CS 314, Lab 2 - Report

Shriram Ghadge (180010015), Rishit Saiya (180010027)

January 22, 2021

1 Abstract

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 Literature

2.1 State Space

The state space is the Euclidean space in which the variables on the axes are the state variables. The state of the system can be represented as a vector within that space. To abstract from the number of inputs, outputs and states, these variables are expressed as vectors. In this problem statement we have used that state space as a stack of the Block World. Since, a state/block has to moved such that it reaches the goal state, the solution space can consist of N! possible exchanges.

2.2 Start State & Goal State

The start states and goal states will be given in the `inputfile`. 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 :
[]
[]
[3, 6, 1, 2, 5, 4]
```

Correspondingly, the **Goal State:** is as below:

```
Goal State is :
[3, 2, 4]
[5]
[6, 1]
```

3 Pseudo Codes

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

3.1 movGen(state)

The function takes a state as input and returns a set of states that are reachable from the input state in one step. [Algorithm 1]

Algorithm 1 movGen(graph)

```
1: procedure MOVEGEN(state)
2:     nextStates ← ()                          ▷ initialize nextStates to empty set
3:     for neighbour n of state in order(HeuristicValues) do
4:         pop.(Queue)                          ▷ Pop the element with highest Heuristic Value
5:         nextStates.append(n)
6:     return nextStates                        ▷ nextStates are required moves generated
```

3.2 GoalTest(state)

This function returns True if the input state is goal and False otherwise. [Algorithm 2]

Algorithm 2 goalTest(state)

```
1: procedure GOALTEST(state)
2:     if state.value == '*' then
3:         return true
4:     return false                             ▷ state is not goal
```

3.3 Heuristic Functions

3.3.1 Heuristic Function-1

In this heuristic function, we implemented the following logic:

In an intermediate state, depending upon the number of matching-positions of blocks with goal state, the heuristic values will be higher for matching blocks. Essentially, here all matching-positions have same weightage assigned as 1000.

```
if(currentState(Block) == goalState(Block)){
   for all blcoks:
        h(Block) += 1000;
}
else
  continue;              //Default value = 0
```

3.3.2 Heuristic Function-2

In this heuristic function, we implemented the following logic:

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.

```
if(currentStack(Block) == goalStack(Block)){
    if(currentState(Block) == goalState(Block)){
        h(Block) += 1000;
    }
    else
        h(Block) += 100;
}
else
  continue;              //Default value = 0
```

3.3.3 Heuristic Function-3

In this heuristic function, we implemented the following logic:

In an intermediate state, if the base of stack matches, then it will check further and gives higher heuristic value otherwise it will skip. That is, for each block matching from base to goal-state will have same weightage assigned as 1000.

```
if(currentBaseStack(Block) == goalBaseStack(Block)){
    h(Block) += 1000;
    increaseBaseStack(Block);
    checkHeuristicValue(Block);
}
else
    continue;              //Default value = 0
```

4 Directions to Run Code

The code `Group12_Lab_2.java` is added in `run.sh` file and correspondingly it will be run as follows:

```
$ ./run.sh input.txt BFS 1 output.txt
```

4.1 Command Line Arguments Usage

In the above, command used to run code has command line arguments. Correspondingly, their information is as below:

1. `input.txt` : Select Input File.
2. `BFS/HillClimbing` : Select one of algorithms - Best First Search/ Hill Climbing.
3. `1/2/3` : Select one of the Heuristic function among 1,2,3
4. `output.txt` : Select Output File.

5 Data - Variation in Heuristic Functions Block World

Using 2 different input files, `input1.txt` & `input2.txt`, we used BFS and HillClimbing Algorithms to test across various Heuristic Functions. Statistics related to the output are mentioned as follows in the Table 1. Here OS in table is Optimal 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 HillClimbing because of a drawback for the HillClimbing algorithm found a local maximum and is stuck in that. This is very general that optimal state is not reached in HillClimbing Algorithm.

6 Graphical Interpretation & Inference

After analyzing 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 2 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 Fn	Input File	No. of States Explored	Time Taken(ms)	OS
BFS	1	input1.txt	27	13	Y
	2	input1.txt	71	22	Y
	3	input1.txt	25	12	Y
HC	1	input1.txt	1	1	N
	2	input1.txt	4	1	N
	3	input1.txt	1	1	N
BFS	1	input2.txt	11	4	Y
	2	input2.txt	216	79	Y
	3	input2.txt	11	4	Y
HC	1	input2.txt	2	1	N
	2	input2.txt	4	3	N
	3	input2.txt	2	1	N

Table 1: Statistics

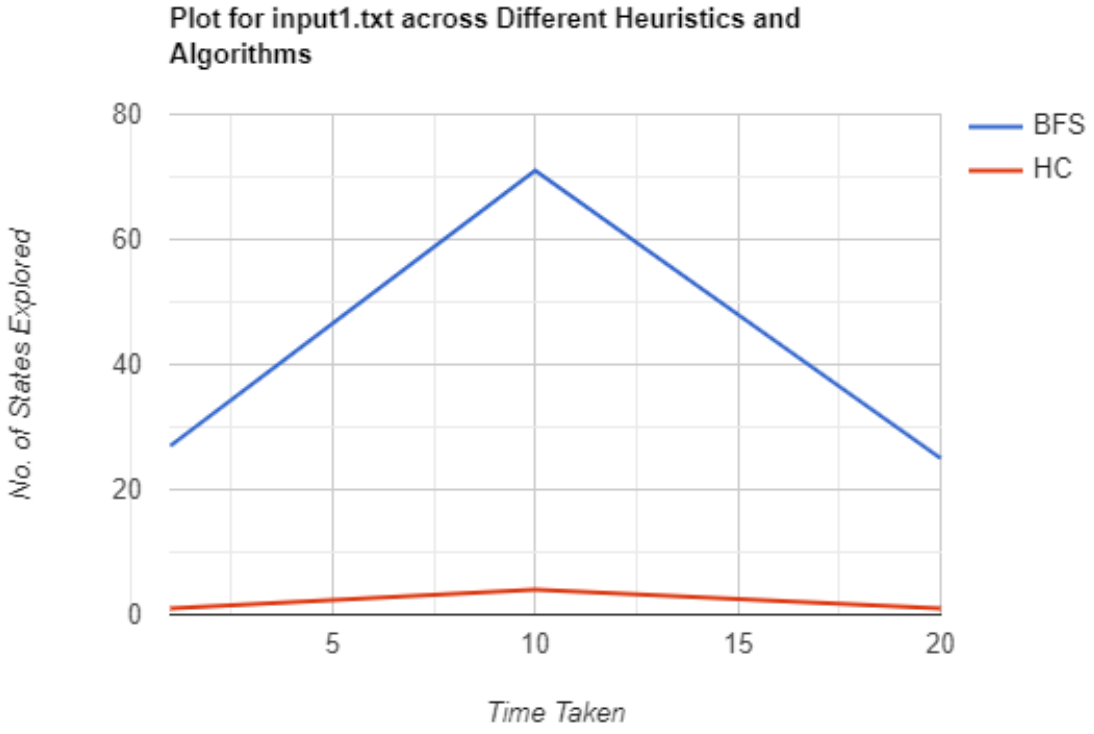

Figure 1: Plot for `input1.txt` across different Heuristics and Algorithms

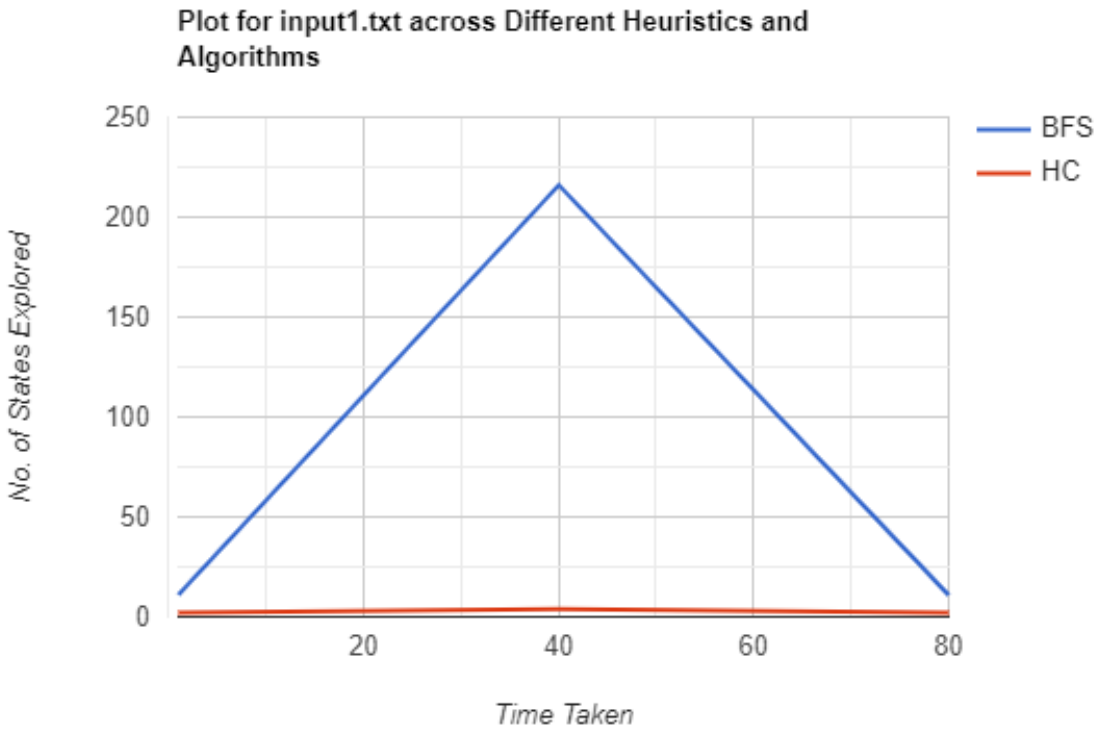

Figure 2: Plot for `input2.txt` across different Heuristics and Algorithms

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.