CS214-Lab 2 Report

Kushagra Khatwani (190020021), Suryavijoy Kar (190020039)

1. Description of domain

a. **State space:** Combination of all possible states of the system. In our problem, we have considered 6 blocks (denoted by A,B,C,D,E,F) which can be placed in 3 stacks (denoted by 1,2,3).
Only the top block from a stack can be moved to the top of another stack at one time. We have considered a state (a.k.a. node) to be a dictionary where the keys represent the stack number and the values are a list of blocks where the first element of the list is at the top of the stack.

Example of a state/node: {1: ['C', 'D'], 2: ['E', 'F'], 3: ['A', 'B']}.
In stack 1, the blocks from top to bottom are C,D; in stack 2, the blocks from top to bottom are E,F and in stack 3, the blocks from top to bottom are A,B.

b. **Start node and goal node:** We have considered 2 inputs.
input 1 is as follows:
start node: {1: ['A','B','C','D'], 2:['E','F'],3:[]}
goal node: {1:[],2:[],3:['D','C','F','B','E','A']}

input 2 is as follows:
start node: {1: ['A','B','C','D'], 2:['E','F'],3:[]}
goal node: {1: ['A','E','B','C','D'], 2:['F'],3:[]}

c. **Pseudo Codes:** In the following sections, the pseudo codes for important functions in the code are explained

i. **MOVEGEN**

```
def movegen(current_state):
    return neighbours(current_state)
```

ii. **HILLCLIMBING**

```
def hill_climbing(current_state):
    neighbours = sort(heuristic_fn(movegen(current_state)))
    for node in neighbours:
        if node > heuristic_fn(current_state):
            return node
    return []
```

iii. **GOALTEST**

```
def goal_test(currrent_state, goal_state):
        if current_state == goal_state:
            return True
        else:
            return False
```

2. Heuristic Functions:

a. **Heuristic Function 1 :** If any block of a stack from the current state is not in the corresponding block of the same stack from the goal state, we subtract 1 from the cost. If the blocks match, we add 1 to the cost. We loop over all the stacks and check using this logic to find the final cost.

```
def h1( state, goal_state):
    cost = 0
    for pos in goal_state:
        index = -1
        while index >= -len(goal_state[pos]):
            if index < -len(state[pos]) or len(state[pos]) == 0:
                break
            if goal_state[pos][index] == state[pos][index]:
                cost += 1
            else:
                cost -= 1
            index -= 1

        while index >= -len(goal_state[pos]):
            cost -= 1
            index -= 1
    return cost
```

b. **Heuristic Function 2 :** If any block of a stack from the current state is not in the corresponding block of the same stack from the goal state, we subtract the height of the block in that stack from the cost. If the blocks match, we add the height of the block in that stack to the cost. We loop over all the stacks and check using this logic to find the final cost.

```
def h2(state, goal_state):
    cost = 0
    for pos in goal_state:
        index = -1
        while index >= -len(state[pos]):
            if index < -len(goal_state[pos]) or len(goal_state[pos]) == 0:
                break
            if self.goal_state[pos][index] == state[pos][index]:
                cost += abs(index)
            else:
                cost -= abs(index)
            index -= 1

        while index >= -len(state[pos]):
            cost -= abs(index)
            index -= 1
    return cost
```

c. **Heuristic Function 3 :** If the configuration of an entire pile on which a block is resting on is correct, 1 is added to the cost for every block in that pile or else 1 is subtracted from the cost for every block in that pile. This logic looks at the correctness of the relative position of the block rather than the individual position.

```
def h3( state, goal_state):
    cost = 0
    for pos in goal_state:
        index = -1
        while index >= -len(state[pos]):
            if index < -len(goal_state[pos]) or len(goal_state[pos]) == 0:
                break
            if self.goal_state[pos][index:] == state[pos][index:]:
                cost += abs(index)-1
            else:
                cost -= abs(index)-1
            index -= 1

        while index >= -len(state[pos]):
            cost -= abs(index)-1
            index -= 1
    return cost
```

3. Directions to run code (also available in readme.txt):

Use python version 3.9.7 to run the file.
The file "1.py" can be run from command line using python3 as follows:

```
python3 1.py -i input_number -hf heuristic_func_number
```

Command line arguments:

a. -i input_number : Select one of the input configurations (start and goal node) from 1/2.
b. -hf heuristic_func_number : Select one of the Heuristic functions from 1/2/3.

4. Results - Hill Climbing:

Using 2 different input configurations, we use Hill Climbing Algorithm to test across various Heuristic Functions. Statistics related to the output are mentioned in Table 1.
We note down the *Number of States Explored*, *Time Taken* and *Optimality of Solution* of the Hill Climbing Algorithm for different Heuristic Functions.

Table 1

Input File	Heuristic Function	No. of States Explored	Time Taken (ms)	Optimal Solution
input1.txt	1	7	1.263	Yes
	2	4	1.134	No
	3	4	1.133	No
input2.txt	1	1	0.902	No
	2	5	1.178	Yes
	3	5	1.186	Yes

5. Conclusions:

Our conclusions are as follows:

- Hill-Climbing algorithm is completely greedy in nature so it takes less time.

- If the heuristic function is convex in nature then hill-climbing leads to an optimal solution for the problem otherwise the algorithm may get stuck in a local optimum and give a suboptimal solution.
- Lesser number of nodes are explored in the case of hill-climbing algorithm.