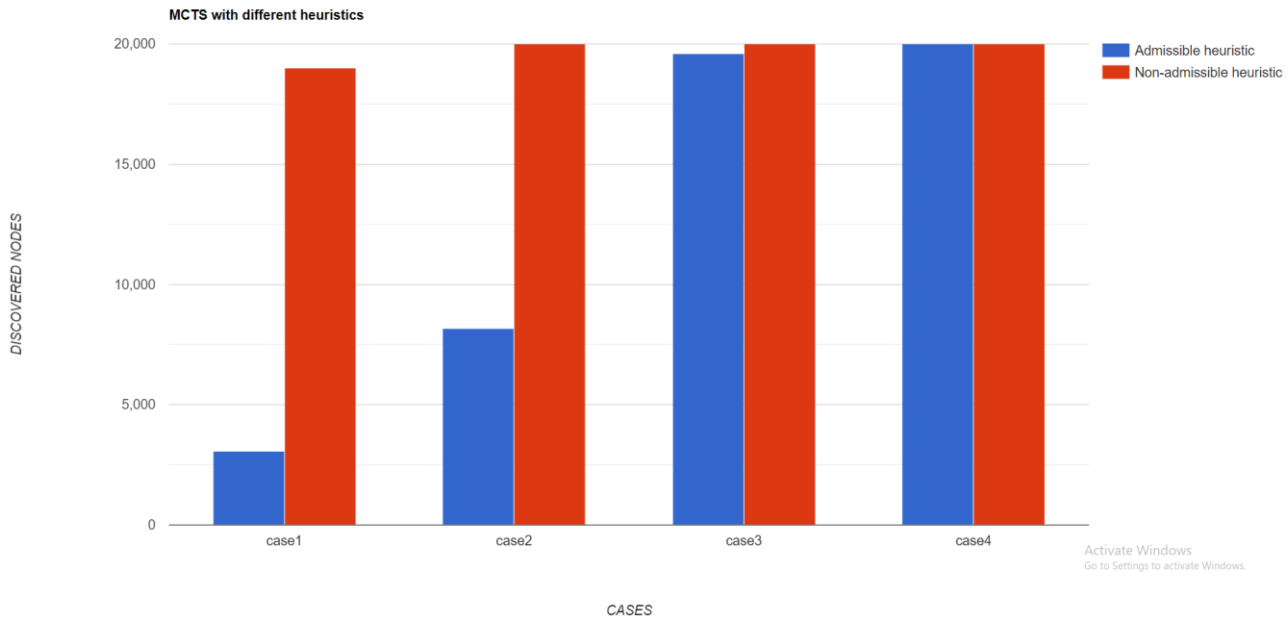
Overall, using the constant as 0.1 yields an inferior result compared to using c = 0,5. We will proceed to use the constant equal to 0,5.

Now, we will compare the heuristics used to solve the implementation. To even the chances, we will use c = 0,5, and budget = 20000. First, let's compare the runtime:

Overall, using the non-admissible heuristic yields a bigger runtime. Now let's compare the states discovered:

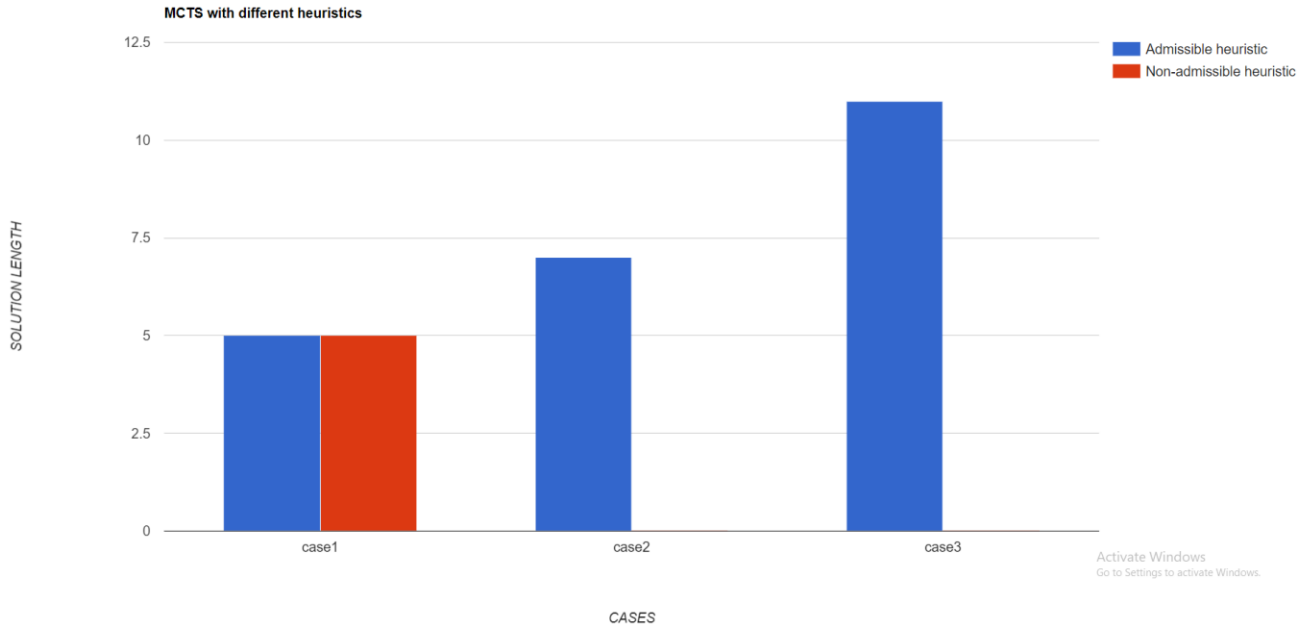


Now we will analyze the number of discovered nodes:



As we can see, the non-admissible heuristic performs pretty bad in comparison with the admissible one. For case 3, non-admissible Manhattan does not yield a result, so the man number of states that it discovers is the maximum budget, but the first heuristic manages to find only one solution in 11588 steps, so the mean will be 19579 nodes discovered, quite a lot of them.

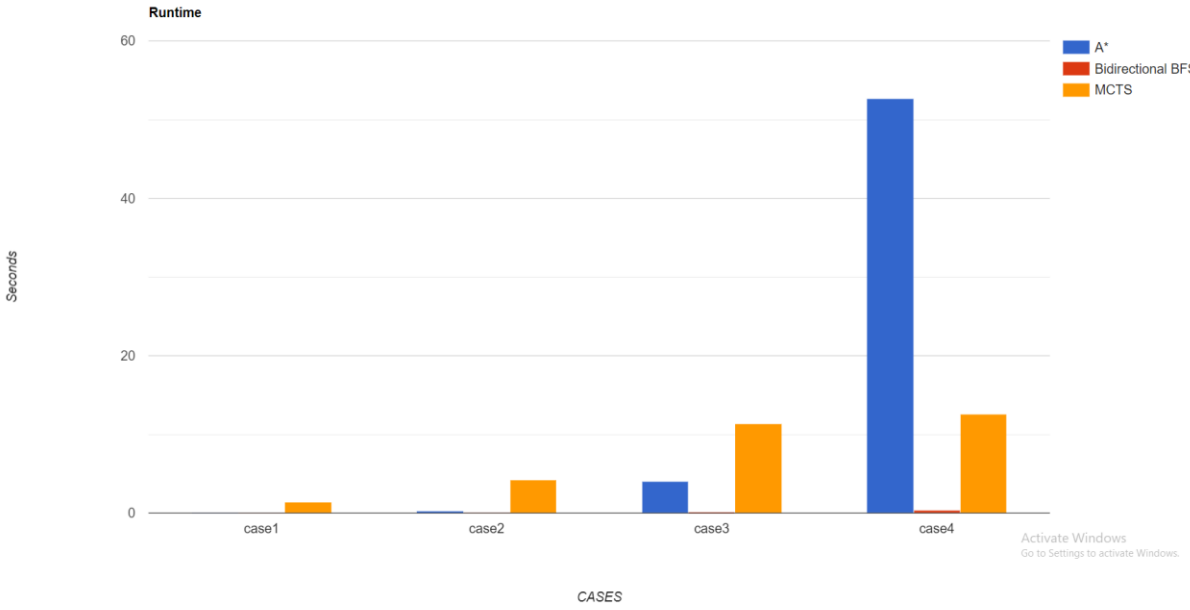
For the solution length, both cases could not find a solution for the last test, and also, the non-admissible heuristic could only find the solution for the first test:

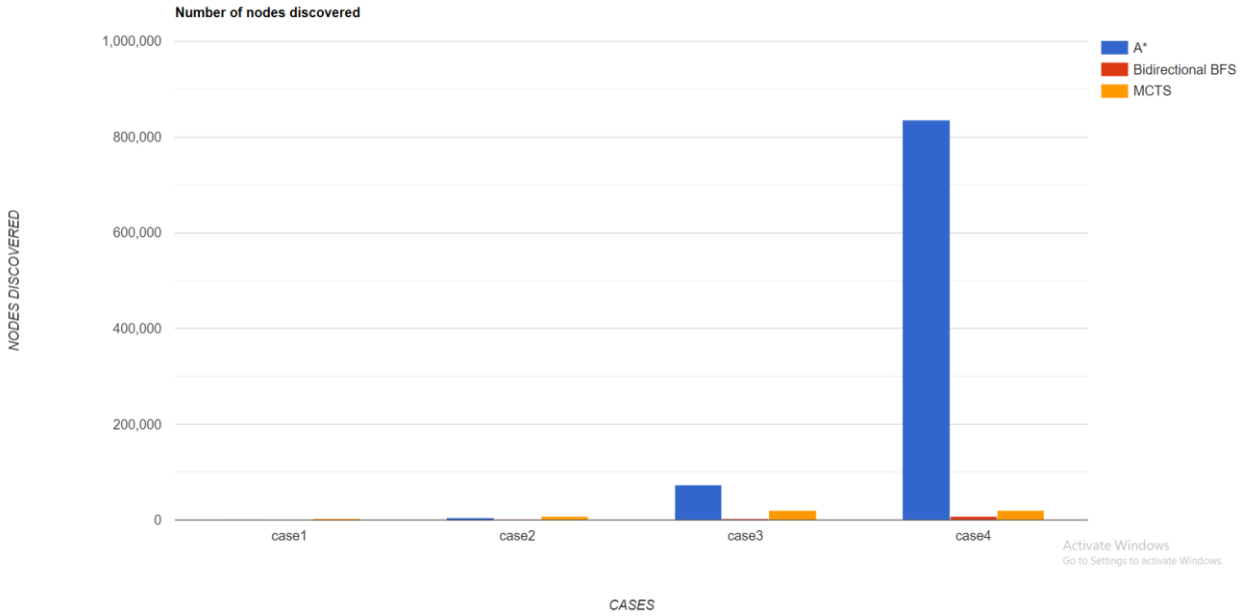


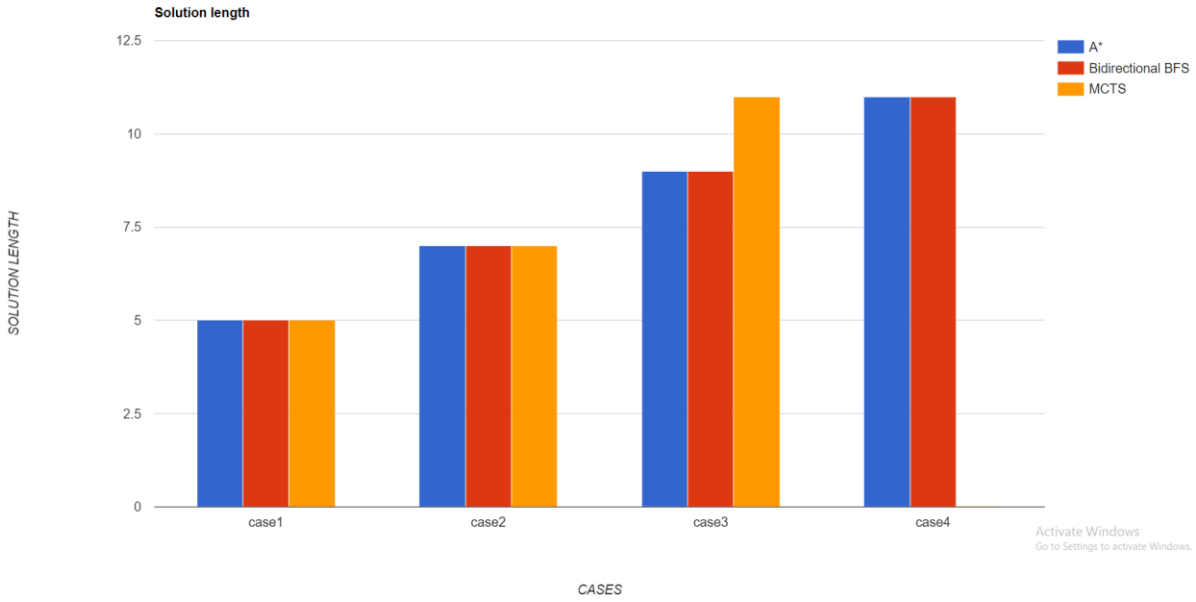
I don’t think there is a need for me to draw bar charts in order to compare the values of the budget because the bigger the budget, the bigger the chance of finding a solution for a case. If the budget is too small, there is a chance that the algorithm will not even find a solution for the first test case. However, 20000 is yet too small in order for my algorithm to even find a solution for the last test case.

In conclusion, based on what I presented above, the best combination of factors is: c = 0,5, budget = 20000 and admissible heuristic used in A*.

Let us now compare the results of MCTS with A* and Bidirectional BFS:







From the graphs above, we can see that MTCS is “faster” for more complex states, but it is not guaranteed to return the cube **solved**! This is a huge disadvantage, because we will end up running this algorithm a huge number of times in order to get an answer, proving that its runtime advantage is very uncertain. It is possible to find the solution, and even solve it under 20000 states, but how many times will it take for it to run until we find it?

For the number of nodes discovered, the advantage of MTCS is that we can adjust the budget, the maximum states discovered being the number represented by the budget, but again, very uncertain. We can’t guarantee that it will find it, so we’re going to have to run it multiple times.

Regarding the solution length (if it finds the solution), for MTCS, it will vary. The solution is not guaranteed to be optimal. For case 4, the solution length is inexistent, because the algorithm can’t find it.

In conclusion, MCST algorithm may have a huge advantage regarding the number of states discovered and the runtime, but the downside is that it does not guarantee a solution, so we’re going to end up running it multiple times in order to obtain an answer. A* and Bidirectional BFS are far better because these algorithms will always provide an answer/solution.

TASK 3

Pattern Database generation

In order to generate all the possible unique nodes, I used a dummy cube and made an algorithm that resembles BFS. This way, I start from the goal state and add all the neighbors into a queue, from where I will visit them one by one. If the depth is greater than 8 or the state was already discovered, the algorithm will skip the iteration. The time the algorithm takes to generate all the nodes is quite good, 10.1 seconds for 44971 states.

To check if the number of states was correct, I found a site that helped me count all the possible states that are at a given distance: http://anttila.ca/michael/devilsalgorithm/

Heuristic structure

The heuristic takes a parameter, the cube, and checks if the state is already in the "discovered" dictionary. If it finds the state, it will return the exact distance to the solved state, if it does not, it will return the value of the first heuristic implemented for A* algorithm that I explained above, in TASK1.

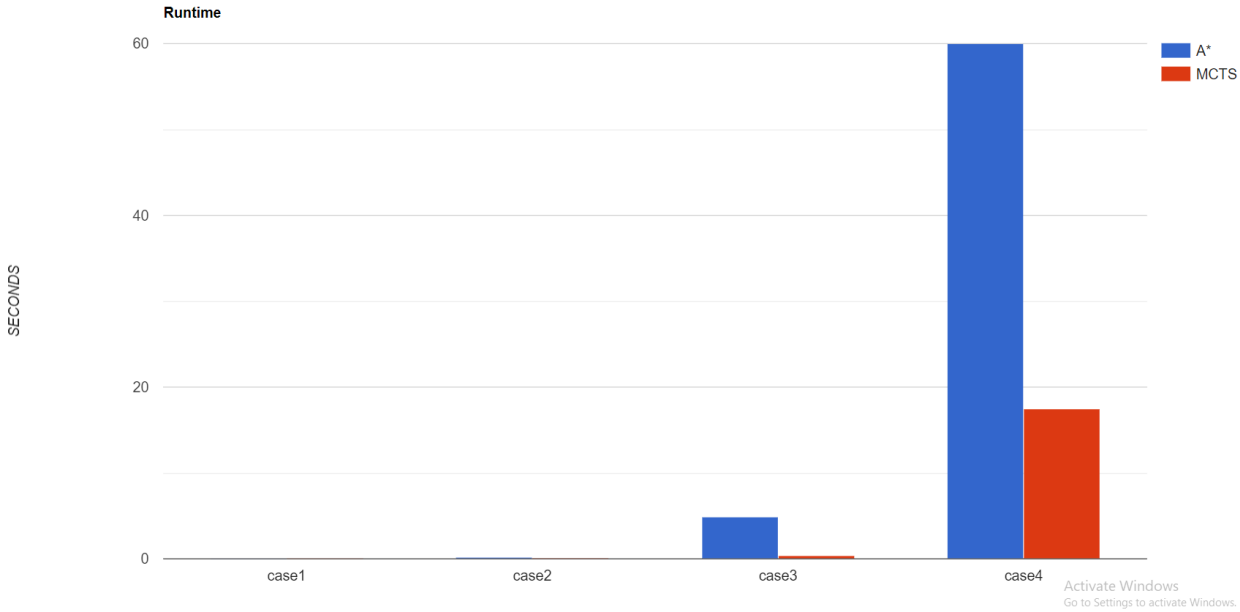
A* vs MCTS using the pattern database heuristic

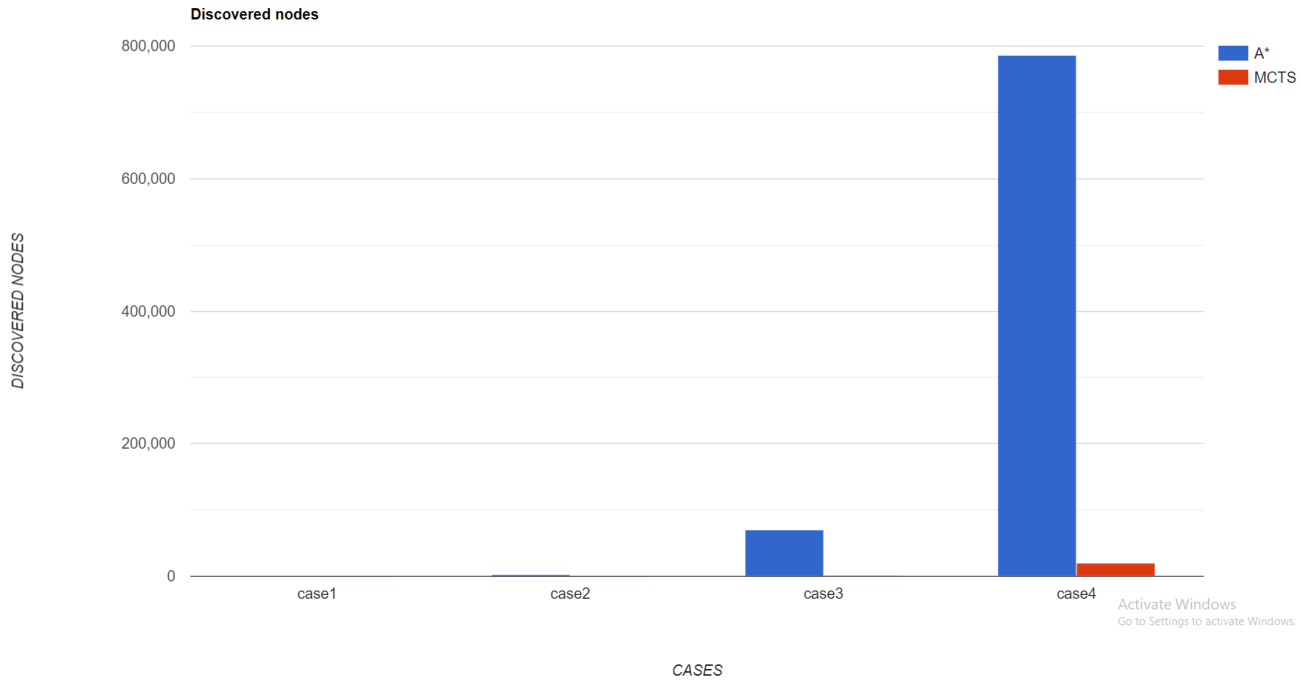
First, we will analyze the runtime, the number of nodes discovered, and in the end, the solution length.

We can see that, overall, A* is faster than MCTS, but again, MCTS does not yield a solution for the last test case, which makes it unreliable. If MCTS **does** actually find the result by luck, it will find it in a much faster time, this is why, for case 4, the second algorithm has the potential to do better.

The values for the first two test (because we can't really see them on the bar chart) are:

- 1,99ms for case1 and 153ms for case2 (A* algorithm)
- 25,95ms for case 1 and 60ms for case2 (MCTS)



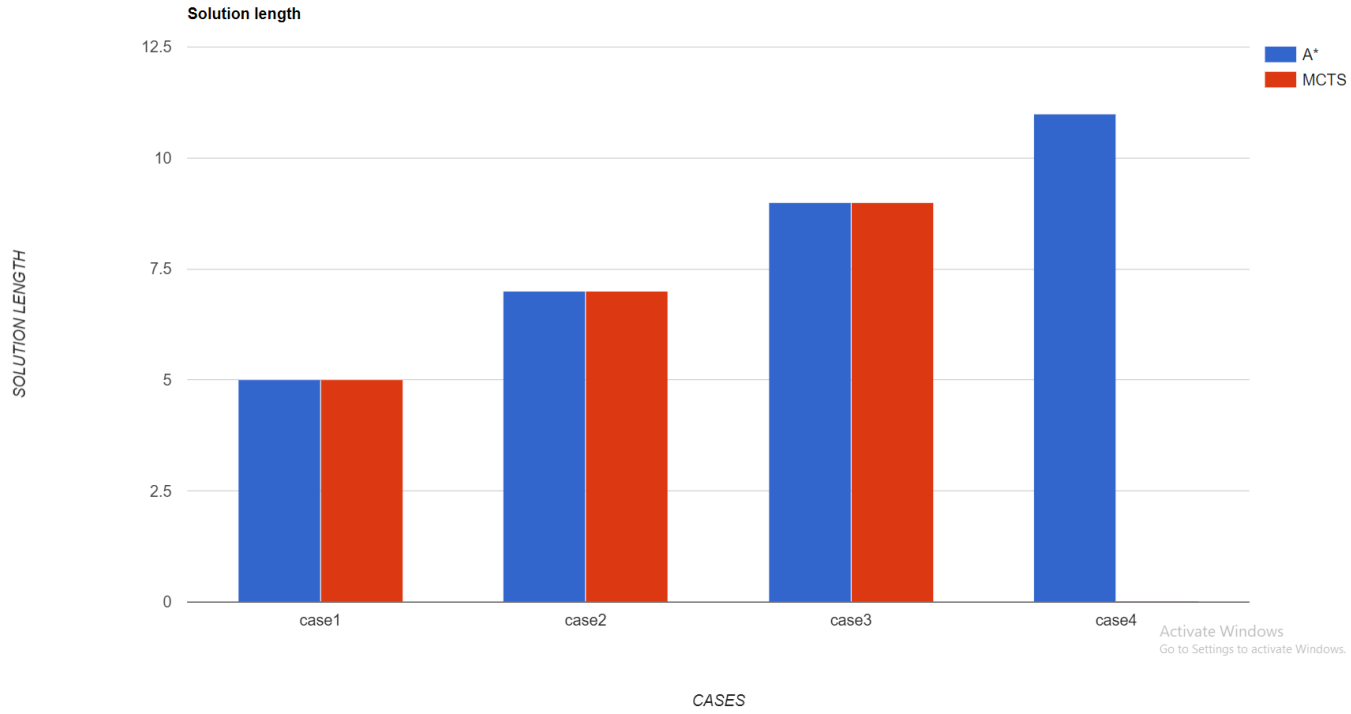


Again, it is pretty hard to see the values for the first 3 tests, so I will list them below:

- 27 case1, 2400 case2, 69962 case3, 786294 case4 (A*)
- 34 case1, 89 case2, 581 case3, 20000 case4 (MCTS)

The number of states discovered shows that, overall, for less complex problems, A* will find the solution (because, for those states, the distances from the goal state are less or equal to the maximum distance of a state inside the “discovered” dictionary that we made in order to help us with the pattern databases heuristic). For more complex states with a distance from goal bigger than 7, the A* will begin to search for better nodes using the

admissible heuristic that is very slow, this is why MCTS will prevail, because it searches for maximum budget of 20000, but again, it is not guaranteed to yield any solutions.



From our last bar chart, we can observe that apart from case 4, MCTS did generate the optimal solution. For case 4, the algorithm did not generate anything.

In conclusion, A* is far better than MCTS because it guarantees a solution and is acceptable when talking about speed.

This new heuristic improved the runtime of A* (except case 4, the previous version took 8 seconds less to find the solution) and decreased the number of states that A* discovered (for case 4 from 836285 to 786294). This is due to the fact that once the algorithm found a state that is in the “discovered” dictionary, it will know the **exact** distance from the goal state, thus being more accurate, and it will search for more accurate states to continue.