Artificial Intelligence
Adversarial Search

Fall 2024

Parsa Darban 810100141

This file contains the report and results of the simulations conducted.

Abstract

In this project, we explore search algorithms.

As we know, these algorithms are categorized into two types: informed and uninformed. In this project, we examine each of the mentioned methods, such as BFS, IDS, A*, and Weighted A*.

Breadth First Search (BFS)

To organize a wedding ceremony, we need to turn off all the lights with the minimum number of switches. The mathematical representation of this problem involves an n×nn \times n matrix containing values of 0 and 1. Each cell represents a table with a lamp, where a value of 1 indicates the lamp is on, and a value of 0 indicates it is off. Our goal is to input an initial matrix with nmax=4 and change the states of its elements to turn all values into 0.

Here, we analyze this problem using the BFS (Breadth First Search) algorithm, which is an uninformed search algorithm.

So we have:

Initial state	The matrix we start with
Goal	Turning all elements of the matrix to zero
Actions	Changing the state of each cell and its neighbors
Path cost	The cost of changing a state is considered uniform

Code Summary

Import: First, we import the required libraries.

LightsOutPuzzle: This class handles toggling the lights on and off and checks if the "all off" state has been achieved. Additionally, it generates the cells for each table, structured in a way that helps identify which cells are toggled during the process.

$$\begin{pmatrix} (0,0) & \cdots & (0,n-1) \\ \vdots & \ddots & \vdots \\ (n-1,0) & \cdots & (n-1,n-1) \end{pmatrix}$$

board: This function generates a matrix as the initial condition for starting the toggling process. It creates an all-zero matrix and randomly modifies it (or adjusts it to a desired size) to produce an initial state.

tests: This function defines the size and inputs for create_random_board to test the algorithms.

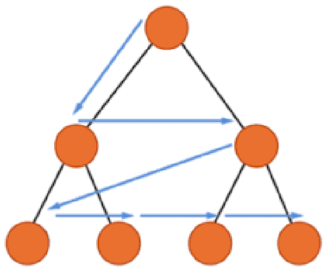
Final functions: Four additional functions are implemented to display the final results in a structured and interpretable format.

Now, let’s delve into the BFS algorithm. The core principle of BFS is to start from an initial state. After evaluating this state, the algorithm explores its children (resulting from applying different actions), followed by the children of those states, and so on. This process examines all states at a given depth before proceeding to the next level of depth.

One of the key features of BFS is its use of two sets:

frontier: A set to add new states at the current depth for exploration.

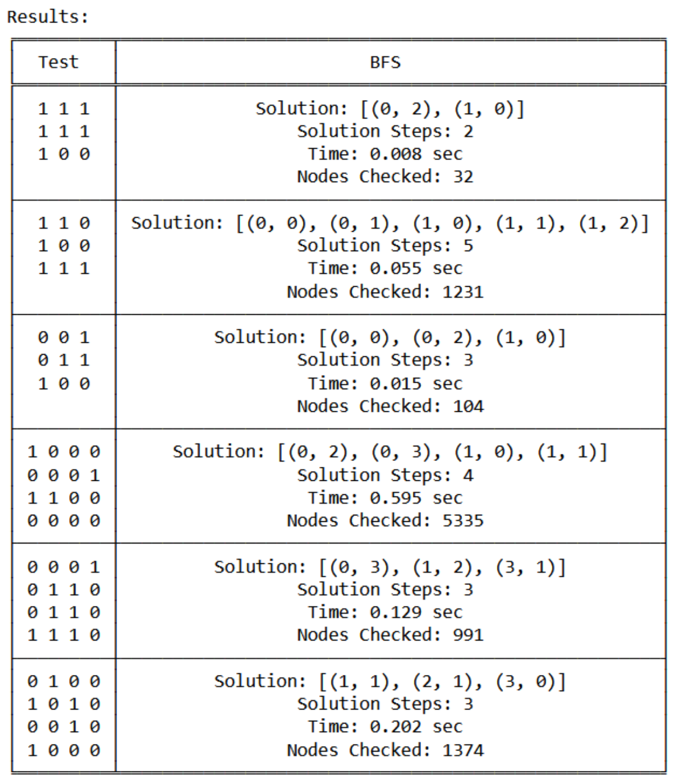
explored: A set to store all evaluated states to avoid redundant evaluations.



Now, let's move on to the code. The main idea for the coding is as explained. First, we create two sets: frontier and explored. It should be noted that the algorithm is FIFO, so we need to remove the first element (the leftmost element) from the frontier using popleft. Then, in the code, we check each state against explored and add it if it's not already in it.

Explanation of board_tuple = tuple(map(tuple, current_board)): This line is added because we cannot directly compare arrays and matrices, so it converts the current state to a tuple of tuples to allow comparison with the explored set.

The result of this algorithm is as follows:



```
Results:
+---------+------------------------------------------------------+
| Test    |                         BFS                          |
+=========+======================================================+
| 1 1 1   |          Solution: [(0, 2), (1, 0)]                  |
| 1 1 1   |              Solution Steps: 2                       |
| 1 0 0   |               Time: 0.008 sec                        |
|         |              Nodes Checked: 32                       |
+---------+------------------------------------------------------+
| 1 1 0   | Solution: [(0, 0), (0, 1), (1, 0), (1, 1), (1, 2)]   |
| 1 0 0   |              Solution Steps: 5                       |
| 1 1 1   |               Time: 0.055 sec                        |
|         |             Nodes Checked: 1231                      |
+---------+------------------------------------------------------+
| 0 0 1   |       Solution: [(0, 0), (0, 2), (1, 0)]             |
| 0 1 1   |              Solution Steps: 3                       |
| 1 0 0   |               Time: 0.015 sec                        |
|         |              Nodes Checked: 104                      |
+---------+------------------------------------------------------+
| 1 0 0 0 |    Solution: [(0, 2), (0, 3), (1, 0), (1, 1)]        |
| 0 0 0 1 |              Solution Steps: 4                       |
| 1 1 0 0 |               Time: 0.595 sec                        |
| 0 0 0 0 |             Nodes Checked: 5335                      |
+---------+------------------------------------------------------+
| 0 0 0 1 |       Solution: [(0, 3), (1, 2), (3, 1)]             |
| 0 1 1 0 |              Solution Steps: 3                       |
| 0 1 1 0 |               Time: 0.129 sec                        |
| 1 1 1 0 |              Nodes Checked: 991                      |
+---------+------------------------------------------------------+
| 0 1 0 0 |       Solution: [(1, 1), (2, 1), (3, 0)]             |
| 1 0 1 0 |              Solution Steps: 3                       |
| 0 0 1 0 |               Time: 0.202 sec                        |
| 1 0 0 0 |             Nodes Checked: 1374                      |
+---------+------------------------------------------------------+
```

As we can see, increasing input data, such as the matrix size, does not necessarily affect the time it takes to reach the solution.

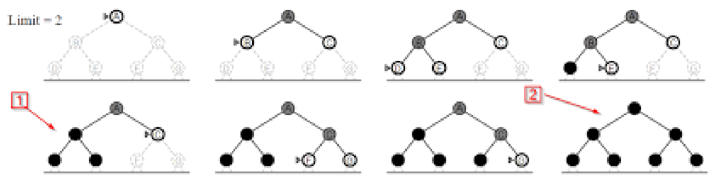
Next, we will examine the IDS algorithm and compare it with these two algorithms.

Iterative Deepening Search (IDS)

In this section, we solve the problem using another uninformed algorithm called IDS (Iterative Deepening Search).

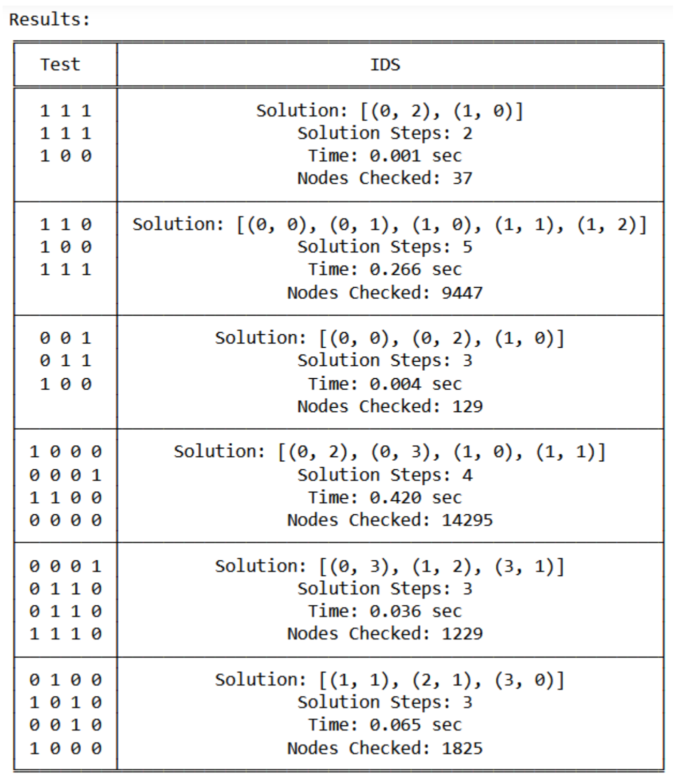
First, we need to familiarize ourselves with the DFS (Depth First Search) algorithm. Unlike BFS, this algorithm reaches the goal by exploring the deepest states first. If we think of the problem's states as a tree structure, DFS first examines all the branches of a given state before moving on to the deepest branch of the tree. As a result, this algorithm is not optimal.

IDS improves DFS by introducing a condition: the search is conducted with a depth limit, and the DFS algorithm is repeated several times with increasing depth. This method doesn't reduce the time needed to find the goal, but it reaches the goal with less memory usage.



Based on the explanations provided, we first need to implement the DFS algorithm. This part is quite similar to the previous section. However, we need to add a depth parameter to allow us to modify the depth and increase it using the ids_solve function. This depth is applied in the depth_limited_search function, which ensures that actions are only applied up to the specified depth. If the goal is not reached, the depth is increased.

The results are as follows:



Results:

Test	IDS

1 1 1
1 1 1
1 0 0
	Solution: [(0, 2), (1, 0)]
	Solution Steps: 2
	Time: 0.001 sec
	Nodes Checked: 37

1 1 0
1 0 0
1 1 1
	Solution: [(0, 0), (0, 1), (1, 0), (1, 1), (1, 2)]
	Solution Steps: 5
	Time: 0.266 sec
	Nodes Checked: 9447

0 0 1
0 1 1
1 0 0
	Solution: [(0, 0), (0, 2), (1, 0)]
	Solution Steps: 3
	Time: 0.004 sec
	Nodes Checked: 129

1 0 0 0
0 0 0 1
1 1 0 0
0 0 0 0
	Solution: [(0, 2), (0, 3), (1, 0), (1, 1)]
	Solution Steps: 4
	Time: 0.420 sec
	Nodes Checked: 14295

0 0 0 1
0 1 1 0
0 1 1 0
1 1 1 0
	Solution: [(0, 3), (1, 2), (3, 1)]
	Solution Steps: 3
	Time: 0.036 sec
	Nodes Checked: 1229

0 1 0 0
1 0 1 0
0 0 1 0
1 0 0 0
	Solution: [(1, 1), (2, 1), (3, 0)]
	Solution Steps: 3
	Time: 0.065 sec
	Nodes Checked: 1825

By comparing these two, we see that due to the presence of repeated states in IDS, the number of nodes checked is higher. However, in the end, their solutions are the same. Regarding the comparison of speed to reach the goal, they are nearly the same, but overall, we cannot conclude that one is faster than the other. However, we do know that:

BFS time : O(b^d)

IDS time : O($b^d \times d$)

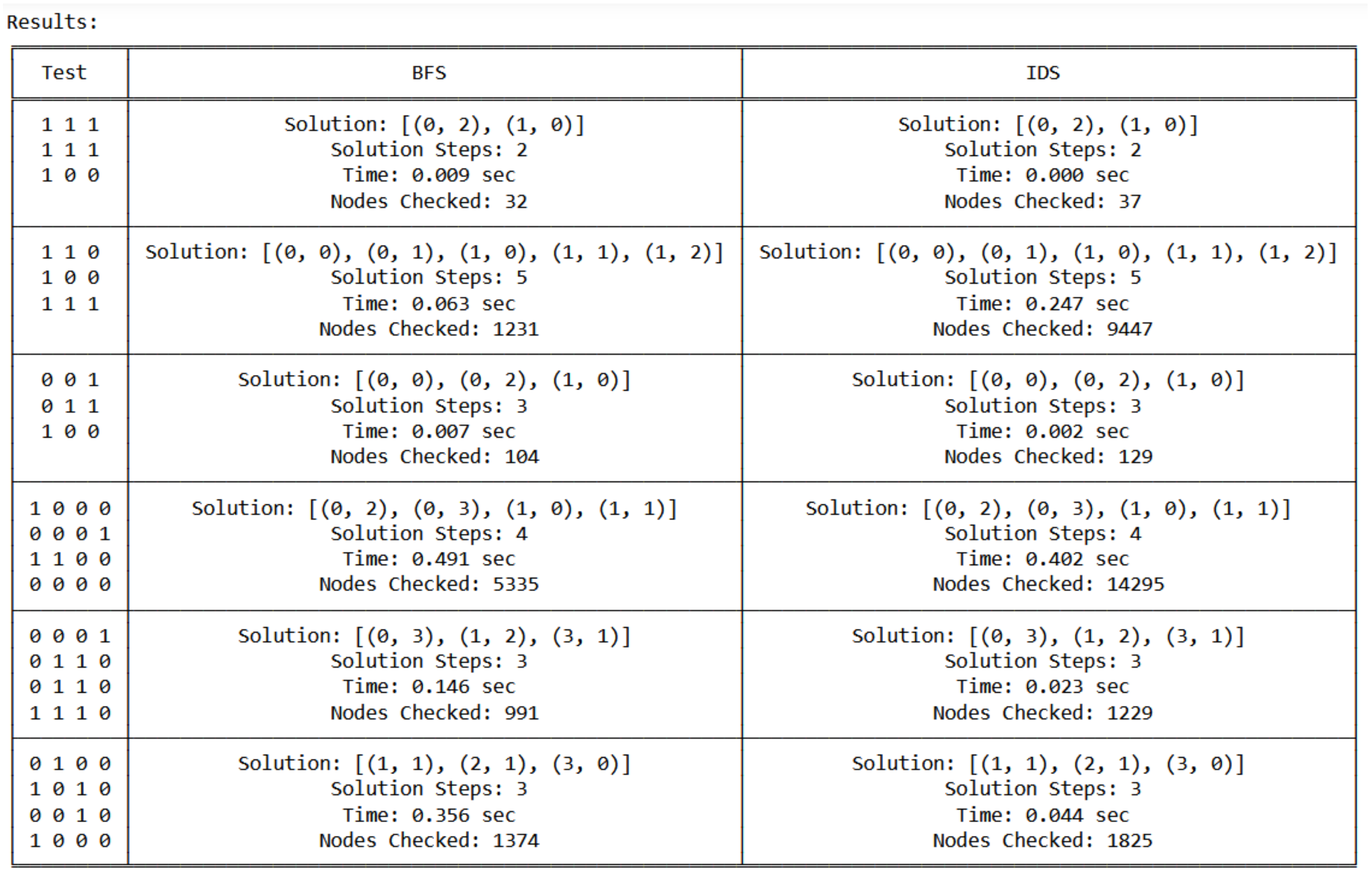
b : branch factors

d : depth of the shallowest goal

Both are also optimal:

BFS: Because the cost of changing states is equal.

IDS: Because it has a depth limitation, it provides an optimal result.



```
Results:
+---------+------------------------------------------------------+------------------------------------------------------+
|  Test   |                         BFS                          |                         IDS                          |
+=========+======================================================+======================================================+
|  1 1 1  |           Solution: [(0, 2), (1, 0)]                 |           Solution: [(0, 2), (1, 0)]                 |
|  1 1 1  |               Solution Steps: 2                      |               Solution Steps: 2                      |
|  1 0 0  |                Time: 0.009 sec                       |                Time: 0.000 sec                       |
|         |               Nodes Checked: 32                      |               Nodes Checked: 37                      |
+---------+------------------------------------------------------+------------------------------------------------------+
|  1 1 0  | Solution: [(0, 0), (0, 1), (1, 0), (1, 1), (1, 2)]   | Solution: [(0, 0), (0, 1), (1, 0), (1, 1), (1, 2)]   |
|  1 0 0  |               Solution Steps: 5                      |               Solution Steps: 5                      |
|  1 1 1  |                Time: 0.063 sec                       |                Time: 0.247 sec                       |
|         |              Nodes Checked: 1231                     |              Nodes Checked: 9447                     |
+---------+------------------------------------------------------+------------------------------------------------------+
|  0 0 1  |       Solution: [(0, 0), (0, 2), (1, 0)]             |       Solution: [(0, 0), (0, 2), (1, 0)]             |
|  0 1 1  |               Solution Steps: 3                      |               Solution Steps: 3                      |
|  1 0 0  |                Time: 0.007 sec                       |                Time: 0.002 sec                       |
|         |              Nodes Checked: 104                      |              Nodes Checked: 129                      |
+---------+------------------------------------------------------+------------------------------------------------------+
| 1 0 0 0 |   Solution: [(0, 2), (0, 3), (1, 0), (1, 1)]         |   Solution: [(0, 2), (0, 3), (1, 0), (1, 1)]         |
| 0 0 0 1 |               Solution Steps: 4                      |               Solution Steps: 4                      |
| 1 1 0 0 |                Time: 0.491 sec                       |                Time: 0.402 sec                       |
| 0 0 0 0 |              Nodes Checked: 5335                     |              Nodes Checked: 14295                    |
+---------+------------------------------------------------------+------------------------------------------------------+
| 0 0 0 1 |       Solution: [(0, 3), (1, 2), (3, 1)]             |       Solution: [(0, 3), (1, 2), (3, 1)]             |
| 0 1 1 0 |               Solution Steps: 3                      |               Solution Steps: 3                      |
| 0 1 1 0 |                Time: 0.146 sec                       |                Time: 0.023 sec                       |
| 1 1 1 0 |              Nodes Checked: 991                      |              Nodes Checked: 1229                     |
+---------+------------------------------------------------------+------------------------------------------------------+
| 0 1 0 0 |       Solution: [(1, 1), (2, 1), (3, 0)]             |       Solution: [(1, 1), (2, 1), (3, 0)]             |
| 1 0 1 0 |               Solution Steps: 3                      |               Solution Steps: 3                      |
| 0 0 1 0 |                Time: 0.356 sec                       |                Time: 0.044 sec                       |
| 1 0 0 0 |              Nodes Checked: 1374                     |              Nodes Checked: 1825                     |
+---------+------------------------------------------------------+------------------------------------------------------+
```

A* Algorithm

In this section, we solve the problem using the A* algorithm, which is an informed search algorithm.

This algorithm is based on the cost incurred along the path and the predicted cost provided by heuristic functions. The total cost is determined by adding the actual cost spent on the path and the predicted cost. If the heuristic function is well-designed, this feature provides an optimal solution.

The A* algorithm is a more complete form of the UCS (Uniform Cost Search) algorithm, and it leads us directly and more quickly to the solution.

First, we will examine the heuristic function, and then we will proceed with the algorithm and its code.

Heuristics:
As mentioned, these functions predict the cost of reaching the goal from the current state. Here, we use three heuristics, each of which we will examine separately.

First, let's review the definitions of admissible and consistency:

Admissible: A heuristic is admissible if the estimated distance from the state to the goal is less than or equal to the actual distance. This condition ensures that A* is optimal.

Consistency: A heuristic is consistent if the estimated distance between two states is less than or equal to the actual distance between them.

Manhattan Heuristic:
This function calculates the distance of each lit light (or cells with value 1) to the goal, which is the state of all zeros, and sums them together. In simpler terms, it calculates how much each light needs to be turned off.

This heuristic is not admissible. It does not necessarily estimate the actual cost to reach the goal because it computes the sum of the distances of each lit light from the point (0,0). However, our goal is to evaluate the cost of the actions.

Additionally, it is not consistent because it is not admissible.

lights_on_heuristic:
This heuristic simply counts the number of lights that are currently on. Its assumption, similar to the Manhattan heuristic, is that turning off each light requires at least one action.

Given the simplicity of this function in terms of mathematical operation (counting the number of ones) and the assumption made, it is admissible. This is because it correctly estimates the number of lights that are on, and the estimated cost never exceeds the actual cost. Even if some actions turn off multiple lights simultaneously, it still does not overestimate the real cost, as each light that is on requires at least one action (according to the assumption).

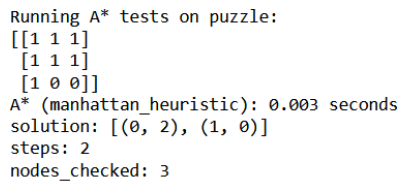
Each move directly decreases the number of lights that are on, and its value appropriately decreases with each change, aligning with the movement cost. Therefore, it is also consistent.

After reviewing the heuristics, we move on to the algorithm.

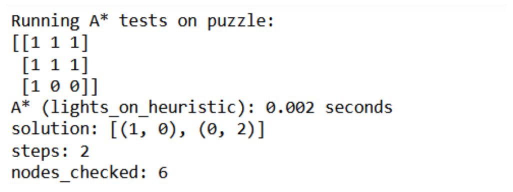
In the code, we follow the reasoning mentioned earlier. There are many similarities with the previous section. We need to add the cost function f. To avoid redundant states, we use a set for that purpose.

The functioning of this algorithm is based on removing the state with the lowest cost from the frontier, testing it against the goal, and updating f at each step.

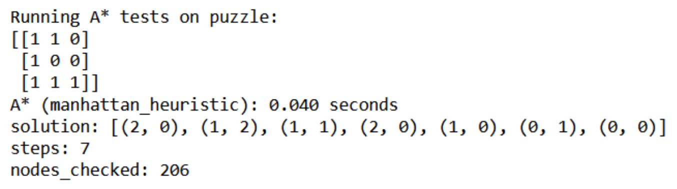
The result is as follows:



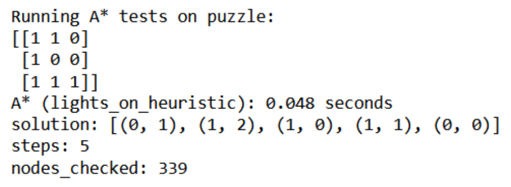
```
Running A* tests on puzzle:
[[1 1 1]
 [1 1 1]
 [1 0 0]]
A* (manhattan_heuristic): 0.003 seconds
solution: [(0, 2), (1, 0)]
steps: 2
nodes_checked: 3
```



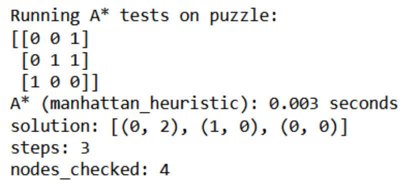
```
Running A* tests on puzzle:
[[1 1 1]
 [1 1 1]
 [1 0 0]]
A* (lights_on_heuristic): 0.002 seconds
solution: [(1, 0), (0, 2)]
steps: 2
nodes_checked: 6
```



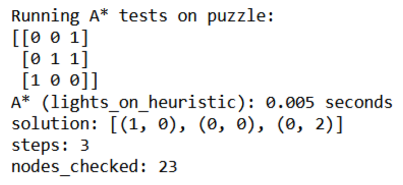
```
Running A* tests on puzzle:
[[1 1 0]
 [1 0 0]
 [1 1 1]]
A* (manhattan_heuristic): 0.040 seconds
solution: [(2, 0), (1, 2), (1, 1), (2, 0), (1, 0), (0, 1), (0, 0)]
steps: 7
nodes_checked: 206
```



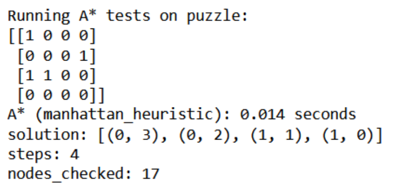
```
Running A* tests on puzzle:
[[1 1 0]
 [1 0 0]
 [1 1 1]]
A* (lights_on_heuristic): 0.048 seconds
solution: [(0, 1), (1, 2), (1, 0), (1, 1), (0, 0)]
steps: 5
nodes_checked: 339
```



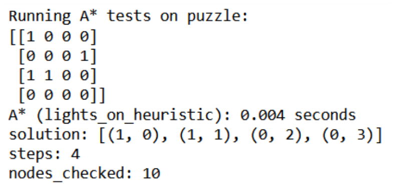
```
Running A* tests on puzzle:
[[0 0 1]
 [0 1 1]
 [1 0 0]]
A* (manhattan_heuristic): 0.003 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 4
```



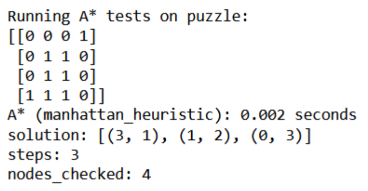
```
Running A* tests on puzzle:
[[0 0 1]
 [0 1 1]
 [1 0 0]]
A* (lights_on_heuristic): 0.005 seconds
solution: [(1, 0), (0, 0), (0, 2)]
steps: 3
nodes_checked: 23
```



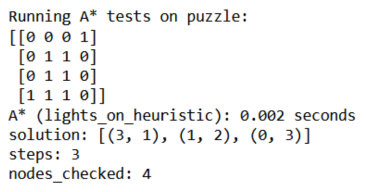
```
Running A* tests on puzzle:
[[1 0 0 0]
 [0 0 0 1]
 [1 1 0 0]
 [0 0 0 0]]
A* (manhattan_heuristic): 0.014 seconds
solution: [(0, 3), (0, 2), (1, 1), (1, 0)]
steps: 4
nodes_checked: 17
```



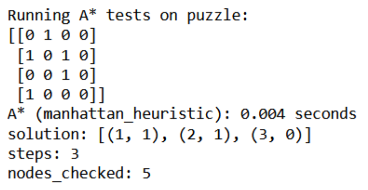
```
Running A* tests on puzzle:
[[1 0 0 0]
 [0 0 0 1]
 [1 1 0 0]
 [0 0 0 0]]
A* (lights_on_heuristic): 0.004 seconds
solution: [(1, 0), (1, 1), (0, 2), (0, 3)]
steps: 4
nodes_checked: 10
```



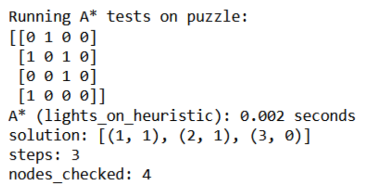
```
Running A* tests on puzzle:
[[0 0 0 1]
 [0 1 1 0]
 [0 1 1 0]
 [1 1 1 0]]
A* (manhattan_heuristic): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4
```



```
Running A* tests on puzzle:
[[0 0 0 1]
 [0 1 1 0]
 [0 1 1 0]
 [1 1 1 0]]
A* (lights_on_heuristic): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4
```



```
Running A* tests on puzzle:
[[0 1 0 0]
 [1 0 1 0]
 [0 0 1 0]
 [1 0 0 0]]
A* (manhattan_heuristic): 0.004 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 5
```



```
Running A* tests on puzzle:
[[0 1 0 0]
 [1 0 1 0]
 [0 0 1 0]
 [1 0 0 0]]
A* (lights_on_heuristic): 0.002 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 4
```

As the results show, based on the predictions, the Manhattan heuristic did not provide the optimal result in some tests. Additionally, the average time for the lights_on heuristic is lower, indicating that it is a better heuristic (assuming the solution is optimal for both methods).

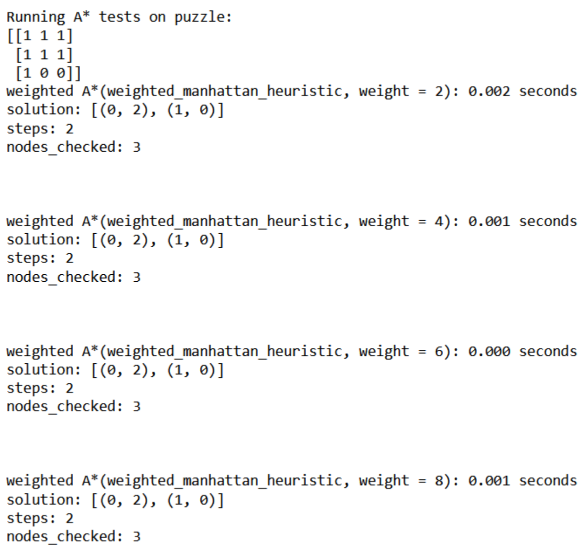
Weighted A* Algorithm

This algorithm is exactly the same as the A* algorithm, with the difference that a weight is multiplied by the heuristic functions. The choice of an appropriate weight helps determine whether the solution is optimal or not. Additionally, the speed of this algorithm should be better, as it moves directly toward the solution.

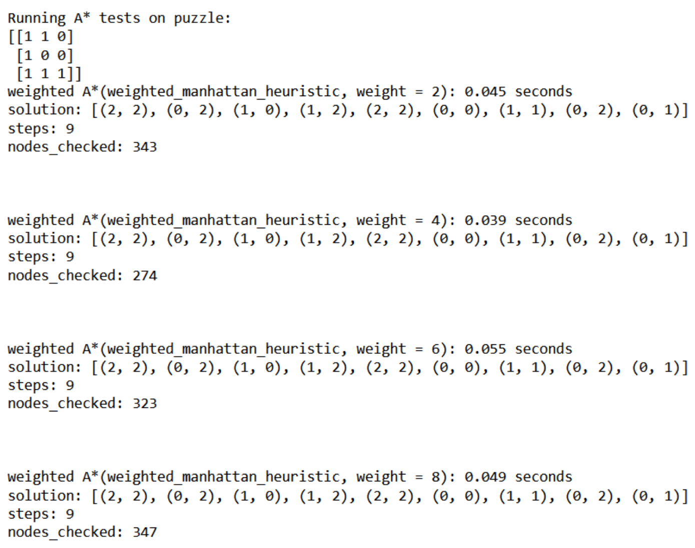
There are no changes to the code, except for the need to define a function to apply the mentioned weight.

With these considerations in mind, let's examine the result.

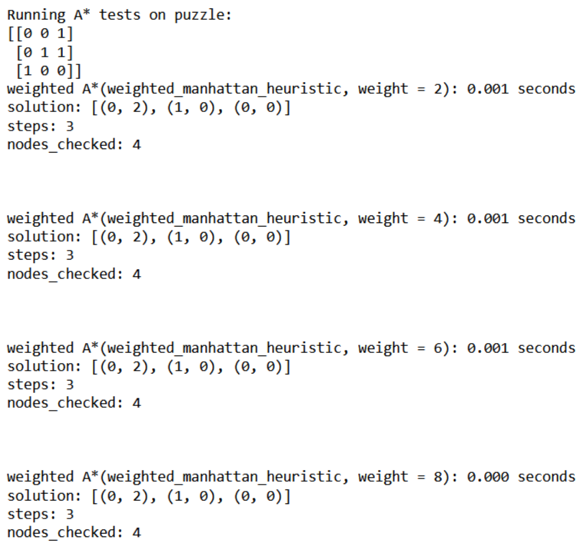
The weights (coefficients) are provided in ascending order to find the best one.



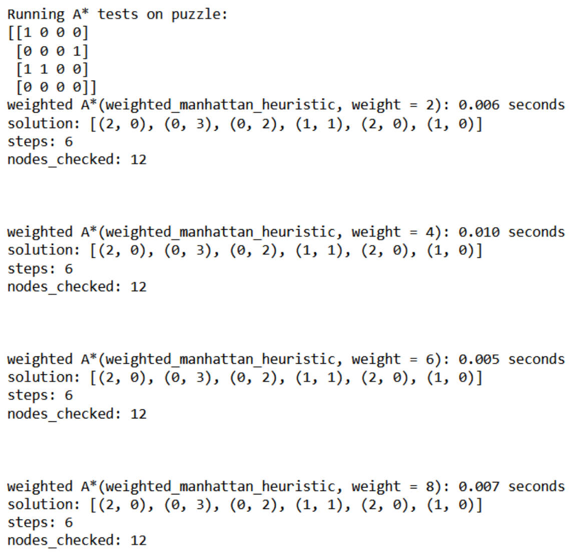
```
Running A* tests on puzzle:
[[1 1 1]
 [1 1 1]
 [1 0 0]]
weighted A*(weighted_manhattan_heuristic, weight = 2): 0.002 seconds
solution: [(0, 2), (1, 0)]
steps: 2
nodes_checked: 3


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.001 seconds
solution: [(0, 2), (1, 0)]
steps: 2
nodes_checked: 3


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.000 seconds
solution: [(0, 2), (1, 0)]
steps: 2
nodes_checked: 3


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.001 seconds
solution: [(0, 2), (1, 0)]
steps: 2
nodes_checked: 3
```



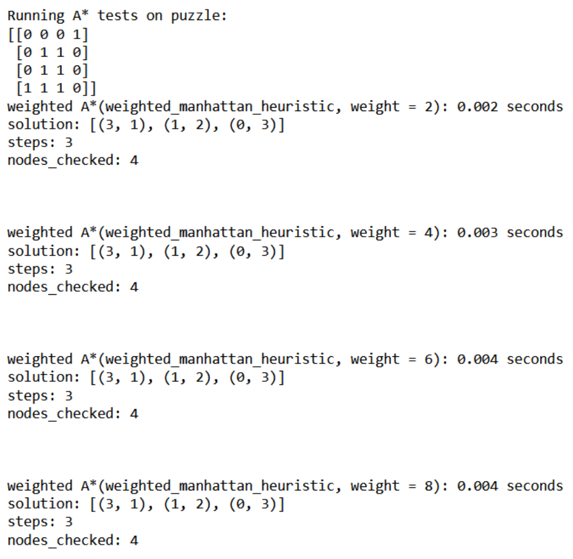
```
Running A* tests on puzzle:
[[1 1 0]
 [1 0 0]
 [1 1 1]]
weighted A*(weighted_manhattan_heuristic, weight = 2): 0.045 seconds
solution: [(2, 2), (0, 2), (1, 0), (1, 2), (2, 2), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 343


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.039 seconds
solution: [(2, 2), (0, 2), (1, 0), (1, 2), (2, 2), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 274


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.055 seconds
solution: [(2, 2), (0, 2), (1, 0), (1, 2), (2, 2), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 323


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.049 seconds
solution: [(2, 2), (0, 2), (1, 0), (1, 2), (2, 2), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 347
```



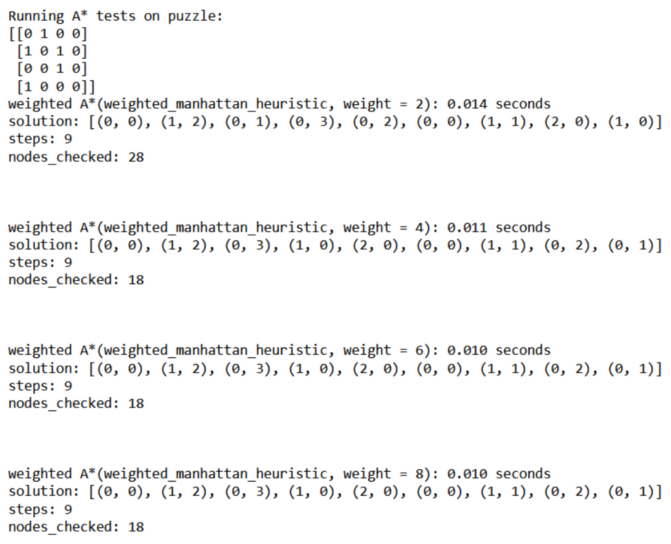
```
Running A* tests on puzzle:
[[0 0 1]
 [0 1 1]
 [1 0 0]]
weighted A*(weighted_manhattan_heuristic, weight = 2): 0.001 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.001 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.001 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.000 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 4
```



```
Running A* tests on puzzle:
[[1 0 0 0]
 [0 0 0 1]
 [1 1 0 0]
 [0 0 0 0]]
weighted A*(weighted_manhattan_heuristic, weight = 2): 0.006 seconds
solution: [(2, 0), (0, 3), (0, 2), (1, 1), (2, 0), (1, 0)]
steps: 6
nodes_checked: 12


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.010 seconds
solution: [(2, 0), (0, 3), (0, 2), (1, 1), (2, 0), (1, 0)]
steps: 6
nodes_checked: 12


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.005 seconds
solution: [(2, 0), (0, 3), (0, 2), (1, 1), (2, 0), (1, 0)]
steps: 6
nodes_checked: 12


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.007 seconds
solution: [(2, 0), (0, 3), (0, 2), (1, 1), (2, 0), (1, 0)]
steps: 6
nodes_checked: 12
```

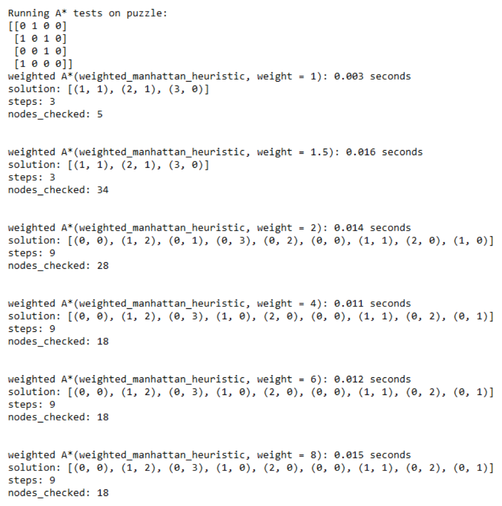


```
Running A* tests on puzzle:
[[0 0 0 1]
 [0 1 1 0]
 [0 1 1 0]
 [1 1 1 0]]
weighted A*(weighted_manhattan_heuristic, weight = 2): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.003 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.004 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.004 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4
```



```
Running A* tests on puzzle:
[[0 1 0 0]
 [1 0 1 0]
 [0 0 1 0]
 [1 0 0 0]]
weighted A*(weighted_manhattan_heuristic, weight = 2): 0.014 seconds
solution: [(0, 0), (1, 2), (0, 1), (0, 3), (0, 2), (0, 0), (1, 1), (2, 0), (1, 0)]
steps: 9
nodes_checked: 28


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.011 seconds
solution: [(0, 0), (1, 2), (0, 3), (1, 0), (2, 0), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 18


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.010 seconds
solution: [(0, 0), (1, 2), (0, 3), (1, 0), (2, 0), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 18


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.010 seconds
solution: [(0, 0), (1, 2), (0, 3), (1, 0), (2, 0), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 18
```

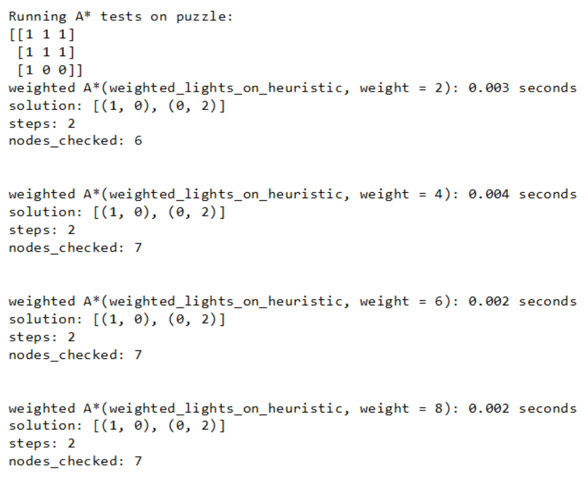
By examining and comparing the results, we notice that as the weights increase, the results deviate from optimality, which can be observed in the last test on this page. However, by comparing it with the A* results, we can see the difference. We also analyze the final matrix with w=1.5.



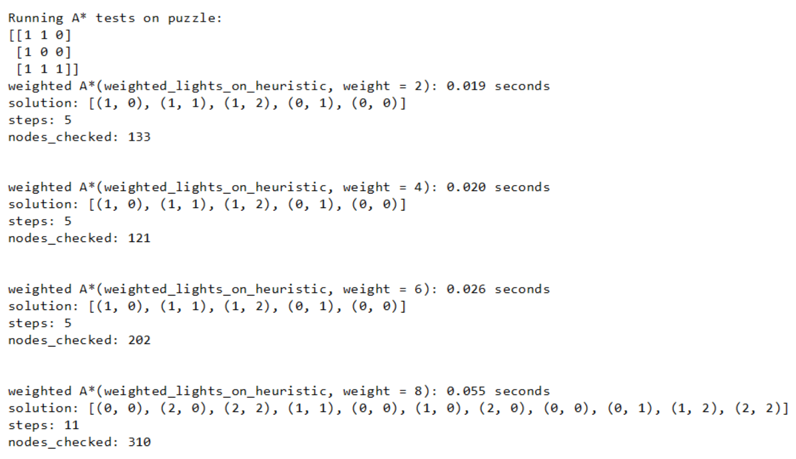
```
Running A* tests on puzzle:
[[0 1 0 0]
 [1 0 1 0]
 [0 0 1 0]
 [1 0 0 0]]
weighted A*(weighted_manhattan_heuristic, weight = 1): 0.003 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 5


weighted A*(weighted_manhattan_heuristic, weight = 1.5): 0.016 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 34


weighted A*(weighted_manhattan_heuristic, weight = 2): 0.014 seconds
solution: [(0, 0), (1, 2), (0, 1), (0, 3), (0, 2), (0, 0), (1, 1), (2, 0), (1, 0)]
steps: 9
nodes_checked: 28


weighted A*(weighted_manhattan_heuristic, weight = 4): 0.011 seconds
solution: [(0, 0), (1, 2), (0, 3), (1, 0), (2, 0), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 18


weighted A*(weighted_manhattan_heuristic, weight = 6): 0.012 seconds
solution: [(0, 0), (1, 2), (0, 3), (1, 0), (2, 0), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 18


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.015 seconds
solution: [(0, 0), (1, 2), (0, 3), (1, 0), (2, 0), (0, 0), (1, 1), (0, 2), (0, 1)]
steps: 9
nodes_checked: 18
```

As we can see, w=1.5 gives the same result as w=1(which corresponds to the A* algorithm). Therefore, the coefficient w=1.5 is a better choice.

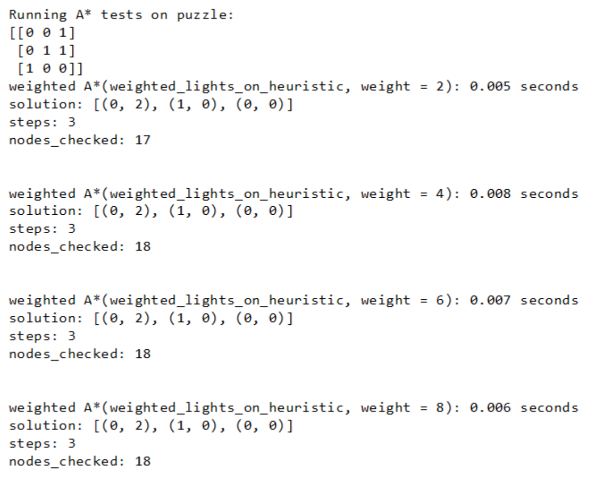
Now, let's observe the results obtained from the lights_on_heuristic.



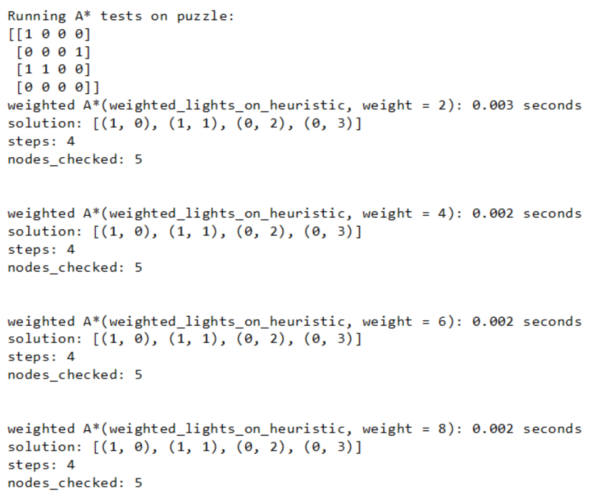
```
Running A* tests on puzzle:
[[1 1 1]
 [1 1 1]
 [1 0 0]]
weighted A*(weighted_lights_on_heuristic, weight = 2): 0.003 seconds
solution: [(1, 0), (0, 2)]
steps: 2
nodes_checked: 6

weighted A*(weighted_lights_on_heuristic, weight = 4): 0.004 seconds
solution: [(1, 0), (0, 2)]
steps: 2
nodes_checked: 7

weighted A*(weighted_lights_on_heuristic, weight = 6): 0.002 seconds
solution: [(1, 0), (0, 2)]
steps: 2
nodes_checked: 7

weighted A*(weighted_lights_on_heuristic, weight = 8): 0.002 seconds
solution: [(1, 0), (0, 2)]
steps: 2
nodes_checked: 7
```



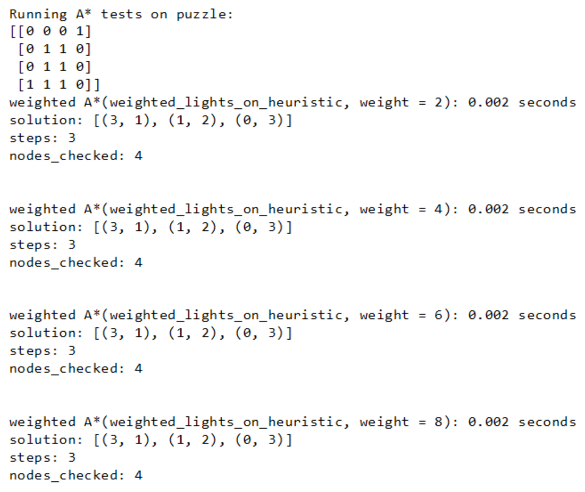
```
Running A* tests on puzzle:
[[1 1 0]
 [1 0 0]
 [1 1 1]]
weighted A*(weighted_lights_on_heuristic, weight = 2): 0.019 seconds
solution: [(1, 0), (1, 1), (1, 2), (0, 1), (0, 0)]
steps: 5
nodes_checked: 133

weighted A*(weighted_lights_on_heuristic, weight = 4): 0.020 seconds
solution: [(1, 0), (1, 1), (1, 2), (0, 1), (0, 0)]
steps: 5
nodes_checked: 121

weighted A*(weighted_lights_on_heuristic, weight = 6): 0.026 seconds
solution: [(1, 0), (1, 1), (1, 2), (0, 1), (0, 0)]
steps: 5
nodes_checked: 202

weighted A*(weighted_lights_on_heuristic, weight = 8): 0.055 seconds
solution: [(0, 0), (2, 0), (2, 2), (1, 1), (0, 0), (1, 0), (2, 0), (0, 0), (0, 1), (1, 2), (2, 2)]
steps: 11
nodes_checked: 310
```



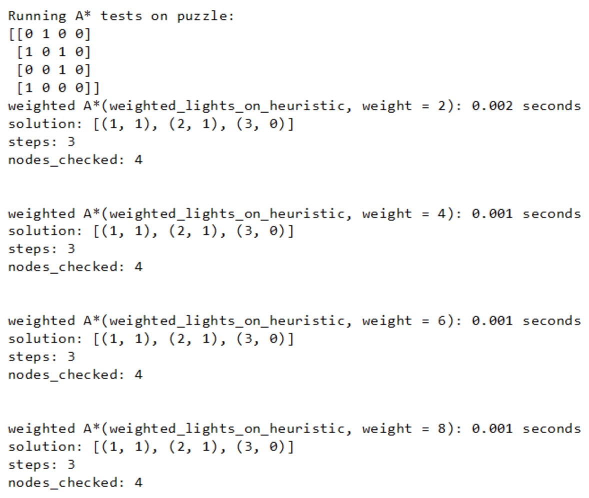
```
Running A* tests on puzzle:
[[0 0 1]
 [0 1 1]
 [1 0 0]]
weighted A*(weighted_lights_on_heuristic, weight = 2): 0.005 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 17

weighted A*(weighted_lights_on_heuristic, weight = 4): 0.008 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 18

weighted A*(weighted_lights_on_heuristic, weight = 6): 0.007 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 18

weighted A*(weighted_lights_on_heuristic, weight = 8): 0.006 seconds
solution: [(0, 2), (1, 0), (0, 0)]
steps: 3
nodes_checked: 18
```



```
Running A* tests on puzzle:
[[1 0 0 0]
 [0 0 0 1]
 [1 1 0 0]
 [0 0 0 0]]
weighted A*(weighted_lights_on_heuristic, weight = 2): 0.003 seconds
solution: [(1, 0), (1, 1), (0, 2), (0, 3)]
steps: 4
nodes_checked: 5

weighted A*(weighted_lights_on_heuristic, weight = 4): 0.002 seconds
solution: [(1, 0), (1, 1), (0, 2), (0, 3)]
steps: 4
nodes_checked: 5

weighted A*(weighted_lights_on_heuristic, weight = 6): 0.002 seconds
solution: [(1, 0), (1, 1), (0, 2), (0, 3)]
steps: 4
nodes_checked: 5

weighted A*(weighted_lights_on_heuristic, weight = 8): 0.002 seconds
solution: [(1, 0), (1, 1), (0, 2), (0, 3)]
steps: 4
nodes_checked: 5
```



```
Running A* tests on puzzle:
[[0 0 0 1]
 [0 1 1 0]
 [0 1 1 0]
 [1 1 1 0]]
weighted A*(weighted_lights_on_heuristic, weight = 2): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4

weighted A*(weighted_lights_on_heuristic, weight = 4): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4

weighted A*(weighted_lights_on_heuristic, weight = 6): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4

weighted A*(weighted_lights_on_heuristic, weight = 8): 0.002 seconds
solution: [(3, 1), (1, 2), (0, 3)]
steps: 3
nodes_checked: 4
```

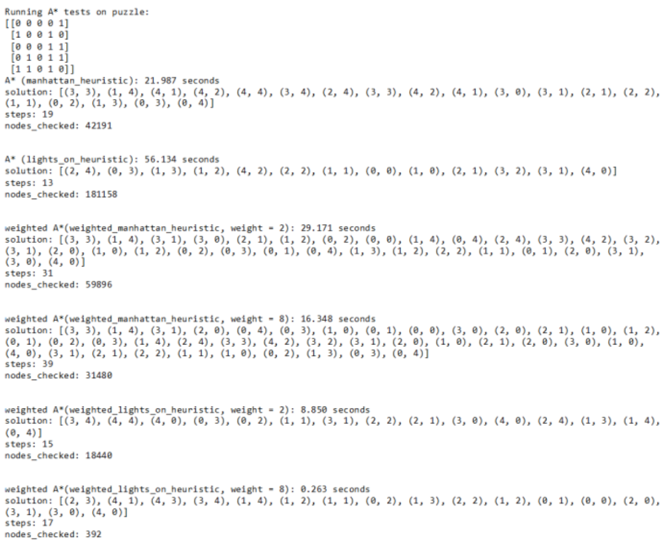


```
Running A* tests on puzzle:
[[0 1 0 0]
 [1 0 1 0]
 [0 0 1 0]
 [1 0 0 0]]
weighted A*(weighted_lights_on_heuristic, weight = 2): 0.002 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 4

weighted A*(weighted_lights_on_heuristic, weight = 4): 0.001 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 4

weighted A*(weighted_lights_on_heuristic, weight = 6): 0.001 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 4

weighted A*(weighted_lights_on_heuristic, weight = 8): 0.001 seconds
solution: [(1, 1), (2, 1), (3, 0)]
steps: 3
nodes_checked: 4
```

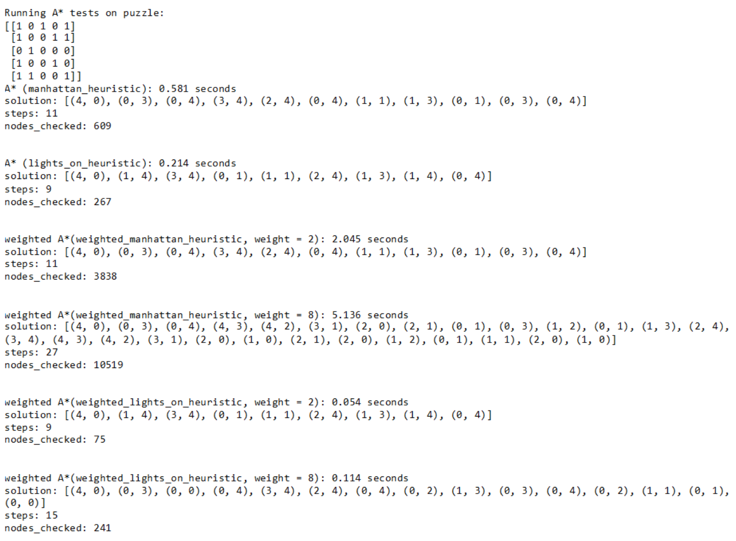
Once again, it is clear that choosing incorrect weights affects the optimality of the result.

The reason for the lack of optimality with inappropriate weights is that the algorithm focuses more on the heuristic and pays less attention to the path cost. The only advantage of this method is that it reaches the solution more quickly. As we can see, the average time for this algorithm with both heuristics is less than a millisecond, which supports our point. However, it sacrifices optimality for this reduced time.

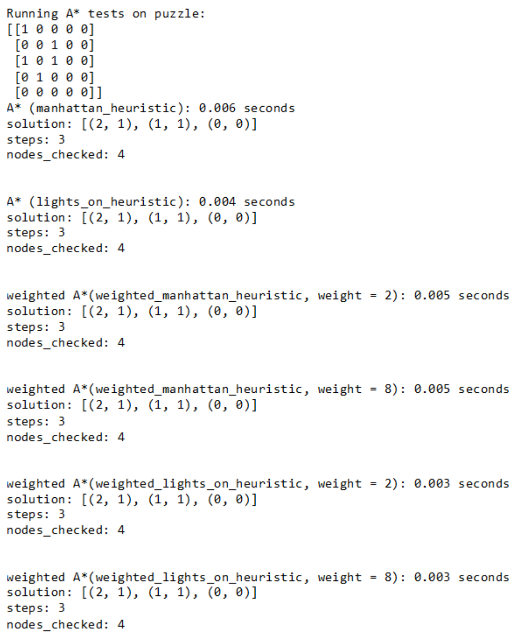
Now, let's examine the 5×5 matrix as the input. Naturally, as the data increases, the time required for processing will also increase. Now, let's examine this topic further to gain a clearer understanding of the two algorithms.



```
Running A* tests on puzzle:
[[0 0 0 0 1]
 [1 0 0 1 0]
 [0 0 0 1 1]
 [0 1 0 1 1]
 [1 1 0 1 0]]
A* (manhattan_heuristic): 21.987 seconds
solution: [(3, 3), (1, 4), (4, 1), (4, 2), (4, 4), (3, 4), (2, 4), (3, 3), (4, 2), (4, 1), (3, 0), (3, 1), (2, 1), (2, 2),
(1, 1), (0, 2), (1, 3), (0, 3), (0, 4)]
steps: 19
nodes_checked: 42191


A* (lights_on_heuristic): 56.134 seconds
solution: [(2, 4), (0, 3), (1, 3), (1, 2), (4, 2), (2, 2), (1, 1), (0, 0), (1, 0), (2, 1), (3, 2), (3, 1), (4, 0)]
steps: 13
nodes_checked: 181158


weighted A*(weighted_manhattan_heuristic, weight = 2): 29.171 seconds
solution: [(3, 3), (1, 4), (3, 1), (3, 0), (2, 1), (1, 2), (0, 2), (0, 0), (1, 4), (0, 4), (2, 4), (3, 3), (4, 2), (3, 2),
(3, 1), (2, 0), (1, 0), (1, 2), (0, 2), (0, 3), (0, 1), (0, 4), (1, 3), (1, 2), (2, 2), (1, 1), (0, 1), (2, 0), (3, 1),
(3, 0), (4, 0)]
steps: 31
nodes_checked: 59896


weighted A*(weighted_manhattan_heuristic, weight = 8): 16.348 seconds
solution: [(3, 3), (1, 4), (3, 1), (2, 0), (0, 4), (0, 3), (1, 0), (0, 1), (0, 0), (3, 0), (2, 0), (2, 1), (1, 0), (1, 2),
(0, 1), (0, 2), (0, 3), (1, 4), (2, 4), (3, 3), (4, 2), (3, 2), (3, 1), (2, 0), (1, 0), (2, 1), (2, 0), (3, 0), (1, 0),
(4, 0), (3, 1), (2, 1), (2, 2), (1, 1), (1, 0), (0, 2), (1, 3), (0, 3), (0, 4)]
steps: 39
nodes_checked: 31480


weighted A*(weighted_lights_on_heuristic, weight = 2): 8.850 seconds
solution: [(3, 4), (4, 4), (4, 0), (0, 3), (0, 2), (1, 1), (3, 1), (2, 2), (2, 1), (3, 0), (4, 0), (2, 4), (1, 3), (1, 4),
(0, 4)]
steps: 15
nodes_checked: 18440


weighted A*(weighted_lights_on_heuristic, weight = 8): 0.263 seconds
solution: [(2, 3), (4, 1), (4, 3), (3, 4), (1, 4), (1, 2), (1, 1), (0, 2), (1, 3), (2, 2), (1, 2), (0, 1), (0, 0), (2, 0),
(3, 1), (3, 0), (4, 0)]
steps: 17
nodes_checked: 392
```



```
Running A* tests on puzzle:
[[1 0 1 0 1]
 [1 0 0 1 1]
 [0 1 0 0 0]
 [1 0 0 1 0]
 [1 1 0 0 1]]
A* (manhattan_heuristic): 0.581 seconds
solution: [(4, 0), (0, 3), (0, 4), (3, 4), (2, 4), (0, 4), (1, 1), (1, 3), (0, 1), (0, 3), (0, 4)]
steps: 11
nodes_checked: 609


A* (lights_on_heuristic): 0.214 seconds
solution: [(4, 0), (1, 4), (3, 4), (0, 1), (1, 1), (2, 4), (1, 3), (1, 4), (0, 4)]
steps: 9
nodes_checked: 267


weighted A*(weighted_manhattan_heuristic, weight = 2): 2.045 seconds
solution: [(4, 0), (0, 3), (0, 4), (3, 4), (2, 4), (0, 4), (1, 1), (1, 3), (0, 1), (0, 3), (0, 4)]
steps: 11
nodes_checked: 3838


weighted A*(weighted_manhattan_heuristic, weight = 8): 5.136 seconds
solution: [(4, 0), (0, 3), (0, 4), (4, 3), (4, 2), (3, 1), (2, 0), (2, 1), (0, 1), (0, 3), (1, 2), (0, 1), (1, 3), (2, 4),
(3, 4), (4, 3), (4, 2), (3, 1), (2, 0), (1, 0), (2, 1), (2, 0), (1, 2), (0, 1), (1, 1), (2, 0), (1, 0)]
steps: 27
nodes_checked: 10519


weighted A*(weighted_lights_on_heuristic, weight = 2): 0.054 seconds
solution: [(4, 0), (1, 4), (3, 4), (0, 1), (1, 1), (2, 4), (1, 3), (1, 4), (0, 4)]
steps: 9
nodes_checked: 75


weighted A*(weighted_lights_on_heuristic, weight = 8): 0.114 seconds
solution: [(4, 0), (0, 3), (0, 0), (0, 4), (3, 4), (2, 4), (0, 4), (0, 2), (1, 3), (0, 3), (0, 4), (0, 2), (1, 1), (0, 1),
(0, 0)]
steps: 15
nodes_checked: 241
```



```
Running A* tests on puzzle:
[[1 0 0 0 0]
 [0 0 1 0 0]
 [1 0 1 0 0]
 [0 1 0 0 0]
 [0 0 0 0 0]]
A* (manhattan_heuristic): 0.006 seconds
solution: [(2, 1), (1, 1), (0, 0)]
steps: 3
nodes_checked: 4


A* (lights_on_heuristic): 0.004 seconds
solution: [(2, 1), (1, 1), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 2): 0.005 seconds
solution: [(2, 1), (1, 1), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_manhattan_heuristic, weight = 8): 0.005 seconds
solution: [(2, 1), (1, 1), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_lights_on_heuristic, weight = 2): 0.003 seconds
solution: [(2, 1), (1, 1), (0, 0)]
steps: 3
nodes_checked: 4


weighted A*(weighted_lights_on_heuristic, weight = 8): 0.003 seconds
solution: [(2, 1), (1, 1), (0, 0)]
steps: 3
nodes_checked: 4
```

Based on the results obtained, we conclude that the acceleration of time may also be related to the initial state. For example, in the first input, a significant amount of time was spent on all algorithms, whereas this was different for the last input. Additionally, the lights_on_heuristic, due to its properties such as being admissible and having a good weight for the response, is the best algorithm for solving the problem.

Overall, the A* method and, even better, Weighted A* are more suitable for solving pathfinding problems.

It is worth mentioning that the table for these two algorithms contains many test cases, which were presented in the report above, but the necessary table is available at the end of the corresponding code. Additionally, the 5×5 inputs are commented out in the test cases, but if the comments are removed, the results will be correctly achieved.