EX 10 Implementation of Block World Problem Algorithm

Akshata Bihany

RA1811003010740

2/05/2021

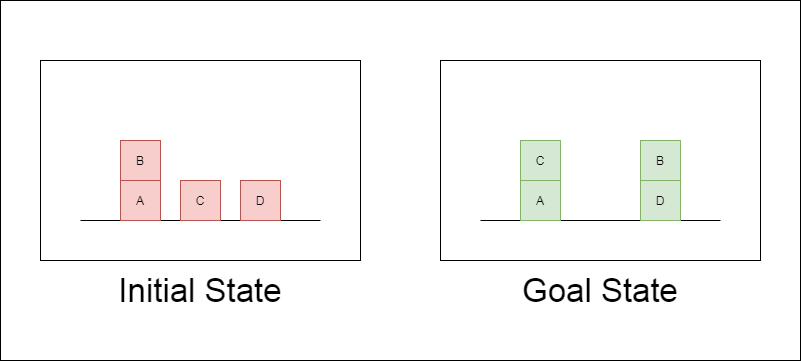
**Problem Description :-**

There is a table on which some blocks are placed. Some blocks may or may not be stacked on other blocks. We have a robot arm to pick up or put down the blocks. The robot arm can move only one block at a time, and no other block should be stacked on top of the block which is to be moved by the robot arm.Our aim is to change the configuration of the blocks from the Initial State to the Goal State, both of which have been specified in the diagram above.

* Given below are the list of predicates as well as their intended meaning

1. ON(A,B) : Block A is on B
2. ONTABLE(A) : A is on table
3. CLEAR(A) : Nothing is on top of A
4. HOLDING(A) : Arm is holding A.
5. ARMEMPTY : Arm is holding nothing
   * Operations performed by the robot arm The Robot Arm can perform 4 operations:
6. STACK(X,Y) : Stacking Block X on Block Y
7. UNSTACK(X,Y) : Picking up Block X which is on top of Block Y
8. PICKUP(X) : Picking up Block X which is on top of the table
9. PUTDOWN(X) : Put Block X on the table

**DIAGRAM FOR UNDERSTANDING**



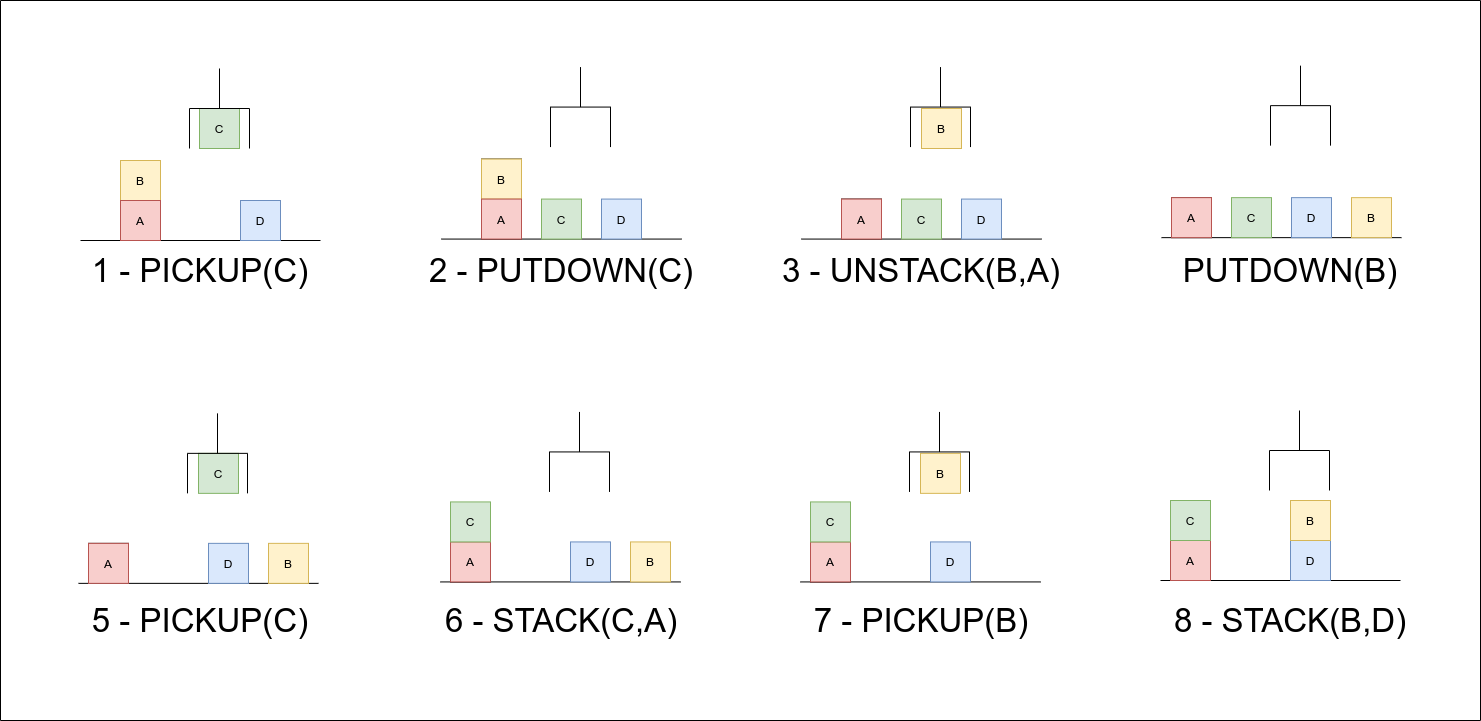
**Initial State** — ON(B,A) ∧ ONTABLE(A) ∧ ONTABLE(C) ∧ ONTABLE(D) ∧ CLEAR(B) ∧

CLEAR(C) ∧ CLEAR(D) ∧ ARMEMPTY

**Goal State** — ON(C,A) ∧ ON(B,D) ∧ ONTABLE(A) ∧ ONTABLE(D) ∧ CLEAR(B) ∧

CLEAR(C) ∧ ARMEMPTY

The visual representation of our steps variable looks like this.



**Code :-**

#Base Classes

#PREDICATE - ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY

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'), ONTABLE('A'),ONTABLE('C'),ONTABLE('D'),

CLEAR('B'),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)

# Screenshot from Output (Jupyter on VSCode) :-



**Result**

The experiment was successfully implemented and executed.