Lab 10: Implementation of block world Problem

**Aim:** - To implement block world problem using python.

**Source Code for the block world problem implementation using python: -**

class PREDICATE:

  def \_\_str\_\_(self):

    pass

  def \_\_repr\_\_(self):

    pass

  def \_\_eq\_\_(self, other) :

    pass

  def \_\_hash\_\_(self):

    pass

  def get\_action(self, world\_state):

    pass

#OPERATIONS - Stack, Unstack, Pickup, Putdown

class Operation:

  def \_\_str\_\_(self):

    pass

  def \_\_repr\_\_(self):

    pass

  def \_\_eq\_\_(self, other) :

    pass

  def precondition(self):

    pass

  def delete(self):

    pass

  def add(self):

    pass

class ON(PREDICATE):

  def \_\_init\_\_(self, X, Y):

    self.X = X

    self.Y = Y

  def \_\_str\_\_(self):

    return "ON({X},{Y})".format(X=self.X,Y=self.Y)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def \_\_hash\_\_(self):

      return hash(str(self))

  def get\_action(self, world\_state):

    return StackOp(self.X,self.Y)

class ONTABLE(PREDICATE):

  def \_\_init\_\_(self, X):

    self.X = X

  def \_\_str\_\_(self):

    return "ONTABLE({X})".format(X=self.X)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def \_\_hash\_\_(self):

      return hash(str(self))

  def get\_action(self, world\_state):

    return PutdownOp(self.X)

class CLEAR(PREDICATE):

  def \_\_init\_\_(self, X):

    self.X = X

  def \_\_str\_\_(self):

    return "CLEAR({X})".format(X=self.X)

    self.X = X

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def \_\_hash\_\_(self):

    return hash(str(self))

  def get\_action(self, world\_state):

    for predicate in world\_state:

      #If Block is on another block, unstack

      if isinstance(predicate,ON) and predicate.Y==self.X:

        return UnstackOp(predicate.X, predicate.Y)

    return None

class HOLDING(PREDICATE):

  def \_\_init\_\_(self, X):

    self.X = X

  def \_\_str\_\_(self):

    return "HOLDING({X})".format(X=self.X)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def \_\_hash\_\_(self):

    return hash(str(self))

  def get\_action(self, world\_state):

    X = self.X

    #If block is on table, pick up

    if ONTABLE(X) in world\_state:

      return PickupOp(X)

    #If block is on another block, unstack

    else:

      for predicate in world\_state:

        if isinstance(predicate,ON) and predicate.X==X:

          return UnstackOp(X,predicate.Y)

class ARMEMPTY(PREDICATE):

  def \_\_init\_\_(self):

    pass

  def \_\_str\_\_(self):

    return "ARMEMPTY"

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def \_\_hash\_\_(self):

    return hash(str(self))

  def get\_action(self, world\_state=[]):

    for predicate in world\_state:

      if isinstance(predicate,HOLDING):

        return PutdownOp(predicate.X)

    return None

class StackOp(Operation):

  def \_\_init\_\_(self, X, Y):

    self.X = X

    self.Y = Y

  def \_\_str\_\_(self):

    return "STACK({X},{Y})".format(X=self.X,Y=self.Y)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def precondition(self):

    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def delete(self):

    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def add(self):

    return [ ARMEMPTY() , ON(self.X,self.Y) ]

class UnstackOp(Operation):

  def \_\_init\_\_(self, X, Y):

    self.X = X

    self.Y = Y

  def \_\_str\_\_(self):

    return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def precondition(self):

    return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]

  def delete(self):

    return [ ARMEMPTY() , ON(self.X,self.Y) ]

  def add(self):

    return [ CLEAR(self.Y) , HOLDING(self.X) ]

class PickupOp(Operation):

  def \_\_init\_\_(self, X):

    self.X = X

  def \_\_str\_\_(self):

    return "PICKUP({X})".format(X=self.X)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def precondition(self):

    return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]

  def delete(self):

    return [ ARMEMPTY() , ONTABLE(self.X) ]

  def add(self):

    return [ HOLDING(self.X) ]

class PutdownOp(Operation):

  def \_\_init\_\_(self, X):

    self.X = X

  def \_\_str\_\_(self):

    return "PUTDOWN({X})".format(X=self.X)

  def \_\_repr\_\_(self):

    return self.\_\_str\_\_()

  def \_\_eq\_\_(self, other) :

    return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

  def precondition(self):

    return [ HOLDING(self.X) ]

  def delete(self):

    return [ HOLDING(self.X) ]

  def add(self):

    return [ ARMEMPTY() , ONTABLE(self.X) ]

def isPredicate(obj):

  predicates = [ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY]

  for predicate in predicates:

    if isinstance(obj,predicate):

      return True

  return False

def isOperation(obj):

  operations = [StackOp, UnstackOp, PickupOp, PutdownOp]

  for operation in operations:

    if isinstance(obj,operation):

      return True

  return False

def arm\_status(world\_state):

  for predicate in world\_state:

    if isinstance(predicate, HOLDING):

      return predicate

  return ARMEMPTY()

class GoalStackPlanner:

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

    self.initial\_state = initial\_state

    self.goal\_state = goal\_state

  def get\_steps(self):

    #Store Steps

    steps = []

    #Program Stack

    stack = []

    #World State/Knowledge Base

    world\_state = self.initial\_state.copy()

    #Initially push the goal\_state as compound goal onto the stack

    stack.append(self.goal\_state.copy())

    #Repeat until the stack is empty

    while len(stack)!=0:

      #Get the top of the stack

      stack\_top = stack[-1]

      #If Stack Top is Compound Goal, push its unsatisfied goals onto stack

      if type(stack\_top) is list:

        compound\_goal = stack.pop()

        for goal in compound\_goal:

          if goal not in world\_state:

            stack.append(goal)

      #If Stack Top is an action

      elif isOperation(stack\_top):

        #Peek the operation

        operation = stack[-1]

        all\_preconditions\_satisfied = True

        #Check if any precondition is unsatisfied and push it onto program stack

        for predicate in operation.delete():

          if predicate not in world\_state:

            all\_preconditions\_satisfied = False

            stack.append(predicate)

        #If all preconditions are satisfied, pop operation from stack and execute it

        if all\_preconditions\_satisfied:

          stack.pop()

          steps.append(operation)

          for predicate in operation.delete():

            world\_state.remove(predicate)

          for predicate in operation.add():

            world\_state.append(predicate)

      #If Stack Top is a single satisfied goal

      elif stack\_top in world\_state:

        stack.pop()

      #If Stack Top is a single unsatisfied goal

      else:

        unsatisfied\_goal = stack.pop()

        #Replace Unsatisfied Goal with an action that can complete it

        action = unsatisfied\_goal.get\_action(world\_state)

        stack.append(action)

        #Push Precondition on the stack

        for predicate in action.precondition():

          if predicate not in world\_state:

            stack.append(predicate)

    return steps

if \_\_name\_\_ == '\_\_main\_\_':

  initial\_state = [

    ON('B','A'),

    ON('C','B'),

    ONTABLE('A'),ONTABLE('D'),

    CLEAR('C'),CLEAR('D'),

    ARMEMPTY()

  ]

  goal\_state = [

    ON('B','D'),ON('C','A'),

    ONTABLE('D'),ONTABLE('A'),

    CLEAR('B'),CLEAR('C'),

    ARMEMPTY()

  ]

  goal\_stack = GoalStackPlanner(initial\_state=initial\_state, goal\_state=goal\_state)

  steps = goal\_stack.get\_steps()

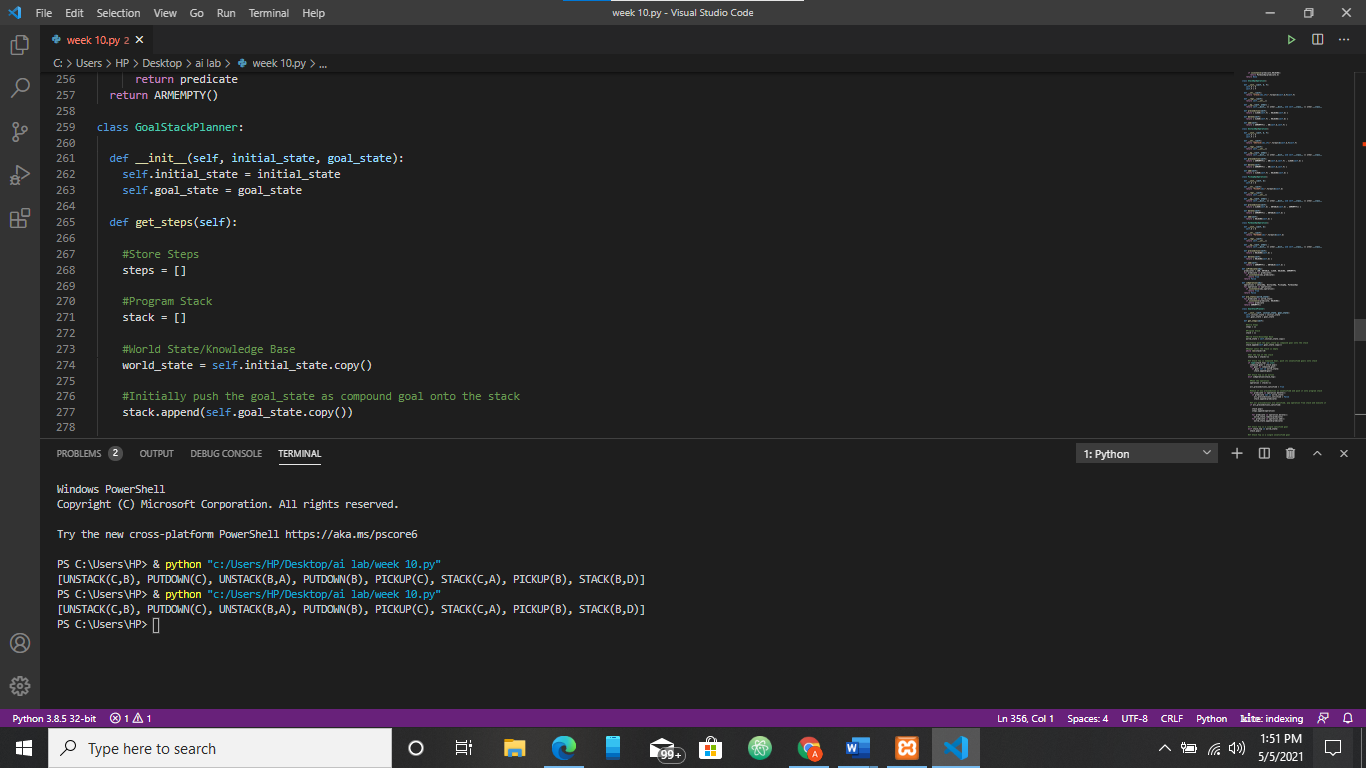
  print(steps)

**Output:-**

**[UNSTACK(C,B), PUTDOWN(C), UNSTACK(B,A), PUTDOWN(B), PICKUP(C), STACK(C,A), PICKUP(B), STACK(B,D)]**

**Screenshots:-**

Output:-



**Result: -** Successfully implemented and executed block world problem using python.