**[I highly recommend viewing the code on our github instead](https://github.com/caseyeaster39/5210-Projects/tree/main/project-2)**

**main.py**

from agent import Agent  
  
  
def main():  
 # Question 6, running for n=100 orders  
 n = 100  
 consecutive\_agent = Agent() # Initialize agent  
 for i in range(n): # 100 orders  
 consecutive\_agent.protocol() # Run protocol  
  
 # Question 5  
 edge\_case\_agent = Agent() # Initialize agent  
 edge\_case\_agent.protocol(rand=False, shelves=[33], # Passing in our own parameters for edge case  
 div=6, single\_run=True)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**agent.py**

from project\_utils import path\_trace, path\_merge, path\_forger  
from warehouse import Warehouse, Division, Order  
  
  
class Agent:  
 def \_\_init\_\_(self): # Agent initialization  
 # Warehouse Components  
 self.order = None  
 self.location = {  
 'warehouse': 1,  
 'div': 1  
 }  
 self.wh = Warehouse()  
 self.div = Division()  
  
 # Scoring and Tracking  
 self.score = 0  
 self.cumulative\_score = 0  
 self.num\_runs = 0  
 self.step\_count = 0  
 self.cumulative\_steps = 0  
 self.max\_path = float('-inf')  
 self.min\_path = float('inf')  
  
 def protocol(self, rand=True, shelves=None, div=None, single\_run=False): # Main protocol  
 self.order = Order(rand, shelves, div) # Generate the order for this run  
 self.wh\_search(self.order.div, single\_run) # Search the warehouse for the division  
 self.div\_search(self.order.shelves, single\_run) # Enter division and search for order  
 if single\_run:  
 print("Returning to Warehouse") # Printouts for single run  
 print(f'Final Score: \t{self.score}')  
 print(f'Final Step Count: {self.score}')  
 self.score = 0  
 self.step\_count = 0  
 else: # Printouts for consecutive runs  
 self.scoring\_func()  
  
 def wh\_search(self, target\_node, single\_run): # Finding the division in warehouse:  
 target\_path = path\_merge(self.location['warehouse'], target\_node) # Merge path agent to target  
 self.move(target\_path, loc='warehouse', single\_run=single\_run) # Move on the merged path  
  
 def div\_search(self, shelves, single\_run): # Finding shelves in a division:  
 agent\_path = path\_trace(self.location['div']) # Trace agent back to root  
  
 targets = self.id\_search(shelves) # Iterative Deepening Search -> Nodes  
 target\_paths = path\_forger(targets, agent\_path) # Create priority queue  
  
 while len(target\_paths) > 0: # Until targets have all been retrieved  
 this\_path = target\_paths[0] # Get shortest path  
 self.move(this\_path, loc='div', single\_run=single\_run) # and move there  
 agent\_path = path\_trace(self.location['div']) # Get agent path to assess next target  
 del targets[f'{this\_path[-1]}'] # Delete this target from targets  
 target\_paths = path\_forger(targets, agent\_path) # Recalculate paths to targets  
 self.go\_home(single\_run) # Return to division root  
  
 def move(self, path, loc, single\_run): # Move along a path  
 current\_tree = self.wh.node\_list if loc == 'warehouse' \  
 else self.div.node\_list # Define current tree  
 for step in path:  
 current\_node = current\_tree[self.location[loc] - 1] # Get current node  
 if step == self.location[loc]:  
 continue # Don't step if already there  
 self.step\_count += 1 # Step counter  
 self.location[loc] = step # Move agent  
 if loc == 'div':  
 self.score += 1 # Division weights = 1  
 else:  
 next\_node = self.wh.node\_list[step - 1] # Get next node  
 self.score\_calc(current\_node, next\_node) # Score weight for moving along edge  
 if single\_run: # Printouts for individual runs  
 print(f'Moved to {self.location[loc]} in {loc}, current score: {self.score}')  
  
 def score\_calc(self, node1, node2):  
 parent\_node = node1 if node2.parent == node1.num else node2 # Figure out which node is parent  
 child\_node = node1 if parent\_node != node1 else node2 # Other node is child  
  
 edge = parent\_node.left if parent\_node.left[0] == child\_node.num \  
 else parent\_node.right # Retrieve edge weight  
  
 self.score += edge[1] # Add edge weight to score  
  
 def id\_search(self, targets): # Iterative Deepening Search  
 results = {}  
 for target in targets: # Loop through current targets  
 results[f'{target}'] = 'None' # No initial result  
 depth = 0  
 while depth <= self.div.max\_depth: # Incrementing depth limit for DLS  
 result = self.dl\_search(target, depth) # Try DLS at current depth limit  
 if result:  
 results[f'{target}'] = result # If target is found, add to results  
 break # and break out of DLS loop  
 depth += 1 # Otherwise, increment depth  
 return results  
  
 def dl\_search(self, target, depth\_limit):  
 frontier = list() # Empty frontier  
 frontier.append(self.div.node\_list[0]) # Start at root  
 while len(frontier) > 0: # Until frontier is exhausted  
 node = frontier.pop() # Pop top node  
 if node.num == target:  
 return node # Return node if target found  
 if not node.depth > depth\_limit: # If within depth limit  
 self.expand(frontier, node) # expand frontier from node  
 return None # If you make it this far, search failed  
  
 def expand(self, frontier, node):  
 right\_index = node.right[0] - 1 # Get frontier node indices  
 left\_index = node.left[0] - 1  
  
 frontier.append(self.div.node\_list[right\_index]) # Add nodes to frontier  
 frontier.append(self.div.node\_list[left\_index])  
  
 def go\_home(self, single\_run):  
 self.move( # Move back to division root  
 path\_trace(self.location['div'], reverse=False),  
 loc='div', single\_run=single\_run  
 )  
  
 def scoring\_func(self):  
 # Score calculations  
 self.num\_runs += 1  
 self.cumulative\_score += self.score  
 average\_score = (self.cumulative\_score + self.score) / self.num\_runs  
  
 # Step calculations  
 self.cumulative\_steps += self.step\_count  
 average\_steps = (self.cumulative\_steps + self.step\_count) / self.num\_runs  
 self.max\_path = self.step\_count if self.max\_path < self.step\_count else self.max\_path  
 self.min\_path = self.step\_count if self.min\_path > self.step\_count else self.min\_path  
  
 # Report printout  
 print(f'Run number {self.num\_runs}:')  
 print('##########################################')  
 print(f'Score this run: \t{self.score}')  
 print(f'Average Score: \t\t{round(average\_score, 2)}\n')  
  
 print(f'Avg Step Count: \t{round(average\_steps)}')  
 print(f'Max Step Count: \t{self.max\_path}')  
 print(f'Min Step Count: \t{self.min\_path}')  
 print('##########################################')  
  
 # Reset non-cumulative metrics  
 self.score = 0  
 self.step\_count = 0

**warehouse.py**

import math  
import random  
  
from project\_utils import path\_trace  
  
  
# WAREHOUSE MAIN FLOOR  
class Node:  
 def \_\_init\_\_(self, num, is\_parent, left=None, right=None):  
 # Warehouse info  
 self.num = num # Reference number  
 self.depth = math.floor(math.log2(num)) # Depth  
 self.root\_path = path\_trace(num) # Path from node to root  
  
 # Children edges (if any) in tuple form (node, weight)  
 self.is\_parent = is\_parent # Has children  
 self.left = left  
 self.right = right  
  
 # Parent node (except for root)  
 self.parent = num // 2 if num > 1 else None # Parent node  
  
  
def floor\_gen():  
 node\_list = list()  
  
 # Node list hard-coded in due to nature of tree.  
 node\_list.append(Node(1, is\_parent=True, left=(2, 20), right=(3, 20)))  
 node\_list.append(Node(2, is\_parent=True, left=(4, 20), right=(5, 30)))  
 node\_list.append(Node(3, is\_parent=True, left=(6, 40), right=(7, 10)))  
 node\_list.append(Node(4, is\_parent=True, left=(8, 10), right=(9, 20)))  
 node\_list.append(Node(5, is\_parent=True, left=(10, 30), right=(11, 20)))  
 node\_list.append(Node(6, is\_parent=True, left=(12, 30), right=(13, 20)))  
 node\_list.append(Node(7, is\_parent=True, left=(14, 20), right=(15, 20)))  
 node\_list.append(Node(8, is\_parent=False))  
 node\_list.append(Node(9, is\_parent=False))  
 node\_list.append(Node(10, is\_parent=False))  
 node\_list.append(Node(11, is\_parent=False))  
 node\_list.append(Node(12, is\_parent=False))  
 node\_list.append(Node(13, is\_parent=False))  
 node\_list.append(Node(14, is\_parent=False))  
 node\_list.append(Node(15, is\_parent=False))  
  
 return node\_list  
  
  
class Warehouse:  
 def \_\_init\_\_(self):  
 self.node\_list = floor\_gen() # Generate Warehouse tree  
  
  
# WAREHOUSE DIVISION  
def generate\_tree(max\_node=63):  
 node\_list = list() # Division generator  
 i = 1  
 while i <= max\_node:  
 if i <= max\_node // 2: # Nodes with children  
 left\_child = (i \* 2, 1) # Generate left child  
 right\_child = (i \* 2 + 1, 1) # Generate right child  
 node\_list.append(Node(i, True, left\_child, right\_child)) # Create Node based on info and add to list  
 else:  
 node\_list.append(Node(i, False)) # Create a leaf Node and add to list  
 i += 1  
 max\_depth = math.floor(math.log2(max\_node)) # Leaf depth  
 return node\_list, max\_depth  
  
  
class Division:  
 def \_\_init\_\_(self):  
 self.node\_list, self.max\_depth = generate\_tree() # Generate Division tree  
  
  
# ORDER GENERATION  
class Order:  
 def \_\_init\_\_(self, rand=True, shelves=None, div=None):  
 if rand: # Random generator if rand  
 self.shelves = random.sample(range(1, 63), random.randint(1, 3))  
 self.div = random.randint(1, 15)  
 else: # Pass in parameters if not rand  
 self.shelves = shelves  
 self.div = div

**project\_utils.py**

import math  
  
  
# PATH UTILITIES  
def path\_trace(node, reverse=True): # Root traceback  
 level = math.floor(math.log2(node)) # Current depth  
 root\_path = [node] # Start from this node  
 next\_node = node  
 for i in range(level):  
 next\_node = next\_node // 2 # Step back to parent,  
 root\_path.append(next\_node) # add it to the path list  
 if reverse:  
 root\_path.reverse() # reverse the order so root is first  
 return root\_path  
  
  
def path\_merge(var1, var2, method='node'): # Path merging utility  
 path1, path2, common = find\_common(var1, var2, method=method) # Find common node  
 path1.reverse() # Because the agent is backtracking  
  
 path1\_merge\_point = path1.index(common) # Find index of common node  
 path2\_merge\_point = path2.index(common) # Find index of common node  
  
 new\_path = path1[0:path1\_merge\_point] + path2[path2\_merge\_point:] # Merge the two at the common node  
  
 return new\_path # Return merged path  
  
  
def find\_common(var1, var2, method='node'): # Utility to find common node  
 if method == 'node': # If passed two nodes,  
 path1 = path\_trace(var1) # run path trace on each  
 path2 = path\_trace(var2) # to get their root paths.  
 else:  
 path1 = var1 # Otherwise, use the paths passed in  
 path2 = var2  
 rng = min(len(path1), len(path2)) # Use shorter path length  
  
 common\_node = 1 # Root  
 for i in range(rng):  
 if path1[i] == path2[i]: # Change common if they are the same  
 common\_node = path1[i]  
 else: # Otherwise, the paths have diverged  
 break  
  
 return path1, path2, common\_node # Return the paths and their common node  
  
  
def path\_forger(targets, agent\_path): # Creates priority queue based on  
 target\_paths = [] # step length  
 for target in targets:  
 target\_paths.append(  
 path\_merge( # Create paths from  
 agent\_path, # current agent location  
 targets[f'{target}'].root\_path, # and targets  
 method='path')  
 )  
 target\_paths.sort(key=len) # Sort ascending by length and return  
 return target\_paths