

Mini-Project 2: Block World

1. Problem Statement:

In this mini-project, it is required to write a code for an agent that can solve Block World problem for an arbitrary initial arrangement of blocks. The task is to achieve a goal arrangement of blocks, given an initial arrangement of blocks and return a list of moves that will transform the initial state into the goal state.

The blocks will be identified as single letters from A to Z. So, there can be at maximum 26 blocks in an initial arbitrary arrangement of block. An example arrangement is shown in the figure below:



Figure 1: Example of an Initial State and Goal State of Blocks

2. Rules and Valid Moves:

The agent can pick and place a block in the following ways:

1. A block can be picked if no other block is placed on top of it i.e., only the block on top of a pile can be picked.
2. A block can be place either on top of another block or on the table.

3. Representation of Block Configuration in Python Code:

A configuration or arrangement of blocks will be represented by lists of lists of characters, where each character represents a different block. For example, an initial and goal state of blocks as shown in Figure 1 will be represented in the following manner:

3.1 Initial State:

In the initial state, the pile 'C', 'B', 'D', where 'C' is on table and 'D' is on top, is represented by a list ['C', 'B', 'D'] and the block 'A' which is on table is represented as ['A']. Thus, the initial state is a list that contains two lists:

```
[[ 'C', 'B', 'D'], ['A']]
```

3.2 Goal State:

In the goal state, there is only one pile 'D', 'C', 'B', 'A', where 'D' is on table and 'A' is on top. This is represented by a list ['D', 'C', 'B', 'A']. Thus, the initial state is a list that contains only one list:

```
[[ 'D', 'C', 'B', 'A']]
```

4. Representation of Block Moves:

In order to achieve the goal state, any move will be represented by a 2-tuple pair. For example, in Figure 1, if the agent moves block 'D' from the initial state and places it on table, it will be shown as (0, 'D'). The table is represented by 0. Logically speaking, the agent will then place block 'B' on table, (0, 'B'). Then 'C' will be placed on 'D' and it will be represented as ('D', 'C'). Block 'B' will then be picked and placed on top of 'C', ('C', 'B'). The last move will be to place 'A' on top of 'B', ('B', 'A'). So, the complete list of moves will be represented as list of 2-tuple pairs:

```
[(0, 'D'), (0, 'B'), ('D', 'C'), ('C', 'B'), ('B', 'A')]
```

5. How Our agent works?

We have to select a method that can work in an optimal manner. This means we have to continuously upgrade our state until we achieve the goal state. To achieve this goal, we have

chosen the Hill Climbing Algorithm. The objective of hill-climbing algorithm is to attain an optimal state that is an upgrade of the existing state. The algorithm achieves this through nodes, where a node comprises of two parts: state and value. We start with a non-optimal state and upgrade this state when a certain precondition is met. A heuristic function is used as the basis for this precondition. The hill-climbing algorithm has the following features:

1. **Assign Heuristic Value:** A heuristic value is assigned to the goal and the initial/current state.
2. **Generate & Test Technique:** The algorithm generates all the possible or candidate states that can be achieved from the current state (see section 2: **Rules and Valid Moves**) and assigns a heuristic value to each of the possible states. Then, a winner state is selected from amongst the candidate/possible states on the basis of heuristic value.
3. **No Backtracking:** The hill-climbing algorithm only works on the current state and future state. There is no backtracking.
4. The process will continue until the goal state is achieved.

6. Example:

6.1 Assign Heuristic Value:

6.1.1 Heuristic Value for Goal State:

Let us consider the example in figure 1. First, we will assign a heuristic value to the goal state. The goal state is the desired block arrangement and we will assign a positive value to each block based on the number of blocks below that block. For example, block 'A' is on top of the pile, there are 3 blocks under 'A', so it will be assigned +3. Similarly, there are two blocks under block 'B' so it will be assigned +2 and so on. The block 'D' is on table so it will be assigned 0. The total heuristic value for goal state will be $h = +6$ ($3+2+1+0$), see figure 2.

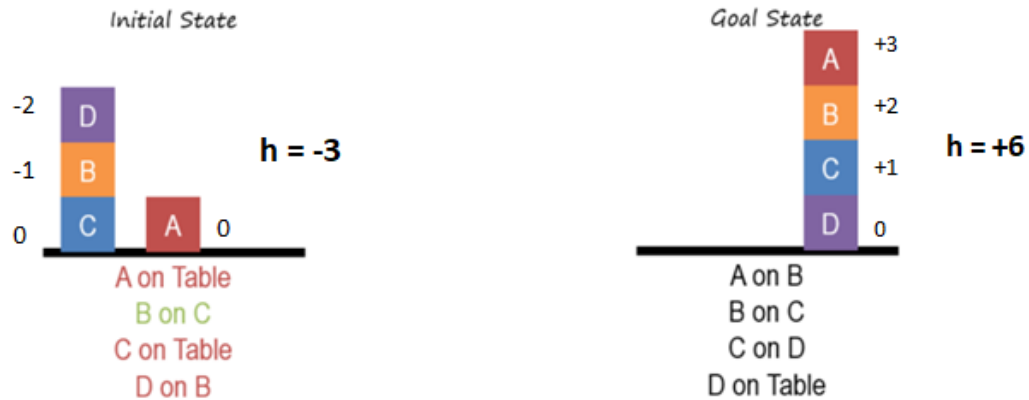


Figure 2: Assign Heuristic Values

6.1.2 Heuristic Value for Initial/Current State:

In the initial or current state, we will assign negative values to any wrong pile or support structure. For example, in figure 2, the block 'D' is on top of 'B' and 'C', so there are two blocks wrongly places under 'D' (compare it with goal state, 'D' should be on table). So, we will assign -2 to block 'D'. Similarly, consider block 'B', block 'C' is placed under 'B' and 'C' is on table, which is wrong (again see goal state, 'C' is under 'B' but it is not on the table, 'D' is under 'C'). So, we will assign -1 to block 'C'. It is very important to mention here that we cannot assign a positive value to a block based on its correct immediate block (as in the case of 'C' under 'B'), we have to consider the complete pile or structure, otherwise there is a chance of the algorithm getting stuck in a local maximum.

6.2 Generate and Test

After assigning the heuristic values to the initial and goal state, the agent will generate all the possible/candidate states following the rules mentioned under the section 2. Then all the candidate states will be assigned a heuristic value based on the discussion under the section 6.1.2. The winner

candidate will be the one which has the closest value to the goal heuristic value. In figure 3, the candidate with $h = -1$:

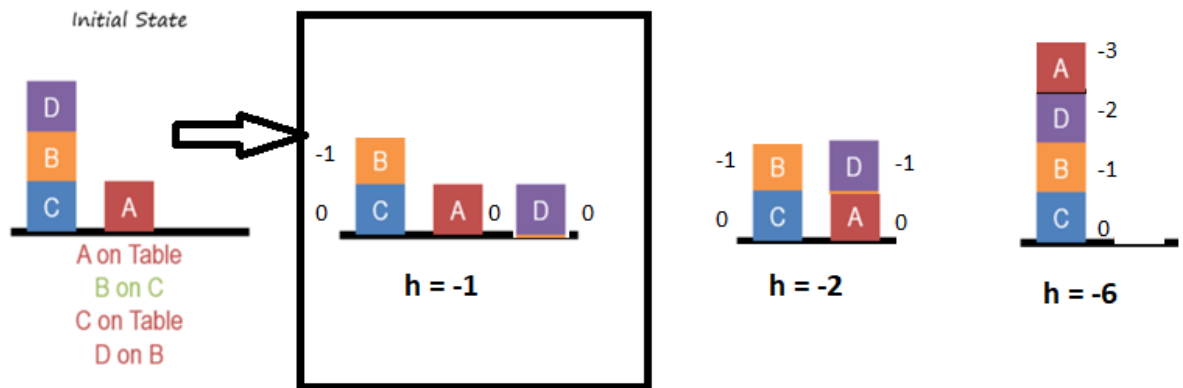


Figure 3: Possible States, their Heuristic Values and the Winner Candidate

This process will continue until we achieve the goal state. The next series of figures will show some examples of how the transformation of blocks to goal state.

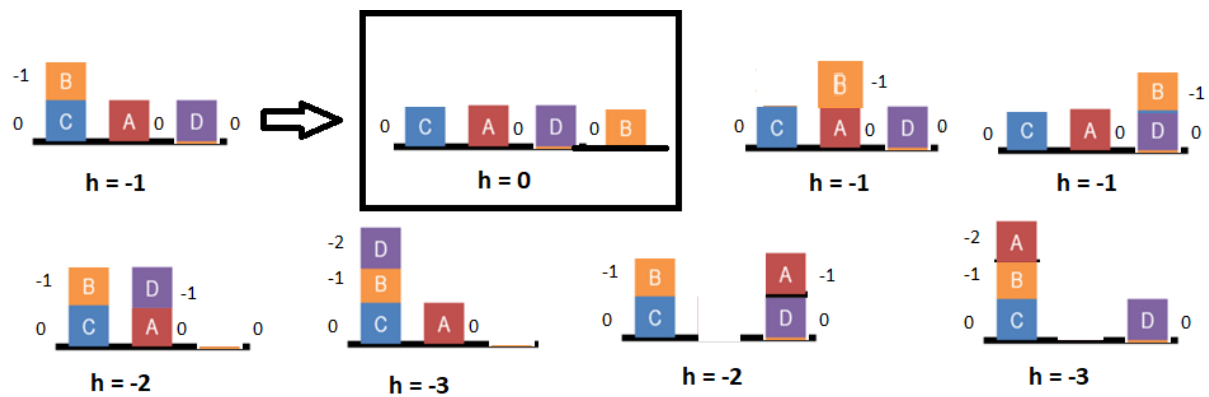


Figure 4: Possible States, their Heuristic Values and the Winner Candidate

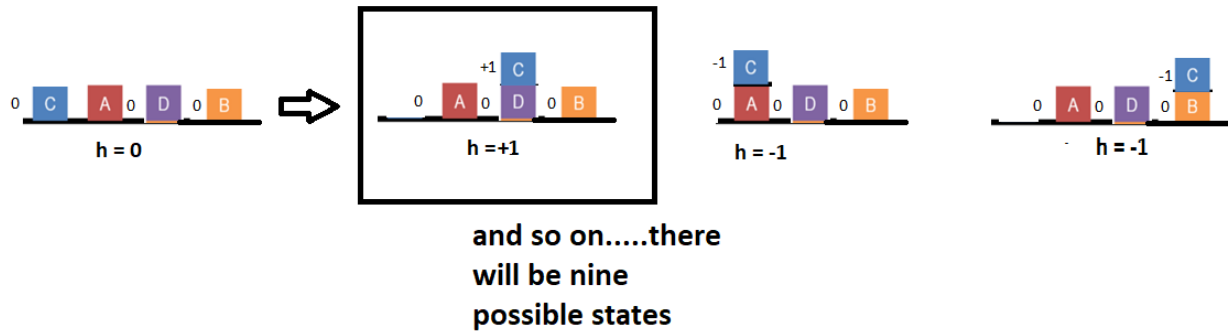


Figure 5: Possible States, their Heuristic Values and the Winner Candidate

7. Python Code:

The code written in python consists of two files: 1. **blockworldAgent.py** and 2. **main_blockworld.py**.

7.1 blockworldAgent.py

This file contains various functions, the details of some of the functions are as follows:

calc_goal_heuristic(state): This function calculates the heuristic value of the goal state. It takes input a variable called 'state' and returns back its heuristic value. The heuristic value is calculated based on the discussion in section 6.1.1. A snippet of the function is shown in the figure 6. We have already discussed in section 4 that a state is represented as list of lists. So, in our code we use for loop to iterate through the state variable. In order to know the number of blocks in the pile or number of elements in the list we use the 'len' function. Then we use 'range' and 'sum' function to calculate the heuristic value of each pile or list and save it in h_counter. The total heuristic value for all the block piles is added in heu.

```
def calc_goal_heuristic(state):
    l = len(state)
    h_counter = 0
    heu = 0

    for lst in state:
        l = len(lst)
        h_counter = sum(range(l))
        heu = heu + h_counter

    return heu
```

Figure 6: calc_goal_heuristic

calc_state_heuristic(state,goal): This function calculates the heuristic value of the current/candidate state. It takes in two inputs: current state (the variable state) and the goal state (the variable goal), it returns the heuristic value of the current state. The heuristic value is calculated based on the discussion in section 6.1.1. A snippet of the function is shown in figure 7.

```
def calc_state_heuristic(state,goal):
    h_counter = 0
    heu = 0
    unmatched = 0

    for lst_state in state:
        for lst_goal in goal:
            match_index_of_state = [i for i, el in enumerate(lst_state) if el in lst_goal]

            if lst_state != lst_goal[:len(match_index_of_state)]:
                unmatched+=1

            elif lst_state == lst_goal[:len(match_index_of_state)]:
                unmatched-=1

        if unmatched == len(goal):
            h_counter = -1*sum(range(len(lst_state)))
            heu = heu + h_counter
            unmatched = 0

        elif unmatched < len(goal):
            h_counter = sum(range(len(lst_state)))
            heu = heu + h_counter
            unmatched = 0

    return heu
```

Figure 7: calc_state_heuristic

The function uses for loop to iterate through all the lists in the current state and compares it with the lists of goal state. The correct pile or structure is assigned the positive values and the wrong structure is assigned the negative values as discussed earlier in section 6.1.2.

block_moves(curr_state,goal) : This function generates all the possible states from the current state and uses the **calc_state_heuristic(state,goal)** function to calculate the heuristic values of all the possible/candidate states. The function returns all the possible states and their respective heuristic values. It generates the possible/states by following the rules mentioned in section 2. A snippet of the function is shown in figure 8.

```
def block_moves(curr_state,goal):

    len_curr_state = len(curr_state) #total list(s) of blocks
    temp_heuristic_values = []
    temp = []
    temp_state = []

    for lst in curr_state:

        for i in range(len_curr_state):

            lst_index = curr_state.index(lst)

            if i == lst_index:

                len_lst = len(lst)
                temp = copy.deepcopy(curr_state)
                temp[lst_index].pop() #pick the block
                place = lst[len_lst-1]
                temp.append([place]) #place the block on table
                temp = [ele for ele in temp if ele != []]
                temp_state.append(temp)
                temp_heuristic = calc_state_heuristic(temp,goal)
                temp_heuristic_values.append(temp_heuristic)

            else:

                len_lst = len(lst)
                temp = copy.deepcopy(curr_state)
                temp[lst_index].pop() #pick the block
                place = lst[len_lst-1]
                temp[i].append(place) #place the block on another block
                temp = [ele for ele in temp if ele != []]
                temp_state.append(temp)
```

Figure 8: block_moves(curr_state,goal)

create_tuples(state): This function converts the current state into list of 2-tuple pairs. A snippet of the function is shown in the figure 9. If a block say 'C' is placed on table, it's 2-tuple pair is shown as (0, 'C') and if a block say 'B' is placed on block 'D', it is shown as ('D', 'B').


```

def create_tuples(state):
    tuples = []
    temp = []

    for lst in state:
        if len(lst) == 1:
            temp.append((0, lst[0])) #the block is on the table

        else:
            for i in (range(len(lst)-1)):
                temp.append((lst[i], lst[i+1])) #the block is on another block

            tuples.append(temp)
            temp = []

    return tuples

```

Figure 9: create_tuples(state)

difference(old_state,new_state): This function takes in the input old state and the new state. Both inputs are in the form of lists of 2-tuple pair. The function calculates the difference between the two states. The difference of the two states will give us the move that has generated the new state. A snippet of the function is shown in the figure 10.

```

def difference(old_state,new_state):
    diff = [element for element in new_state if element not in old_state]
    return diff

```

Figure 10: difference(old_state,new_state)

7.2 main blockworld.py

This file imports all the modules from the **blockworldAgent.py** file using the following commands:

```

from blockworldAgent import block_moves
from blockworldAgent import calc_goal_heuristic
from blockworldAgent import create_tuples
from blockworldAgent import difference

```

An initial state and the goal state is initiated in the file e.g.:

```
initial = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"]]
```

```
goal = [["H", "E", "F", "A", "C"], ["B", "D"], ["G", "I"]]
```

To calculate the heuristic value of the goal state, **calc_goal_heuristic** function is used. Then a while loop is used to generate all the possible block moves from the current state, calculate their respective heuristic values, select from them the best candidate based on their heuristic values, find the difference between the new state and the old state to know the move that has generated the new state. The while loop will break once the heuristic value of the current state equals the heuristic value of the goal state:

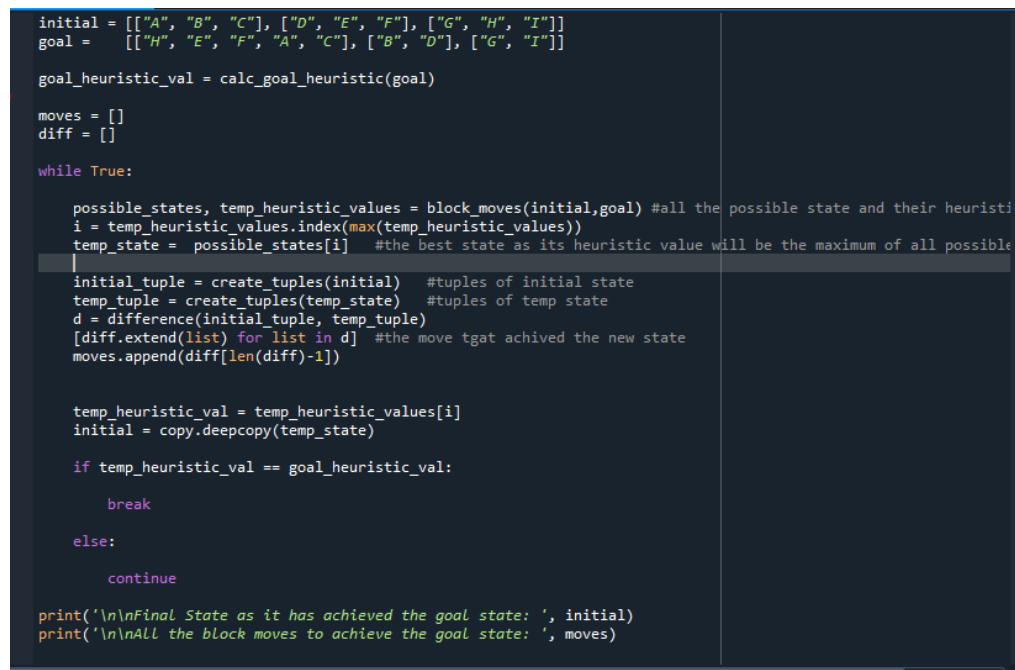
```
if temp_heuristic_val == goal_heuristic_val:
```

```
    break
```

```
else:
```

```
    continue
```

A snippet of the file **main_blockworld.py** is shown in figure 11.



```
initial = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"]]
goal =    [["H", "E", "F", "A", "C"], ["B", "D"], ["G", "I"]]

goal_heuristic_val = calc_goal_heuristic(goal)

moves = []
diff = []

while True:

    possible_states, temp_heuristic_values = block_moves(initial,goal) #all the possible state and their heuristic values
    i = temp_heuristic_values.index(max(temp_heuristic_values))
    temp_state = possible_states[i] #the best state as its heuristic value will be the maximum of all possible states

    initial_tuple = create_tuples(initial) #tuples of initial state
    temp_tuple = create_tuples(temp_state) #tuples of temp state
    d = difference(initial_tuple, temp_tuple)
    [diff.extend(list) for list in d] #the move that achieved the new state
    moves.append(diff[len(diff)-1])

    temp_heuristic_val = temp_heuristic_values[i]
    initial = copy.deepcopy(temp_state)

    if temp_heuristic_val == goal_heuristic_val:

        break

    else:

        continue

print('\n\nFinal State as it has achieved the goal state: ', initial)
print('\n\nAll the block moves to achieve the goal state: ', moves)
```

Figure 11: main_blockworld.py

8. Questions:

- Q. How well does your agent perform?

Ans: The agent does not get stuck at any point; it can start from any particular state and reach the goal state. There are 26 letters in the alphabet and so we can have 26 different blocks.

- Q. How efficient is your agent?

It can handle all the blocks in the alphabet and the number of blocks does not affect its performance.

- Q. Does your agent do anything particularly clever to try to arrive at an answer more efficiently?

Hill climbing algorithm is an efficient technique that works on generate and test method. So, it will achieve the goal in an incremental and optimal way.

- How does your agent compare to a human?

Artificial Intelligence has not achieved a status where we can equate it with the human intelligence but if we consider the hill climbing algorithm, it tries to mimic the human behavior and achieves great results. This will help when the task is huge. Computers are a clear example of how complex operations are done in nano seconds yet they are designed by humans. No human can do such complex computations in just nano seconds. Similarly, if we are able to train our agent or robot/machine to mimic certain human behaviors we can achieve great results.