CS312 - Artificial Intelligence Lab

Assignment 2

Arun R Bhat (190010007), Mohammad Sameer (190010024)

1 Introduction

The goal of this assignment is to simulate the Blocks Word Domain Game using Best-First Search (BFS) and Hill Climbing (HC). The Blocks World Domain Game begins with a given amount of blocks arranged in three stacks, and we can only move the top blocks of the stacks. We must move these blocks to reach a goal state, which is a specific arrangement of blocks. Blocks World is a planning problem in which the goal state is known ahead of time and the pathway to it is more significant.

2 Literature

2.1 State Space

A state space is the set of all possible configurations of a system. In our case the system is the block world, or more specifically the stacks configuration of the block world. The possible configurations are those block worlds or stacks configurations which can be reached in a sequence of steps from a given initial state.

2.2 Start State & Goal State

In our case the start state and goal state are simply the stack configurations. We have represented the start & goal state in an input file, an example of which is as follows:

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

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

3 Psuedo Codes

The section below has pseudo codes of a few important functions.

3.1 moveGen(state)

This function generates and returns the states that can be reached from a given state in one step. Note that at most 6 new neighbours are possible for a given state.

Algorithm 1 moveGen(state)

```
1:  neighbours ← list()                          ▷ initialize neighbours to empty list
2:  if state.stack1 not empty then
3:      neighbour1 ← state.copy()
4:      neighbour1.stack2.push(neighbour1.pop())
5:      neighbour1.setHeuristic()
6:      neighbours.append(neighbour1)
7:
8:      neighbour2 ← state.copy()
9:      neighbour2.stack3.push(neighbour2.pop())
10:     neighbour2.setHeuristic()
11:     neighbours.append(neighbour2)
12: end if
13:
14: if state.stack2 not empty then
15:     neighbour3 ← state.copy()
16:     neighbour3.stack1.push(neighbour3.pop())
17:     neighbour3.setHeuristic()
18:     neighbours.append(neighbour3)
19:
20:     neighbour4 ← state.copy()
21:     neighbour4.stack3.push(neighbour4.pop())
22:     neighbour4.setHeuristic()
23:     neighbours.append(neighbour4)
24: end if
25:
26: if state.stack3 not empty then
27:     neighbour5 ← state.copy()
28:     neighbour5.stack1.push(neighbour5.pop()
29:     neighbour5.setHeuristic()
30:     neighbours.append(neighbour5)
31:
32:     neighbour6 ← state.copy()
33:     neighbour6.stack2.push(neighbour6.pop())
34:     neighbour6.setHeuristic()
35:     neighbours.append(neighbour6)
36: end if
37:
38: return neighbours
```

3.2 goalTest(state, goalState)

This function returns true if the given state is goal state other wise returns false.

Algorithm 2 goalTest(state, goalState)

```
1: if state.stack1 != goalState.stack1 then
2:     return False
3: end if
4: if state.stack2 != goalState.stack2 then
5:     return False
6: end if
7: if state.stack3 != goalState.stack3 then
8:     return False
9: end if
10: return True
```

4 Heuristics Functions

The section below explains the different heuristic functions used.

4.1 Heuristic Function 1

This function assigns a value say 'hValue' to each block, which is made up of 3 components. The hValue has an initial value of 0, and then to this hValue

1. +1 is added if the block belongs to correct stack with respect to goal state. Otherwise nothing is added.

2. +1 is added if the block is at correct height in the stack with respect to goal state. Otherwise nothing is added.

3. +1 is added if the block is sitting on correct block with respect to goal state. Otherwise nothing is added.

Then the hValues of all the blocks are added to form the heuristic value of the state. Note that the heuristic value of the goal state is
$3 * \textit{(total number of blocks)}$.

```
hValue = 0
for block in state.allBlocks():
    label = block.label
        goalBlock = goalState.findBlock(label)
        if(block.stackNo == goalBlock.stackNo):
                hValue += 1
        if(block.blockPos == goalBlock.blockPos):
                hValue += 1
        if(block.sittingOn == goalBlock.sittingOn):
                hValue += 1
state.heuristic = hValue
```

4.2 Heuristic Function 2

This function assigns a value say 'hValue' to each block, which is defined as negative of sum of absolute value of difference of block number with respect to goal state and absolute value of difference of block position with respect to goal state. Or simply,

$$\text{hValue} = -1 * \{|s_{block} - s_{goalBlock}| + |p_{block} - p_{goalBlock}|\}$$

Note that

- s_{block} represent stack number of the block
- $s_{goalBlock}$ represent stack number of block in goal state
- p_{block} represent vertical position of the block in the stack
- $p_{goalBlock}$ represent vertical position of the block in the stack in goal state

Then the hValues of all the blocks are added to form the heuristic value of the state. The reason to take negative of this value is to make a maximization problem. Note that the heuristic value of the goal state is 0.

```
hValue = 0
for block in state.allBlocks():
    label = block.label
    goalBlock = goalState.findBlock(label)
    hValue += -1 * {|block.stackNo - goalBlock.stackNo|
                    +|block.blockPos - goalBlock.blockPos|}
state.heuristic = hValue
```

4.3 Heuristic Function 3

This function assigns a value say 'hValue' to each block, which is adding upon the heuristic value of the previous heuristic function. Defined as trice or twice the negative sum of absolute value of difference of block number with respect to goal state and absolute value of difference of block position with respect to goal state. We take twice if the block is sitting on correct block with respect to goal state otherwise we take trice. Or simply,

$$hValue = \begin{cases} \text{-2*}\{|s_{block} - s_{goalBlock}| + |p_{block} - p_{goalBlock}|\} & \text{if } l_{sitting} != l_{goalBockSitting} \\ \text{-3*}\{|s_{block} - s_{goalBlock}| + |p_{block} - p_{goalBlock}|\} & \text{otherwise} \end{cases}$$

Note that

- s_{block} represent stack number of the block
- $s_{goalBlock}$ represent stack number of block in goal state
- p_{block} represent vertical position of the block in the stack
- $p_{goalBlock}$ represent vertical position of the block in the stack in goal state
- $l_{sitting}$ represent the label of the block on which the given block is sitting

- $l_{goalBockSitting}$ represent the label of the block on which the given block is sitting in goal state

Then the hValues of all the blocks are added to form the heuristic value of the state. Note that the heuristic value of the goal state is 0.

```
hValue = 0
for block in state.allBlocks():
    label = block.label
    goalBlock = goalState.findBlock(label)
    if(block.sittingOn != goalBlock.sittingOn):
        hValue += -3 * {|block.stackNo - goalBlock.stackNo|
                    +|block.blockPos - goalBlock.blockPos|}
    else:
        hValue += -2 * {|block.stackNo - goalBlock.stackNo|
                    +|block.blockPos - goalBlock.blockPos|}
state.heuristic = hValue
```

5 Directions to run code

The code `Group12.py` is added in the zip file `Assignment2_Group12.zip`. To run it use the following command:
`$ python3 <Group12>.py <input.txt> <BFS/HC> <1/2/3> <output.txt>`

Command line arguments are:

- `<Group12>.py` : The code implementing Best First Search and Hill Climbing algorithms along with three heuristic functions.
- `<input.txt>` : Name of the input file used in .txt format.
- `<BFS/HC>` : The algorithm used - BFS stands for Best First Search and HC stands for Hill Climbing.
- `<1/2/3>` : The index of heuristic function used.
- `<output.txt>` : Name of the output file to be obtained on running the code in .txt format.

For ease of running the code and to avoid typing all the above command line arguments, we have made a script file `run.sh` which has the above command line for different cases. To run it use the following command:
`$ bash run.sh`

6 Statistical and Graphical Analysis

The following tables and graphs record number of states explored and time taken to explore them for three different heuristics using different algorithms for two different input files.

Algorithm	Parameters			
	Heuristic Function	No. of States Explored	Time Taken	Solution Found
Best First Search	1	35	75	Yes
	2	44	111	Yes
	3	36	77	Yes
Hill Climbing	1	1	2	No
	2	5	14	No
	3	1	2	No

Table 1: Stats observed using input1.txt file as input

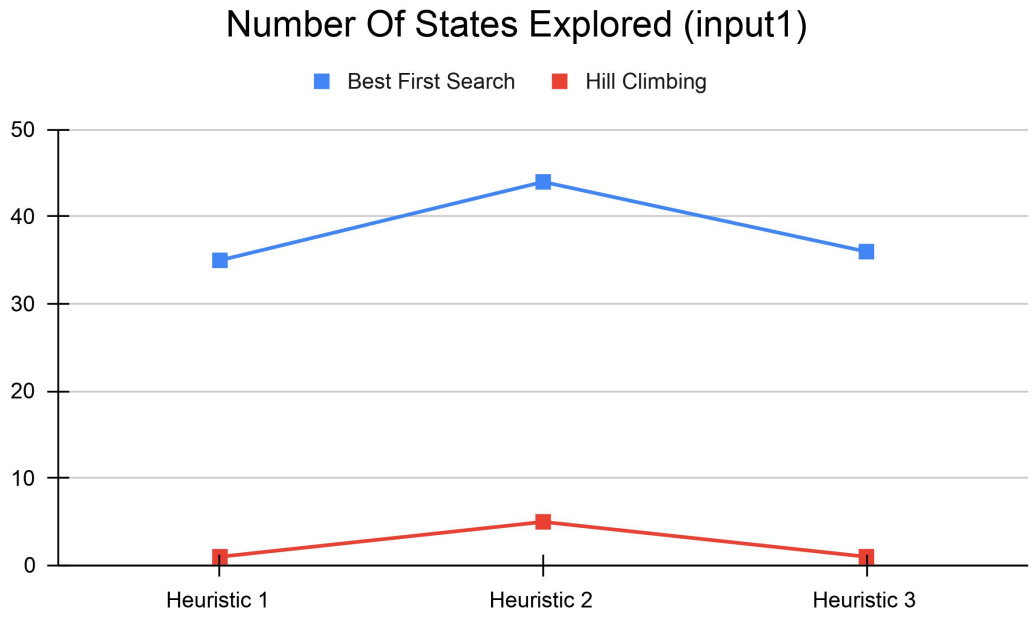

Figure 1: Graph comparing number of states explored by two algorithms when different heuristics are used with input1.txt as input.

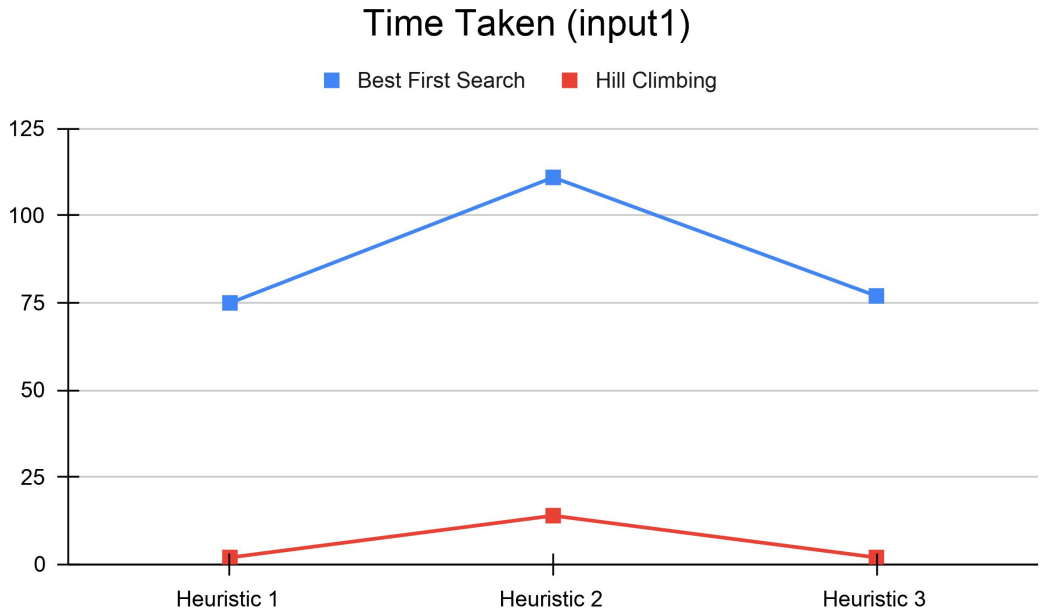

Figure 2: Graph comparing time taken by two algorithms when different heuristics are used with input1.txt as input.

Algorithm	Parameters			
	Heuristic Function	No. of States Explored	Time Taken	Solution Found
Best First Search	1	33	60	Yes
	2	93	306	Yes
	3	117	432	Yes
Hill Climbing	1	4	8	No
	2	4	7	No
	3	4	7	No

Table 2: Stats observed using input2.txt file as input

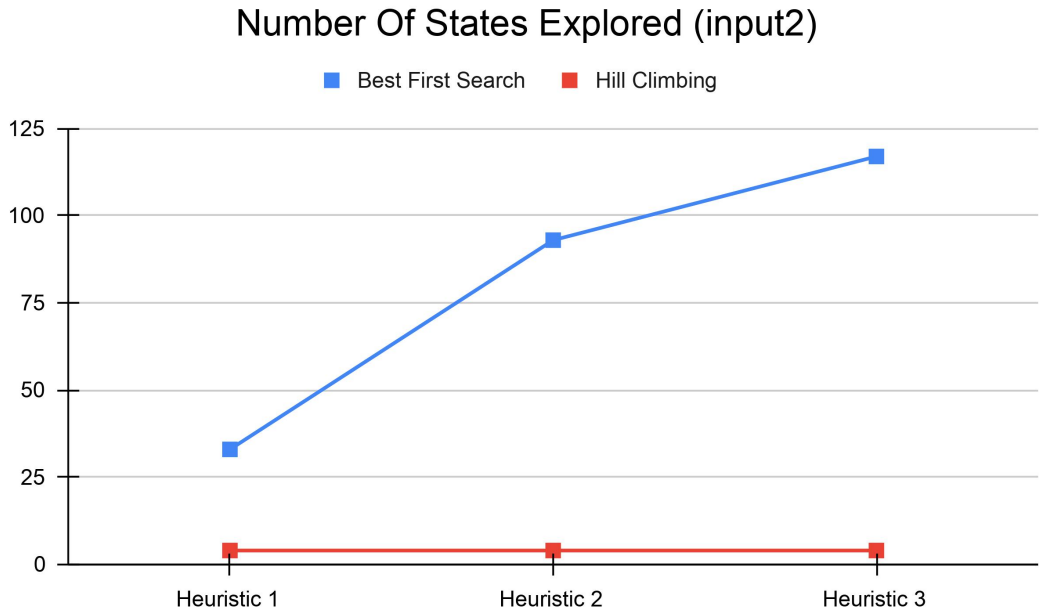

Figure 3: Graph comparing number of states explored by two algorithms when different heuristics are used with input2.txt as input.

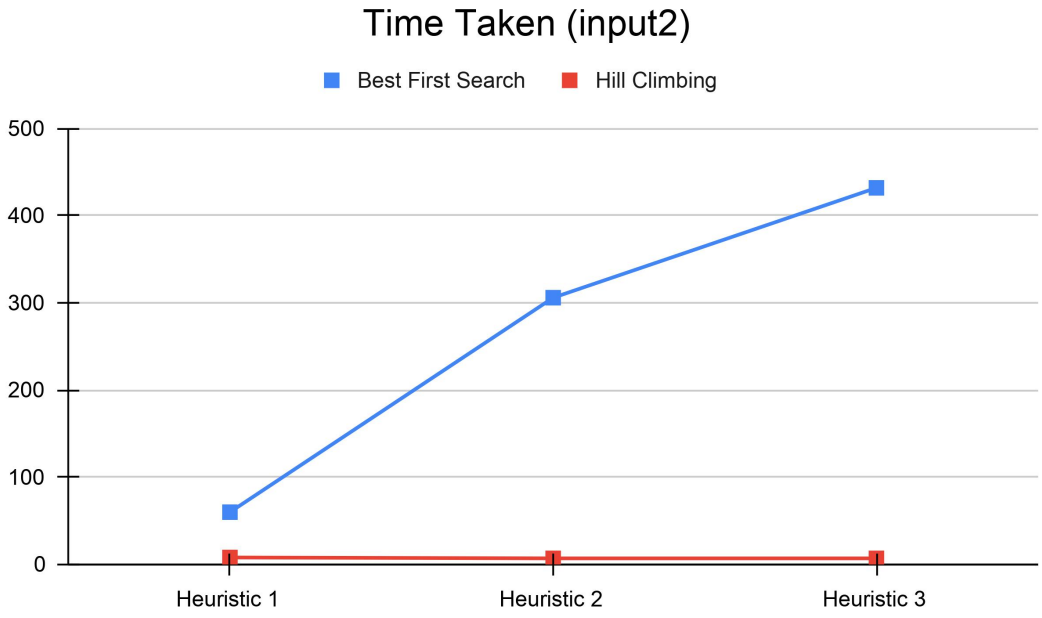

Figure 4: Graph comparing time taken by two algorithms when different heuristics are used with input2.txt as input.

7 Inference and Conclusion

- Best First Search explores far more number of states than Hill Climbing. This is because Best First Search searches exhaustively until the goal state is reached whereas Hill Climbing uses greedy approach and doesn't always reach the goal state.
- Consequently, the time taken by Best First Search is also much higher than that by Hill Climbing. The reason remains same as above.
- We can observe that number of states explored and time taken are greater for heuristic 2 than the other two heuristics for input1 whereas they are greater for heuristic 3 when input2 is used in Best First Search. This shows that selecting a more logical heuristic and experimenting can lead us to the goal state at a faster pace and the optimal heuristic can be different for different inputs. So, using more test cases can give us a better picture on which heuristic is generally the most optimal one.
- Clearly, Hill Climbing is much faster than Best First Search and hence can be tried first. If we can't reach to goal state using it, we can then go for Best First Search that surely finds the goal state as it traverses all N! number of states (in worst case).