Lab 2 – Report

Group 7

Bhavesh Borse(200010005), Eshita Pagare(200010016)

1 Introduction

In this task, Block World Domain had to be implemented. Blocks World Domain Game starts with an initial state consisting of a fixed number of blocks arranged in 3 stacks and we can move only top blocks of the stacks. Blocks World is a planning problem where we know goal state beforehand and path to Goal state is more important. Essentially we have to achieve a goal state that is a particular arrangement of blocks by moving these blocks.

2 Start State & Goal State

The start states and goal states will be given in the input file. However such an example is given in the problem statement and sample I/O. An example of Start State is as below:

Initial State is:

[A]

[B, C]

[D]

Correspondingly, the Goal State: is as below:

Goal State is:

[A, B, C, D]

[]

[]

3 Pseudo Codes

In the following subsections, pseudo codes for important functions in the code are explained.

3.1 **MoveGen**(state)

The function takes a state as input and returns a set of states that are reachable from the input state .

MoveGen(state)

```
nextStates ← ()
for neighbour n of state in order(Heuristic Values) do
    pop.(Queue)
    nextStates.append(n)
return nextStates
```

3.2 **GoalTest**(state)

This function returns state if the input state is goal and otherwise continue.

GoalTest(state)

```
if state == goal_state:     #If the state is the goal state, return it and break.
    count = str(counter)
    print('No of states explored:' + count  )
    return state
```

3.3 Heuristic Functions

3.3.1 Heuristic Function-1

The function counts how many blocks from each state in F are out of place, and returns the index of the state with the minimum score i.e. the smallest number of blocks that are not in the final position.

```
def heuristic_1(F, goal_layout, blocks_keys):

    scores = []     #A list object containing the score of each
state.

    for state in F:
        out_of_place = 0     #Initialize each score to 0.

        for key in blocks_keys:     #For each block...
            if state.layout[key] != goal_layout[key]:     #If
it is not in its final position...
                out_of_place += 1     #Add 1 to score.

        scores.append(out_of_place)

    return scores.index(min(scores))     #Return the index of
the state with the minimun score.
```

3.3.2 Heuristic Function-2

The function is a variation of the heuristic_1 function, with the addition of the distance attribute in the calculation of the scores. It counts how many blocks are not in the final position and adds the total number of steps from the root. So between two states with equal out of place blocks, the heuristic function chooses the one with the smallest distance from the root.

```
def heuristic_2(F, goal_layout, blocks_keys):

    scores = []     #A list object containing the score of each
state.

    for state in F:
        score = 0     #Initialize each score to 0.

        for key in blocks_keys:     #For each block...
            if state.layout[key] != goal_layout[key]:     #If
it is not in its final position...
                score += 1     #Add 1 to score.

        score += state.distance     #Add the distance from the
root to score.

        scores.append(score)

    return scores.index(min(scores))     #Return the index of
the state with the minimun score.
```

3.3.3 Heuristic Function-3

In an intermediate state, depending upon number of matching-positions of blocks with goal state and additionally non-matching positions is used. If block is in same as goal state stack, then the heuristic value will be higher. Essentially, here all matching-positions have weightage assigned as 1000. If the block is in same goal-stack, weightage will be assigned as 100.

```
def heuristic_3(F, goal_layout, blocks_keys):
    all_scores = []
    for state in F:
        score = 0
        for key in blocks_keys:
            if state.layout[key] != goal_layout[key]:
                score = 100
            elif state.layout[key] == goal_layout[key]:
                score = 1000

        all_scores.append(score)

    return all_scores.index(max(all_scores))
```

4 Directions to Run Code

In the Windows Terminal, give the command as follows:

python main.py <method> <heuristic> <input filename> <output filename>

for example:

python Group7_main.py best 0 input.txt output.txt

5 Data - Variation in Heuristic Functions Block World

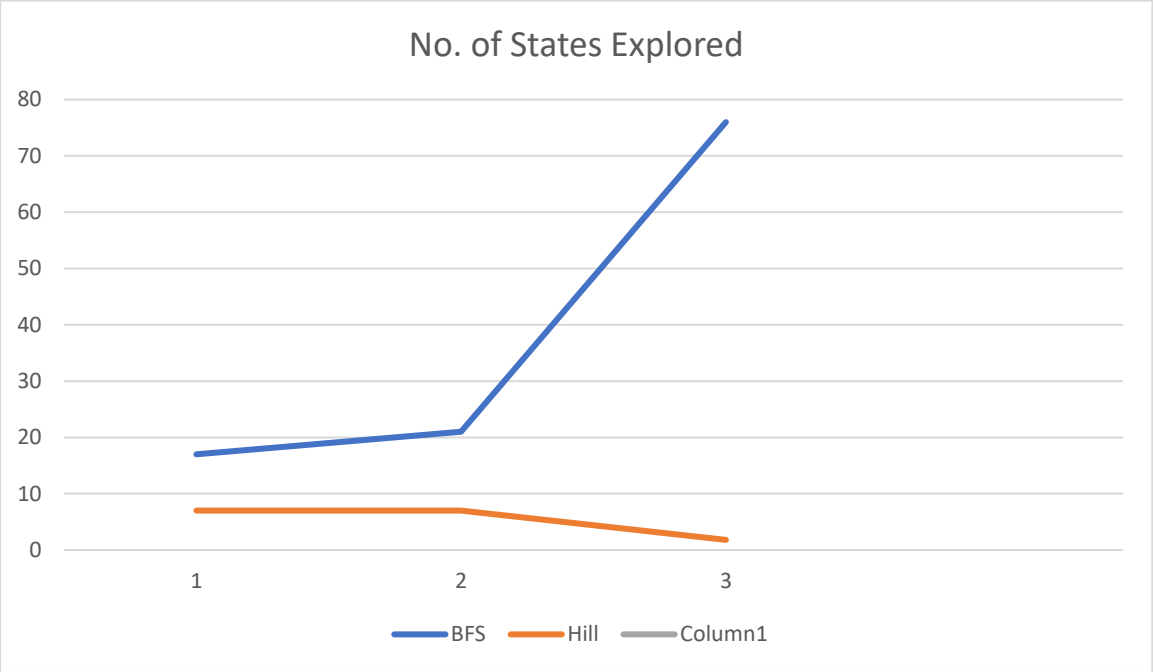
Using 2 different input files, input1.txt & input2.txt, we used BFS and Hill Climbing Algorithms to test across various Heuristic Functions. Statistics related to the output are mentioned as follows in the Table-1. Here Sol in table is Final/Goal State reached or not and Y represents Yes, N represents No. The below Table 1 summarizes the

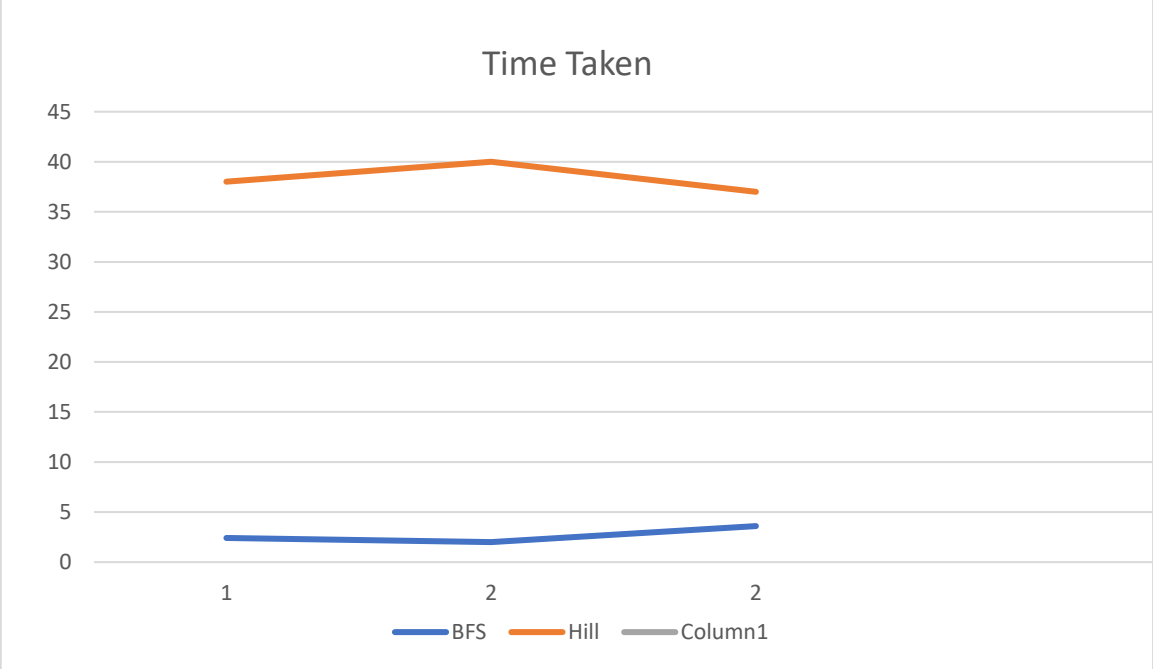
results obtained between Best First Search and Hill Climbing algorithm. As expected, the Hill Climbing algorithm runs faster than Best First Search due to its greedy nature and lesser number of states explored. In the taken 2 examples, we couldn't reach Optimal State in Hill Climbing because of a drawback for the Hill Climbing algorithm found a local maximum and is stuck in that. This is very general that optimal state is not reached in Hill Climbing Algorithm.

6 Graphical Interpretation & Inference

After analysing data from above graph, below graphs in Figure 1 and Figure 2 were plotted for input1.txt & input2.txt respectively. From the graphs, we can see that in general Heuristic Function 3 takes more time and more number of steps as compared to other. Correspondingly, the number of states explored for those cases are also more.

Algorithm	Parameters				
	Heuristic Func	Input File	States Explored	Time Taken(ms)	Sol
BFS	1	Input1.txt	17	2.4	Y
	2	Input1.txt	21	2	Y
	3	Input1.txt	76	3.6	Y
HILL	1	Input1.txt	7	38	N
	2	Input1.txt	7	40	N
	3	Input1.txt	7	37	N
BFS	1	Input2.txt	67	3.2	Y
	2	Input2.txt	79	4	Y
	3	Input2.txt	1852	86	Y
HILL	1	Input2.txt	4	40	N
	2	Input2.txt	4	38	N
	3	Input2.txt	4	43	N


No. of States Explored
80
70
60
50
40
30
20
10
0
1
2
3
BFS
Hill
Column1


Time Taken
45
40
35
30
25
20
15
10
5
0
1
2
2
BFS
Hill
Column1

7 Conclusion

From the results above, it is seen that Best First Search (BFS) always finds an optimal solution with the trade off of time, as it explores all possible N! states in the solution space. Conversely, on the other hand, the Hill Climbing Algorithm, has lesser execution time due recursive greedy selection in iterations but it cannot guarantee an optimal solution in all cases.