AI

ASSIGNMENT-4

Kishan Pandey

CS12

102016114

Q1: Solve the following blocks world problem using Simple Hill Climbing Algorithm.

Let the heuristic be

+1 if the block is resting on the correct block and

-1 if it is resting on the incorrect block



**CODE:**

import copy

class kishan\_pandey:

    def \_\_init\_\_(self, start, goal):  # constructor gets automatically called when object calls class

        self.currentState = start     # start state initialise the parameters

        self.goalState = goal         # goal state

        self.prevState = None         # initially no previous zstate

    def isGoalReached(self):         # checks if goal reached or not

        for i in range(0, 4):

            if self.currentState[i] == goal:    # checks if current state is equal to goal state

                return True

        return False   # else false if current state bot equal to goal state

    def kp\_102016114(self):    # display current states

        for i in range(0, 4):

            if self.currentState[i] != []:    # checks if start state is not empty

                print(f"Stack {i}:")

                print(self.currentState[i])

                print("------------------")

                print('xxxxxxxxxxxxxxxxxxxxxxxxxx')

    def \_eq\_(self,other):   # objects of class self,others

        return self.currentState == other.currentState

    def move\_StackX\_StackY(self, x, y):  # movement of blocks

        if self.currentState[x] != [] and len(self.currentState[y]) != 4:

            self.prevState = copy.deepcopy(self)

            block = self.currentState[x].pop()

            self.currentState[y].append(block)

            return True

        else:

            return False

    def possibleNextStates(self):  # generates possible next states

        stateList = []

        for i in range(0, 4):

            for j in range(0, 4):

                copy\_state = copy.deepcopy(self)   # changes made in object not reflected on original object

                if i != j and copy\_state.move\_StackX\_StackY(i, j):

                    stateList.append(copy\_state)

        return stateList

    def heuristic(self):  # calculates heuristic value

        value = 0

        for i in range(0, 4):

            if self.currentState[i] != []:  # checks current state not equal to empty

                if self.currentState[i][0] == self.goalState[0]:   # checking if block lies on currect block

                    value += 1

                else:

                    value -= 1

        for i in range(0, 4):

            goalBlock = self.goalState[i]

            goalBlockIndex = i

            for j in range(0, 4):

                flag = 0

                for k in range(0, len(self.currentState[j])):

                    if self.currentState[j] != []:

                        if self.currentState[j][k] == goalBlock:

                            currentBlockIndexX = j

                            currentBlockIndexY = k

                            flag = 1

                            break

                if flag == 1:

                    flag = 0

                    break

            if self.currentState[currentBlockIndexX][currentBlockIndexY - 1] == self.goalState[goalBlockIndex - 1] and currentBlockIndexY != 0 and goalBlockIndex != 0:

                value += 1

            else:

                if currentBlockIndexY != 0:

                    value -= 1

        return value

def constructPath(goalState):   # shows path traversed from start to end

    print("Displaying path from start to goal")

    while goalState:

        goalState.kp\_102016114()

        goalState = goalState.prevState

    return 1

def HillClimbing(startState):

    open = []   #intilalise empty lists

    closed = []

    # Step 1    insert all elements of intial into open

    open.append(startState)

    # Step 2

    returnVal = 0

    while open:   # untill true(not empty), loop runs

        thisState = open.pop(0)   # removes element at 0th index from list

        thisState.kp\_102016114()

        # Step 4

        if thisState.isGoalReached():   # checks if current state goal state

            print("Goal State FOUND.. STOP SEARCHING...")

            returnVal = constructPath(thisState)

            break

        # Step 5    Generate next possible states

        nextStates = thisState.possibleNextStates()

        # Step 6

        for eachState in nextStates:

            if eachState not in open and eachState not in closed:

                # If next state is better than current state (higher heuristic value is better)

                if eachState.heuristic() > thisState.heuristic():

                    open.append(eachState)

                    closed.append(thisState)

    if returnVal != 1:

        print("ERROR !! LOCAL MAXIMA")

# start of code    where :  1 --> A , 2--> B , 3--> C , 4 --> D

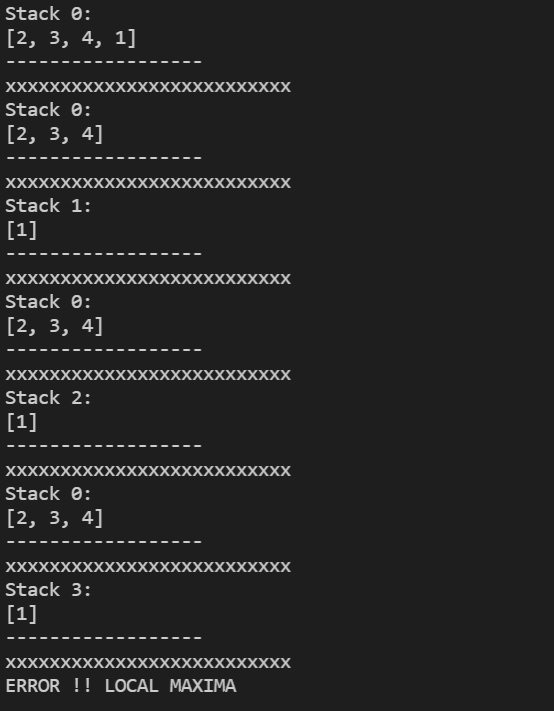
start = [[2, 3, 4, 1], [], [], []]      # start state

goal = [1, 2, 3, 4]                     # goal state

problem = kishan\_pandey(start,goal)     #  object of class kishan\_pandey

HillClimbing(problem)                   # passing object "problem" as parameter to function HillClimbing

**OUTPUT:**



1. Solve the following blocks world problem using Simple Hill Climbing Algorithm.

Take the following heuristic function:

+n for block which is resting on the current support structure and n is equal to number

of blocks below it.

-n for block which is resting on the incurrent support structure and n is equal to

number of blocks below it.



import copy

class kishan\_pandey:

    def \_\_init\_\_(self, start, goal):

        self.currentState = start      # start state initialise the parameters

        self.goalState = goal          # goal state

        self.prevState = None

    def isGoalReached(self):          # checks if goal reached or not

        for i in range(0, 4):

            if self.currentState[i] == goal:      # checks if current state is equal to goal state

                return True

        return False

    def kp\_102016114(self):     # display current states

        for i in range(0, 4):

            if self.currentState[i] != []:    # checks if start state is not empty

                print(f"Stack {i}:")

                print(self.currentState[i])

                print("------------------")

        print('xxxxxxxxxxxxxxxxxxxxxxxxxx')

    def \_eq\_(self, other):            # objects of class self,others

        return self.currentState == other.currentState

    def move\_StackX\_StackY(self, x, y):    # movement of blocks

        if self.currentState[x] != [] and len(self.currentState[y]) != 4:

            self.prevState = copy.deepcopy(self)            # changes made in object not reflected on original object

            block = self.currentState[x].pop()

            self.currentState[y].append(block)

            return True

        else:

            return False

    def possibleNextStates(self):     # generates possible next states

        stateList = []

        for i in range(0, 4):

            for j in range(0, 4):

                copy\_state = copy.deepcopy(self)

                if i != j and copy\_state.move\_StackX\_StackY(i, j):

                    stateList.append(copy\_state)

        return stateList

    def heuristic(self):   # calculate heuristic value

        value = 0

        for i in range(0, 4):

            goalBlock = self.goalState[i]

            goalBlockIndex = i

            for j in range(0, 4):

                flag = 0

                if self.currentState[j] != []:

                    for k in range(0, len(self.currentState[j])):

                        if self.currentState[j][k] == goalBlock:

                            currentBlockIndexX = j

                            currentBlockIndexY = k

                            flag = 1

                            break

                if flag == 1:

                    flag = 0

                    break

            if currentBlockIndexY != goalBlockIndex:

                value -= currentBlockIndexY

            else:

                value += currentBlockIndexY

        return value

def constructPath(goalState):   # shows path from start to end

    print("Displaying path from Start --> Goal")

    while goalState:

        goalState.kp\_102016114()

        goalState = goalState.prevState

    return 1

def HillClimbing(startState):

    open = []    # initialise empty list

    closed = []

    # Step 1    insert all elements of intial into open

    open.append(startState)

    # Step 2

    returnVal = 0

    while open:  # until true(not empty), loop runs

        thisState = open.pop(0)      # removes element at 0th index from list

        # print("Printing thisState")

        thisState.kp\_102016114()

        # Step 4        # checks if current state goal state

        if thisState.isGoalReached():      # checks if goal state found

            returnVal = constructPath(thisState)

            break

        # Step 5    Generate all possible States

        nextStates = thisState.possibleNextStates()

        # Step 6

        for eachState in nextStates:

            if eachState not in open and eachState not in closed:

                # If next state is better than current state (higher heuristic value is better)

                if eachState.heuristic() > thisState.heuristic():

                    open.append(eachState)

                    closed.append(thisState)

    if returnVal != 1:

        print("ERROR !! LOCAL MAXIMA")

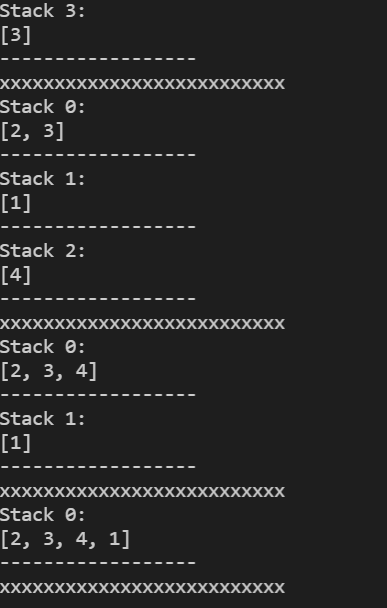
start = [[2, 3, 4, 1], [], [], []]     # start state

goal = [1, 2, 3, 4]                    # goal state

problem = kishan\_pandey(start, goal)

HillClimbing(problem)

**OUTPUT:**



1. Solve the following blocks world problem using Steepest Hill Climbing Algorithm.

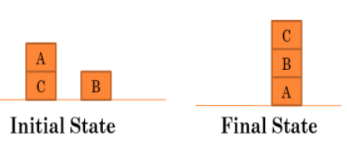
Take the following heuristic function:

+n for block resting on the current support structure and n is equal to number of

blocks below it.

-n for block resting on the incurrent support structure and n is equal to number of

blocks below it



**CODE:**

import copy

class kishan\_pandey:

    def \_\_init\_\_(self, start, goal):   # constructor gets automatically called when object calls class

        self.currentState = start      # start state initialise the parameters

        self.goalState = goal          # goal state

        self.prevState = None          # call returns

    def isGoalReached(self):           # checks if goal reached or not

        for i in range(0, 3):

            if self.currentState[i] == goal:

                return True

        return False

    def displayState(self):           # checks if current state is equal to goal state

        for i in range(0, 3):

            if self.currentState[i] != []:  # checks if start state is not empty

                print(f"Stack {i}:")

                print(self.currentState[i])

                print("------------------")

        print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    def \_\_gt\_\_(self, other):         # objects of class self,others

        return self.heuristic() > other.heuristic()

    def \_\_lt\_\_(self, other):

        return self.heuristic() < other.heuristic()

    def \_eq\_(self, other):

        return self.currentState == other.currentState

    def move\_StackX\_StackY(self, x, y):     # movement of blocks

        if self.currentState[x] != [] and len(self.currentState[y]) != 3:

            self.prevState = copy.deepcopy(self)  # changes made to object are not reflected at original object

            block = self.currentState[x].pop()

            self.currentState[y].append(block)

            return True

        else:

            return False

    def possibleNextStates(self):    # generates possible next states

        stateList = []

        for i in range(0, 3):

            for j in range(0, 3):

                copy\_state = copy.deepcopy(self)

                if i != j and copy\_state.move\_StackX\_StackY(i, j):

                    # copy\_state.displayState()

                    stateList.append(copy\_state)

        return stateList

    def heuristic(self):    # calculate heuristic value

        value = 0

        for i in range(0, 3):

            goalBlock = self.goalState[i]   # intialise elements of goal state

            goalBlockIndex = i              # position of element in gial state at particular index

            for j in range(0, 3):

                flag = 0

                if self.currentState[j] != []:  # if current state not empty

                    for k in range(0, len(self.currentState[j])):

                        if self.currentState[j][k] == goalBlock:    # if goal state found

                            currentBlockIndexX = j                  # store index of the position

                            currentBlockIndexY = k

                            flag = 1

                            break

                if flag == 1:

                    flag = 0

                    break

            if currentBlockIndexY != goalBlockIndex:    # calculate no. of blocks below it

                value -= currentBlockIndexY

            else:

                value += currentBlockIndexY            # else gives +index

        return value

def kp\_102016114(goalState):    # shows path from start--->goal

    print("Displaying path from start to goal")

    while goalState:

        goalState.displayState()

        goalState = goalState.prevState

    return 1

def SteepestHillClimbing(startState):   # call reaches here\*\*\*\*>

    open = []     # intialise empty list

    closed = []

    # Step 1    insert all elements of start into open

    open.append(startState)

    # Step 2   # untill true(not empty), loop runs

    returnVal = 0

    while open:

        thisState = open.pop(0)  # removes element at 0 index

        thisState.displayState()

        # Step 4

        if thisState.isGoalReached():    # checks if current state = goal state

            print("Goal state FOUND.. STOP SEARCHING ")

            returnVal = kp\_102016114(thisState)

            break

        # Step 5    Generate next possible states

        nextStates = thisState.possibleNextStates()

        # Step 6

        for eachState in nextStates:

            if eachState not in open and eachState not in closed:

                # If next state is better than current state(higher heuristic value is better)

                if eachState.heuristic() > thisState.heuristic():

                    open.append(eachState)

                    closed.append(thisState)

        # Step 7

        # Sort in descending order

        open.sort(reverse=True)

    if returnVal != 1:

        print("ERROR !!  LOCAL MAXIMA")

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# start of code    where :  1 --> A , 2--> B , 3--> C

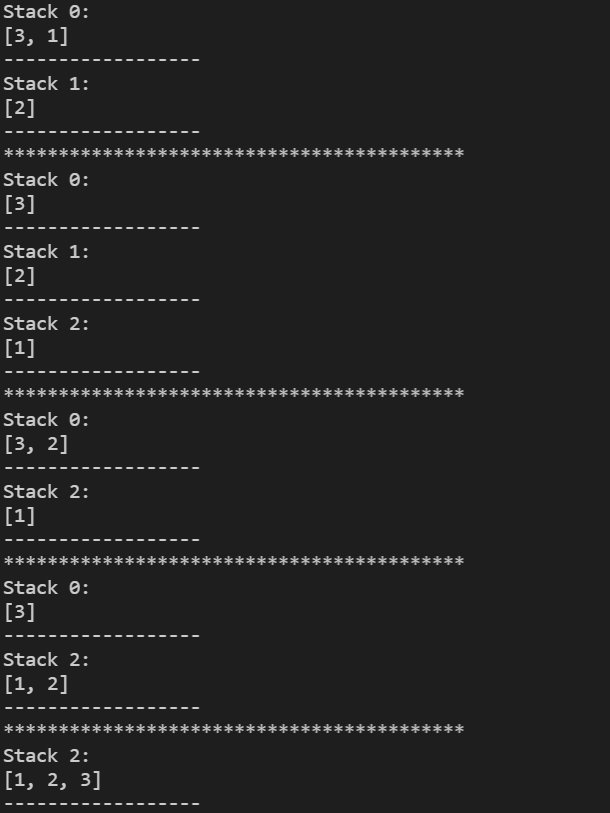
start = [[3, 1], [2], []]              # start state

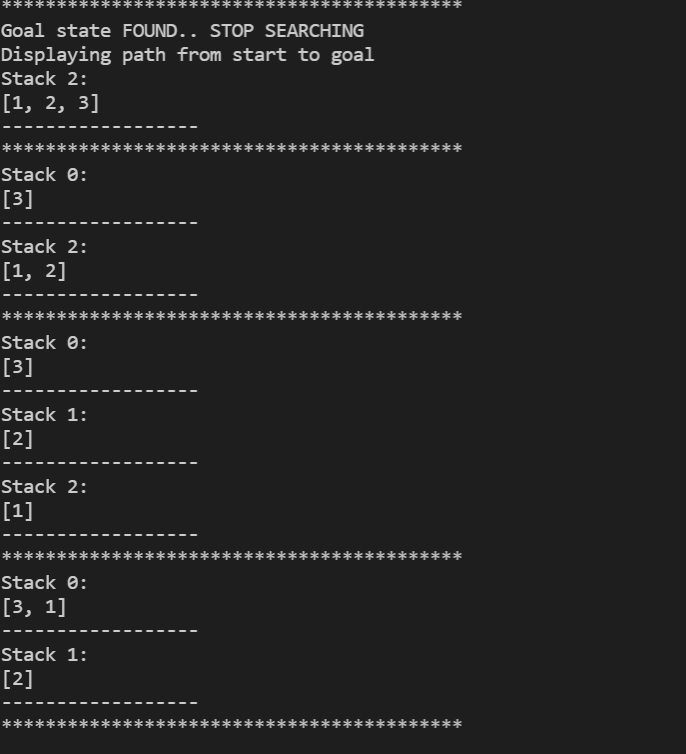
goal = [1, 2, 3]                       # goal state

problem = kishan\_pandey(start, goal)   #  object of class kishan\_pandey

SteepestHillClimbing(problem)          # passing object "problem" as parameter , call goes to function HillClimbing

**OUTPUT:**

****

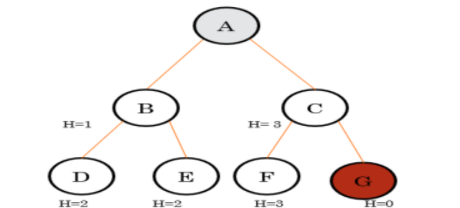


1. Solve the following problem using beam search algorithm, by taking

(i) Beam width =2

(ii) Beam width =3

Heuristic values are given in the diagram; A is the starting node and G is the goal node.



**CODE:**

import copy

class kishan\_pandey:

     def \_\_init\_\_(self, map):

          kishan\_pandey.map=map

          self.currentNode=0

          self.goalNode=6

          self.heuristic=0

          self.visitedList=[]

          self.visitedList.append(self.currentNode)

          self.prevState=None

     def kp\_102016114(self): # The current state is displayed

          print("--------------------------------")

          print("Current node:{self.currentNode} Visited nodes={self.visitedList}")

     def \_\_gt\_\_(self, other):   #retuns true if self.heuristic>other.heuristic

          return self.heuristic>other.heuristic

     def \_\_lt\_\_(self, other): #returns true if self.heuristic<other.heuristic

          return self.heuristic<other.heuristic

     def \_\_eq\_\_(self, other): #returns true if self.visitedList==other.visitedList

          return self.visitedList==other.visitedList

     def isGoalReached(self): #checks if the goal state is reached or not

          if self.goalNode in self.visitedList:

               return True

          else:

               return False

     def move(self, node):

          if node!=self.currentNode and node not in self.visitedList and kishan\_pandey.map[self.currentNode][node]!=-1:

               print(f"Moving from node {self.currentNode} to {node}")

               self.heuristic=kishan\_pandey.map[self.currentNode][node]

               self.currentNode=node

               self.visitedList.append(self.currentNode)

               return True

          else:

               return False

     def possibleNextStates(self): # Used to define the next possible state

          stateList=[]

          for i in range(0, len(kishan\_pandey.map[0])):

               state=copy.deepcopy(self)

               if state.move(i):

                    self.prevState=copy.deepcopy(self) # deepcopy is used so that any changes made in the copy object is not reflected in main object

                    stateList.append(state)

          return stateList

def constructPath(goalState): # constucts path from start node to end node

    print("The solution path from Goal to Start")

    while goalState is not None:

        goalState.kp\_102016114()

        goalState=goalState.prevState

def BeamSearch(startState, beta):

     open=[]

     closed=[]

     flag=0

     B=beta

     open.append(startState)

     # performing the queue operation wherein we make use of priority queue in open and add states in closed respectively

     while(open):

          thisState=open.pop(0)

          thisState.kp\_102016114()

          if thisState not in closed: # if current state not in closed then we add it to closed state

               closed.append(thisState)

               if thisState.isGoalReached(): # checks for each state if goal node is present or not

                    print("Goal state found.. stopping search")

                    constructPath(thisState)

                    flag=1

                    break

               nextStates=thisState.possibleNextStates() #for every state we define the possible next states

               for eachState in nextStates:

                    open.append(eachState)

               open.sort()

               if len(open)>B:

                    while len(open)!=B:

                         open.pop()

     if flag!=1:

          print("Error, can't reach goal node")

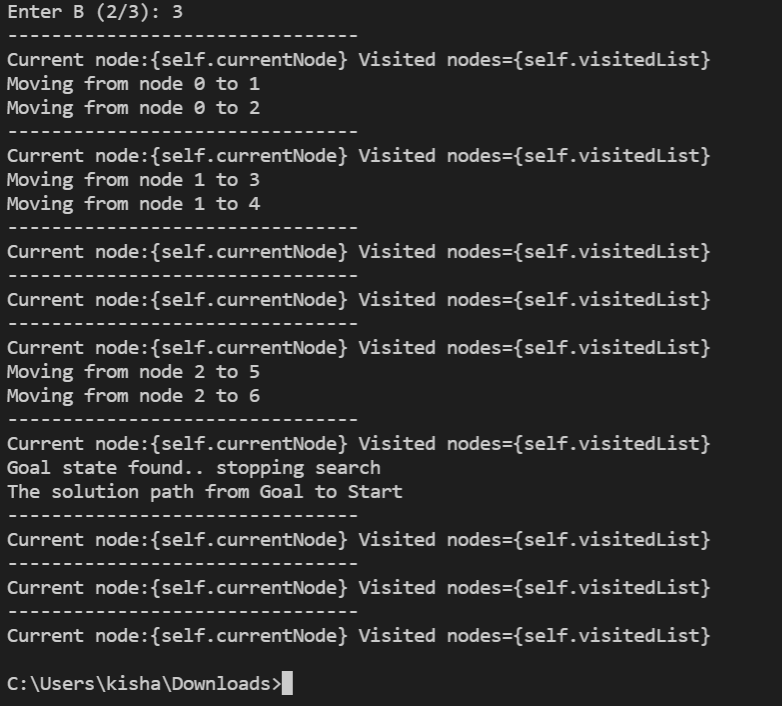
map=[[-1,  1,  3, -1, -1, -1, -1], [-1, -1, -1,  2,  2, -1, -1],[-1, -1, -1, -1, -1,  3,  0], [-1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1]]

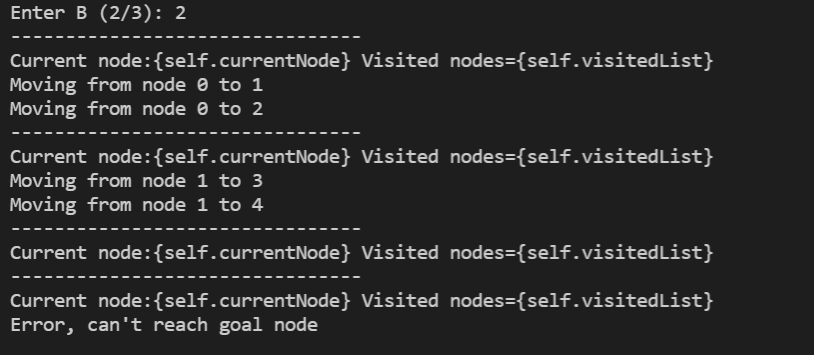
beta=int(input("Enter B (2/3): "))

problem=kishan\_pandey(map)

BeamSearch(problem, beta)

**OUTPUT:**

****

****

**END**