Block World Hill Climbing code:

import random

class BlockWorldHillClimbing:

def \_\_init\_\_(self, initial\_state, goal\_state):

self.current\_state = initial\_state

self.goal\_state = goal\_state

def heuristic(self, state):

"""

Heuristic function: calculates the number of blocks in the correct positions

relative to the goal state.

"""

score = 0

for i, stack in enumerate(state):

if i >= len(self.goal\_state):

break

goal\_stack = self.goal\_state[i]

for j, block in enumerate(stack):

if j >= len(goal\_stack) or block != goal\_stack[j]:

break

score += 1

return score

def generate\_neighbors(self, state):

"""

Generate all possible next states by moving one block to another stack.

"""

neighbors = []

for i, stack in enumerate(state):

if not stack:

continue # Skip empty stacks

for j in range(len(state)):

if i != j: # Move block to a different stack

new\_state = [list(s) for s in state] # Deep copy of state

block = new\_state[i].pop() # Remove block from stack i

new\_state[j].append(block) # Add block to stack j

neighbors.append(new\_state)

return neighbors

def hill\_climbing(self):

"""

Perform the hill climbing search.

"""

while True:

# Generate neighbors and evaluate them

neighbors = self.generate\_neighbors(self.current\_state)

if not neighbors:

break

# Find the best neighbor based on heuristic

best\_neighbor = max(neighbors, key=self.heuristic)

if self.heuristic(best\_neighbor) <= self.heuristic(self.current\_state):

# Stop if no improvement is possible

break

# Move to the best neighbor

self.current\_state = best\_neighbor

return self.current\_state

def random\_restart(self, max\_restarts=10):

"""

Perform hill climbing with random restarts to avoid getting stuck in local optima.

"""

best\_state = self.current\_state

best\_score = self.heuristic(best\_state)

for \_ in range(max\_restarts):

current\_solution = self.hill\_climbing()

current\_score = self.heuristic(current\_solution)

if current\_score > best\_score:

best\_state = current\_solution

best\_score = current\_score

# Randomize the state for the next restart

self.current\_state = self.randomize\_state()

return best\_state

def randomize\_state(self):

"""

Randomly shuffle the blocks to create a new state.

"""

all\_blocks = [block for stack in self.current\_state for block in stack]

random.shuffle(all\_blocks)

num\_stacks = len(self.current\_state)

new\_state = [[] for \_ in range(num\_stacks)]

for i, block in enumerate(all\_blocks):

new\_state[i % num\_stacks].append(block)

return new\_state

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

initial\_state = [["A", "B"], ["C"], []]

goal\_state = [["A"], ["B"], ["C"]]

solver = BlockWorldHillClimbing(initial\_state, goal\_state)

solution = solver.random\_restart(max\_restarts=5)

print("Final solution:", solution)

Output:

