**Eight Puzzle**

1. Misplaced Tiles Heuristic

This particular heuristic calculates the heuristic cost for each state by comparing it to the goal state tile by tile: If the tile position in the current node state is different from its position in the goal state, then the heuristic cost is incremented by 1. It removes physical constraints and it assumes only a cost of 1 for removing it from a remote place to a close place and disregards all the other tiles that need to move as a result.

*Admissibility*:

The misplaced tiles heuristic is admissible. In fact, the total number of moves to order the tiles correctly is at least the number of misplaced tiles: each tile needs to move at least by one position to reach its correct goal position, so the heuristic never overestimates and is therefore always optimistic.

*Consistency*:

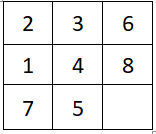
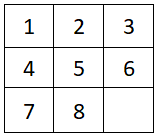
For consistency, for each arc in our tree, only one tile is moved. Therefore, for each arc, the heuristic is either decreased by 1 if you move it to its accurate place or remains the same if it is moved but still in a misplaced position or increased by 1 if you move it from its accurate position to a misplaced one. Also, we have 1 as the cost for each move.

So, the change in the heuristic from point A to point B should be less or equal to the cost **h(B)-h(A) <= c(A-B).**

*Performance according to the search algorithm used*:

* With Greedy Best First Search:

Initial state Goal state

In this particular example, using the misplaced tiles heuristic, the GBFS expanded 79 nodes to reach the goal and 61 already visited nodes were found in the closed list. Meanwhile, if we use an uninformed search method such as BFS, less nodes are expanded (46 to be exact) and only 26 already visited nodes are found. However, even if it expanded more nodes, the cost of the solution was still only 32 compared to BFS for instance where the cost was 46.

Generally, the misplaced tiles heuristic doesn’t take into consideration the obstacles in front of each tile and therefore optimistic values are assigned to several states, so the GBFS keeps expanding these nodes because of their low heuristic values before ultimately reaching the goal.

* With A\*: The heuristic performs much better using A\*. For instance, using the same example above, it only expanded 8 nodes and 0 already visited nodes were found. The cost of the solution found was also lower (only 8).

Generally, A\* offers a more realistic value to the evaluation function by including, in addition to the heuristic, the path cost from the initial node to the current node. This allows the algorithm to order the frontier more efficiently; even if the heuristic value of a certain state is overly optimistic the evaluation function will also consider its high path cost and therefore will not place it at the front of the fringe.

1. Manhattan Distance Heuristic

The manhattan distance of a certain state in the search space is calculated by considering the number of moves each tile has to perform in order to get to its goal position which is the sum of the horizontal distance and the vertical distance to the right place of each tile. This heuristic also gets rid of some physical constraints (the fact that while moving one tile, other tiles will have to move along the way) but it takes into account how many positions the tile needs to move to its destination, contrary to the misplaced tiles which always assumes the cost of 1.

*Admissibility*:

The manhattan distance heuristic is admissible. It is derived from a relaxed version of the problem, and therefore the cost it estimates will always be lower than the lowest possible cost to reach the goal. In fact, each tile has to move at least the number of positions between its current place and its goal position, and will possibly have to perform more moves in case it changes its position to a further place from its goal destination if it gets in the way of another tile that needs to be moved.

*Consistency*:

For consistency, for each arc representing one tile move, the tile is either one vertical/horizontal move closer to its actual state or one vertical/horizontal move further from it. Therefore, the heuristic drop in one arc is less or equal to the actual cost which is 1.

**h(B)-h(A) <= c(A-B)**

*Performance according to the search algorithm used*:

* With Greedy Best First Search: Using the same examples of the initial state and goal states above, GBFS expanded 21 nodes using the Manhattan heuristic, and 13 already visited nodes were found.

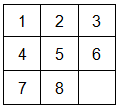
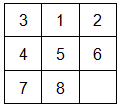
This heuristic performs better with GBFS compared to the misplaced tiles since it assigns less optimistic values and therefore it dominates it.

* With A\*: In this case, 12 nodes were expanded and 8 were found to be already visited. However, the same solution cost was obtained since A\* is guaranteed to find the optimal solution using an admissible heuristic.

1. Custom Heuristic

With the Manhattan heuristic, we are not taking into consideration that other tiles might need to be moved out of the way. Therefore in our heuristic, in addition to the Manhattan distance, for every row and column, we check for conflicts. In other words, tile A is in conflict with tile B if the latter presents an obstacle while trying to move A to its goal position and vice versa with the condition that both A and B are in the same row/column as their goal positions.

Initial state Goal state



In the given puzzle, 3 is in conflict with 1. Meaning, that in order for 3 to get into its right position 1 has to be moved out of its way. To do so, we need to move 1 down leading to 2 extra moves for 1 since it has to be moved out of the way by one move and we also need one move for it to get back to the row/column from which it was moved.

*Admissibility*:

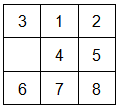
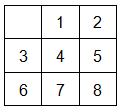
So far, the heuristic is not guaranteed to be admissible. For instance, in the example above, 3 is in conflict with both 1 and 2. Using that logic, both 1 and 2 should be moved out of the way resulting in 2 extra moves for each. However, while solving the puzzle, it is possible that the user moves 3 out of the way instead of both 1 and 2. In order to guarantee admissibility, we must execute the least costly resolution of conflicts in that row/column which is what we did in our code.

To further analyze admissibility, let’s take the worst case for our heuristic. In the latter, we will have each tile in the row/column in conflict with the other two for instance the first row is 3 2 1. ‘2’ creates a conflict for ‘3’ and ‘1’, ‘3’ creates a conflict for ‘2’ and ‘1’, and ‘1’ creates a conflict for ‘3’ and ‘2’. In such a case, we will need to make 4 additional moves to the total Manhattan distance and repeat for each row/column. Here, we are assuming that while moving out the two tiles in conflict we won’t need to make any additional moves. However, say we have to move 2 and 1 out of the way. If we are optimistic and assume that the blank is under 2 then we can remove 2 out of the way in one move and the first row becomes 3 \_ 1. So we cannot just move 1 out of the row in one move since the blank is not below it. So that means we will need more than 4 moves. Therefore, our heuristic underestimates the number of moves for conflict resolution, making our heuristic admissible.

*Consistency*:

For consistency, we will redo the same analysis but for a single arc. For each arc representing one tile move, the tile is either one vertical/horizontal move closer to its actual state or one vertical/horizontal move further from it. Additionally, for each arc, the heuristic is either increased by 2 if the tile moved creates a conflict, decreased by 2 if moving the tile resolves a conflict, or stays the same if there was no conflict to begin with, and moving the tile didn’t create one.

To show that this heuristic is consistent given that the cost of one arc is one, we need to show that when the heuristic increases by two creating a conflict, the Manhattan decreases so that the sum is still less or equal to one. A new conflict is created only if a tile is moved to its goal row/column from a wrong row (if the tile was already in its goal row/column the conflict would have existed before moving it), so since we moved to the goal row/column the horizontal/vertical distance to the goal state is decreased by one and therefore the manhattan distance drop is -1, so the overall heuristic changes by 2-1=1 which is equal to the actual cost.



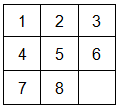
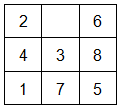
For this example, if we move the blank down (the cost is one), a conflict will be created by ‘3’ in the first row, so the heuristic will increase by ‘2’ however the Manhattan distance of ‘3’ decreases by 1.

For the other two cases (a conflict is removed or no change in conflicts) it is consistent because the Manhattan is already consistent so if 2 is subtracted or 0 is added the drop in the heuristic the sum is still less than the cost.

*Performance according to the search algorithm used*:

We tested our code on :

Initial state Goal state

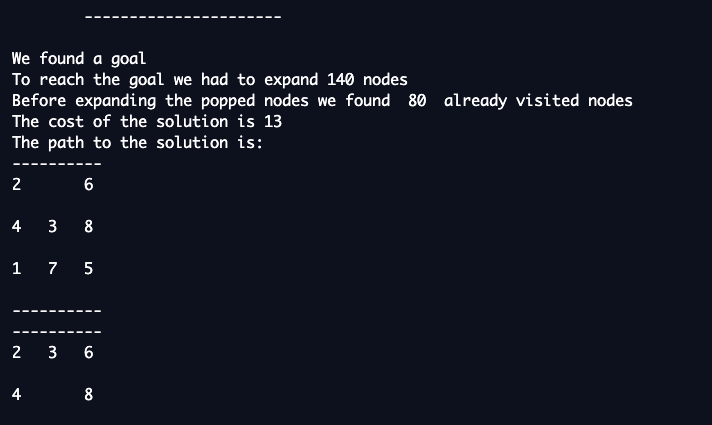


|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Misplaced tiles | | Manhattan | | Custom | |
| A\* | Greedy | A\* | Greedy | A\* | Greedy |
| Expanded nodes | 140 | 742 | 59 | 330 | 24 | 87 |
| Already visited | 80 | 755 | 30 | 313 | 7 | 76 |
| Path Cost | 13 | 141 | 13 | 63 | 13 | 29 |

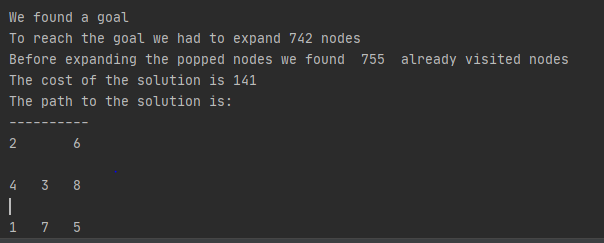
Note that values can vary for different runs because operators are written as a set (no order which gives randomness) and the order at which nodes are put in the frontier can vary for nodes with similar heuristic values.

**Misplaced:**

A\*:

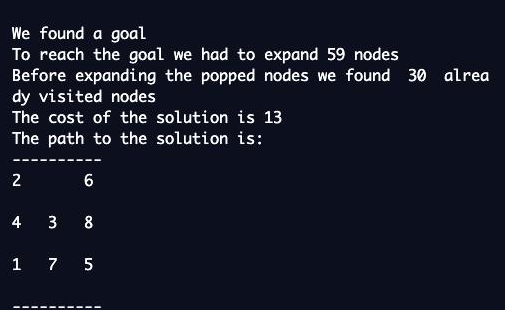


Greedy:

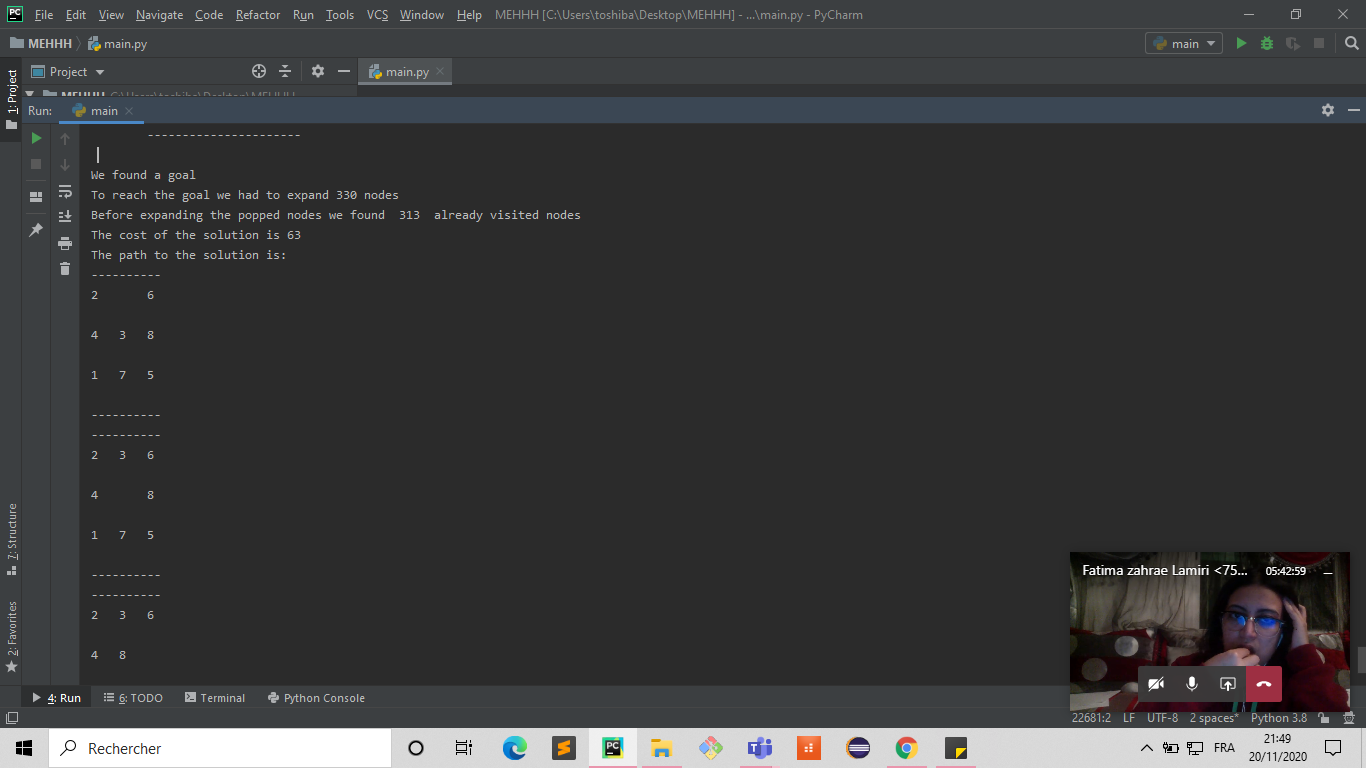


**Manhattan:**

A\*:

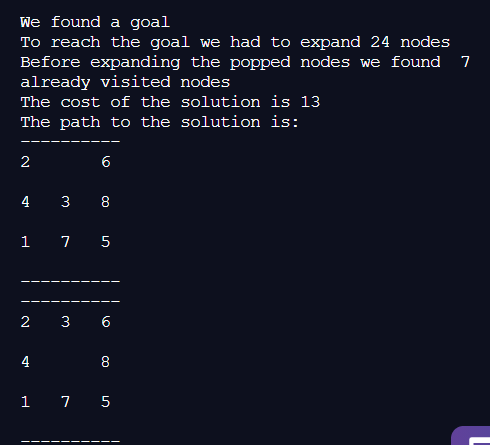


Greedy:

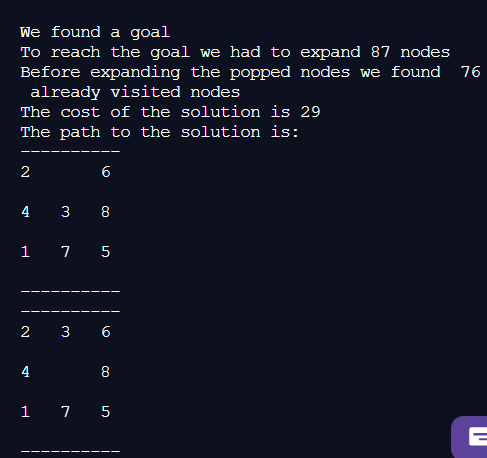


**Custom:**

A\* :



Greedy:

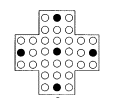


* We also tested on a few other cases and left them running for a while. Generally, the greedy takes longer than A\* and expands more nodes except for a few cases where the optimal path might require A\* to expand a lot of nodes. In terms of heuristic, as seen from the table, the custom heuristic dominates the Manhattan distance heuristic (because it is equal to the manhattan plus a value (0,2, or 4) ) and the latter dominates the misplaced tiles heuristics.

**Peg Solitaire**

Isolated Pegs Heuristic**:**

This heuristic checks the number of pegs that are isolated and have no neighbors.



In this example, all five pegs are isolated.

*Admissibility*:

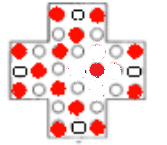
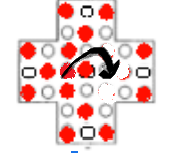
To solve this puzzle we need to remove all the pegs but one. Therefore, the number of steps required to solve it is at least the number of pegs remaining on the board minus one. Each step is associated with a cost of one, so the minimum cost is N-1 (N is the number of pegs left in the board).

For our heuristic, we assume we need to remove only the isolated pegs. The number of isolated pegs is less or equal to the number of remaining pegs.

* If the number of isolated pegs ‘M’ was less than the number of remaining pegs, it means there are some pegs that are not isolated, then M is less than N which means M<=N-1 which is less or equal than the actual cost.
* If the number of isolated pegs ‘M’ is equal to ‘N’, then all the pegs left are isolated, so the state has no solution and our heuristic is still admissible.

*Consistency*:

For consistency, we believe our heuristic is not consistent because of the counterexample:

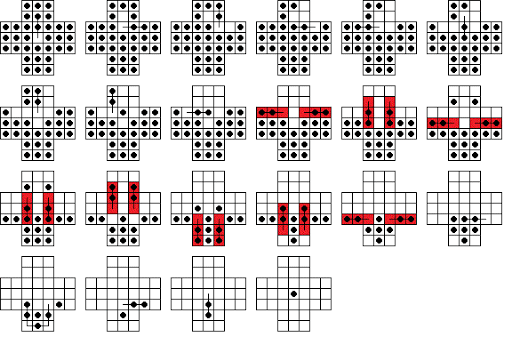


In this case, one jump of cost 1 can lead to 3 isolated pegs so the heuristic drop is greater than the cost, making our heuristic non-consistent.

*Performance according to the search algorithm used*:

We prompt the user for the initial state. We included some examples of initial states in a .txt file “peg\_initial\_states.txt”.

The further you are from the goal the more time it will take.



In some of these cases, the search might take too long.

*Performance according to the search algorithm used*:

* \* is a hole and O is a peg

Staring from the given initial state in the handout we reached the goal after a long time on DFS:



Running the same initial state takes way too long with the other strategies. It makes sense since the branching factor is too big for BFS to find a quick solution.

Let us test on the case of

[

0, 0, 2, 2, 2, 0, 0,

0, 0, 2, 2, 2, 0, 0,

2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 2, 2, 2, 2, **≡**

1, 1, 2, 1, 2, 1, 1,

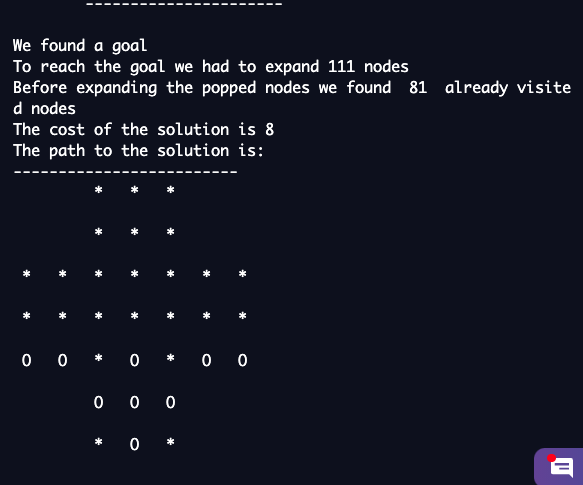
0, 0, 1, 1, 1, 0, 0,

0, 0, 2, 1, 2, 0, 0

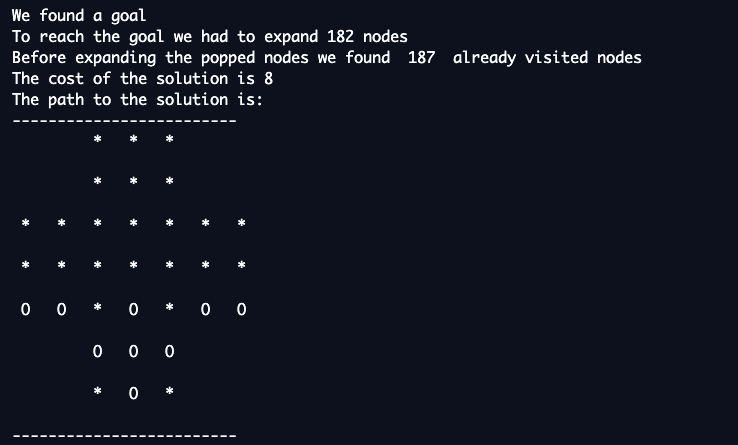
]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Isolated Pegs | | | |
| DFS | BFS | Greedy | A\* |
| Expanded nodes | 111 | 182 | 72 | 72 |
| Already visited | 81 | 187 | 25 | 25 |
| Path Cost | 8 | 8 | 8 | 8 |

DFS:



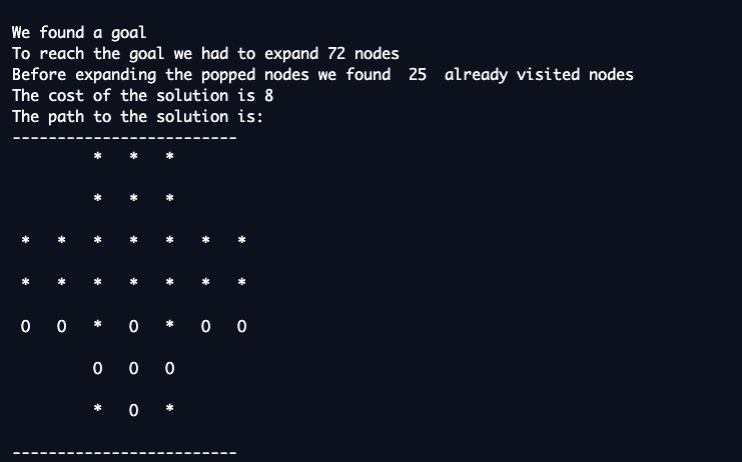
BFS:



Greedy:



A\*:



DFS finds a solution in a relatively short time regardless of how dense the initial state is (in terms of number of pegs), and this is due to the fact that the goal state is always far from the initial state.

Informed search on the other hand expands less nodes than the uninformed, for this example the greedy and A\* performances are similar, but the difference should show up when testing on cases that are far from the goal state. Unfortunately, we could not test on those cases because of time complexity issues.

**Missionaries and Cannibals:**

1. Trip without return

This heuristic estimates the number of trips left, without taking into consideration that someone needs to go back to the starting side for the next trip to take place, and that cannibals can eat the missionaries if they outnumber them. So it calculates the number of people left on the left side of the river divided by the boat capacity (2).



For the example above, the heuristic will return 6/2=3.

*Admissibility:*

The heuristic always underestimates the number of moves needed to take everyone from one state to another since it doesn’t take into consideration the extra move needed to bring the boat back, and ignores the fact that someone will come back each time.

Initially, we have 6 people on the left side. After one trip, we are left with 4 people there. The heuristic will then assume that we are left with 2 trips, however we will need to have one cannibal/missionary go back in the boat so we are actually left with 5 people. Therefore, the heuristic underestimates the total trips needed and is admissible.

*Consistency:*

For consistency, if we take one arc in the search tree, it represents one trip either from the left to the right or from the right to the left.

* In case the trip is from the left to the right the boat will hold 2 people resulting in a heuristic drop of one (one trip less).
* While going from the right to the left, one or two people will go back with the boat the arc heuristic increases by 0,5 or 1 in that case. However, this trip back will cost 1 because we cannot have 0.5 trips.

So, the change in the heuristic from point A to point B is less or equal to the cost **h(B)-h(A) <= c(A-B).**

1. Trip with return

Unlike the previous heuristic, this one actually considers that we need to bring back someone with the boat from the right side except for the last trip.

Nbr of C&M on the left -1

*Admissibility*:

In this case, we are always assuming that 2 trips (going right then left) will result in the transfer of at most one person to the right side resulting in a trip. In the real problem at least 1 or 2 people can go back to the left side with the boat resulting either way in a trip. Therefore, the heuristic never overestimates making it admissible.

*Consistency*:

This heuristic is not always consistent, because one trip back from the right to the left side while bringing two people will add 2 to the heuristic drop value while the cost is just one trip.



For instance in the following picture:

The heuristic before this trip A is:

2-1 = 1

The heuristic after the trip B is:

4-1 = 3

But the cost associated with the trip

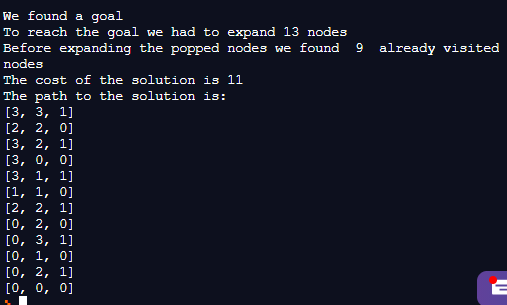
is 1.

Therefore:

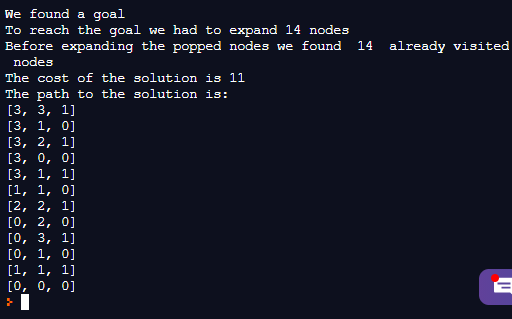
**h(B)-h(A) = 2 > c(A-B) = 1**

*Performance according to the search algorithm and heuristic used*:

DFS:

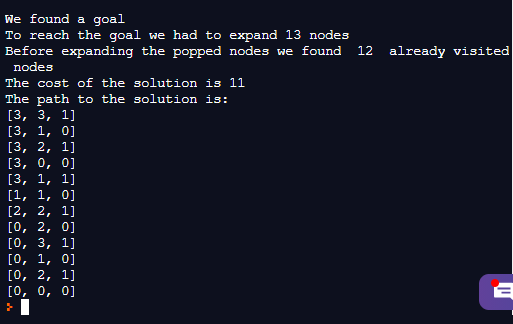


BFS:

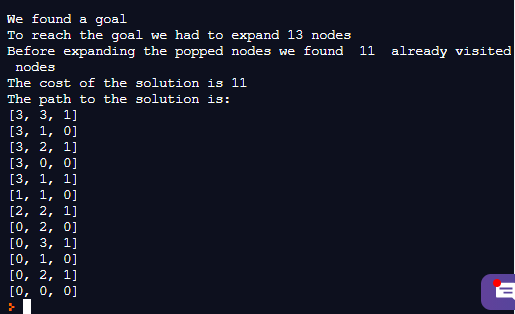


A\*:

* With trip without returns:

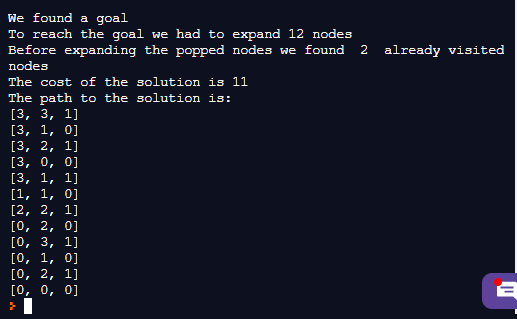


* With trip with returns:

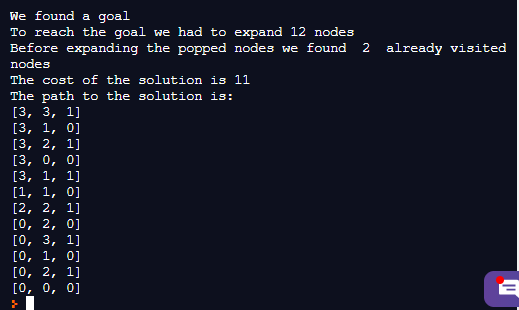


GBFS:

* With trip with returns:



* With trip without returns:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | DFS | BFS | Trip without return | | Trip with return | |
| A\* | Greedy | A\* | Greedy |
| Expanded nodes | 13 | 14 | 13 | 12 | 13 | 12 |
| Already visited | 9 | 14 | 12 | 2 | 11 | 2 |
| Path Cost | 11 | 11 | 11 | 11 | 11 | 11 |

* Trip with returns dominates Trip without returns heuristic because

∀ x>=2: x-1 >= x/2

* We did not notice a big difference in terms of heuristics. It should probably show more if we expand our problem to have more than 6 people in total.
* Greedy showed a significant decrease in the number of loops than A\* even though they both led to an optimal solution which is not surprising because A\* needs to make sure that the path is actually optimal and needs to explore more possibilities in a sense to not select a suboptimal solution. The same pressure is not on Greedy although in this case it found the optimal solution which is not always guaranteed.
* DFS performed slightly better than BFS in terms of expanded nodes and loops since the solution is deep and not close to the root.