ARTIFICIAL INTELLIGENCE

EXPERIMENT 10

IMPLEMENTATION OF BLOCK WORLD PROBLEM

Nikith Kumar Seemakurthi
RA1911003020480

AIM:

To implement a block world problem.

ALGORITHM:

1. The block world is a NP-hard problem and we wanted to find a smart solution to solve it.
2. We used a number of algorithms to solve the problem. These include DFS, BFS, UCS, A* and simulated annealing. We used six different heuristics to solve the problem using A*.
3. Predicates can be thought of as a statement which helps us convey the information about a configuration in Blocks World.
4. Using the following predicates we present the initial state and goal state:
    - ON(A,B) : Block A is on B
    - ONTABLE(A) : A is on table
    - CLEAR(A) : Nothing is on top of A
    - HOLDING(A) : Arm is holding A.
    - ARMEMPTY : Arm is holding nothing.
5. Print the output.

PROGRAM:

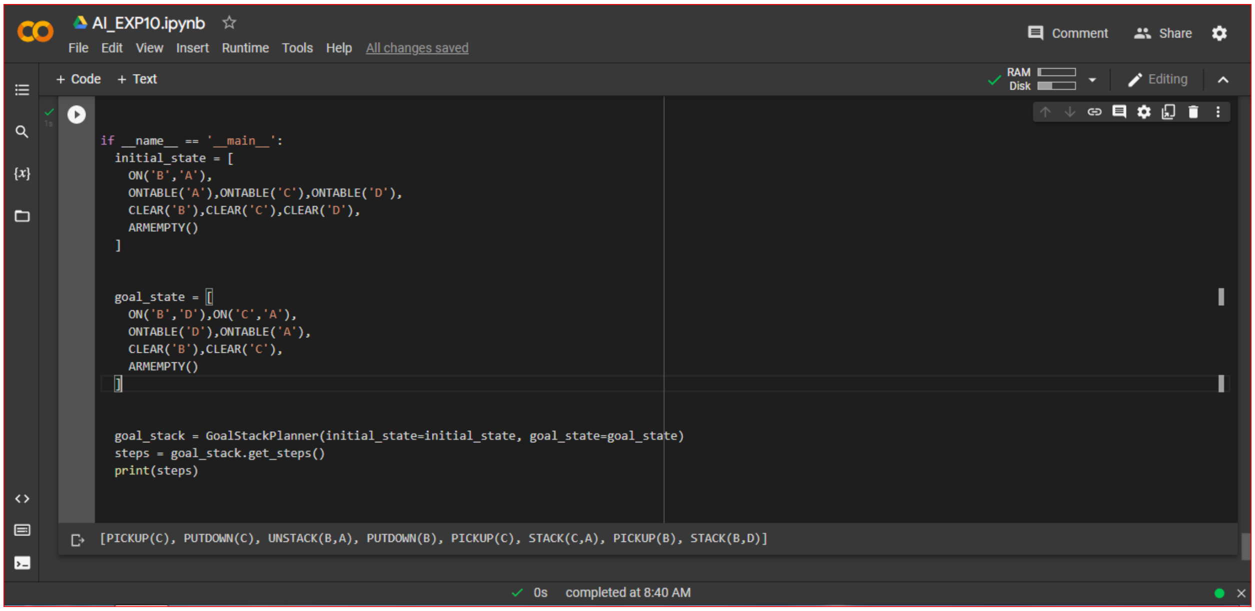
```
class PREDICATE:
  def __str__(self):
    pass
  def __repr__(self):
    pass
  def __eq__(self, other) :
    pass
  def __hash__(self):
    pass
  def get_action(self, world_state):
    pass
```

```
#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)
```

OUTPUT:



```
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)
```

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

RESULT:

Thus, the block world problem is implemented.