11/25, 12:19] Krishnendu Samanta (SNU): def dfs(graph, start, visited=None):

```
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'F'],
    'B': ['D', 'E'],
    'F': ['C'],
    'C': [],
    'D': [],
    'E': []
}

visited_nodes = dfs(graph, 'A')
print("Visited nodes:", visited_nodes)
```

[11/25, 12:19] Krishnendu Samanta (SNU): class AONode:

```
    def __init__(self, name, cost=0):
        self.name = name
        self.cost = cost
        self.children = []
        self.solution = False
```

```
    def add_child(self, children):
        self.children.append(children)

def ao_star(node):
    if node.solution:
        return node.cost

    if not node.children:
        return node.cost

    for child_set in node.children:
        cost = 0
        for child in child_set:
            cost += ao_star(child)
        if not node.solution or cost < node.cost:
            node.cost = cost
            node.solution = True
    return node.cost

# Example of using AO* algorithm on an And-Or tree
A = AONode("A")
B = AONode("B", 5)
C = AONode("C", 2)
D = AONode("D", 3)
E = AONode("E", 6)

A.add_child([B, C])
```

A.add_child([D])

C.add_child([E])

print("Minimum cost to solve:", ao_star(A))

[11/25, 12:19] Krishnendu Samanta (SNU): def tower_of_hanoi(n, source, auxiliary, target):

```
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)


# Number of disks
n = 3
tower_of_hanoi(n, 'A', 'B', 'C')
```

[11/25, 12:19] Krishnendu Samanta (SNU): Eyita A* er ta

[11/25, 12:19] Krishnendu Samanta (SNU): def water_jug(jug1, jug2, target):

```
    visited = set()
    queue = [(0, 0)]
    while queue:
        a, b = queue.pop(0)
        if (a, b) in visited:
            continue
        print(f"({a}, {b})")
        if a == target or b == target:
            return (a, b)
        visited.add((a, b))
```

```
        queue.extend([
            (jug1, b), (a, jug2),
            (0, b), (a, 0),
            (min(jug1, a + b), a + b - min(jug1, a + b)),
            (a + b - min(jug2, a + b), min(jug2, a + b))
        ])
    return None

# Example usage
jug1, jug2, target = 4, 3, 2
print("Solution:", water_jug(jug1, jug2, target))
```

[11/25, 12:19] Krishnendu Samanta (SNU): from queue import PriorityQueue

```
class Graph:
    def __init__(self, graph, heuristic):  # Fix the constructor method
        self.graph = graph
        self.heuristic = heuristic

    def a_star(self, start, goal):
        open_list = PriorityQueue()
        open_list.put((0, start))
        came_from = {start: None}
        cost_so_far = {start: 0}

        while not open_list.empty():
            current_cost, current_node = open_list.get()

            if current_node == goal:
```

```
                break

            for neighbor, weight in self.graph.get(current_node, []):  # Handle missing keys
                new_cost = cost_so_far[current_node] + weight
                if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + self.heuristic.get(neighbor, float('inf'))  # Default heuristic
                    open_list.put((priority, neighbor))
                    came_from[neighbor] = current_node

        # Reconstruct path
        path = []
        current = goal
        while current:
            path.append(current)
            current = came_from[current]
        path.reverse()
        return path

# Example usage
graph = {
    (0, 0): [((1, 0), 1), ((2, 0), 2)],
    (1, 0): [((3, 0), 2)],
    (2, 0): [((4, 0), 2)],
    (3, 0): [((4, 1), 1)],
    (4, 0): [((4, 1), 1)],
    (4, 1): [((4, 2), 1)],
    (4, 2): [((4, 3), 1)],
```

```
    (4, 3): [((4, 4), 1)],
    (4, 4): []
}
heuristic = {
    (0, 0): 10, (1, 0): 8, (2, 0): 7,
    (3, 0): 6, (4, 0): 5, (4, 1): 4,
    (4, 2): 3, (4, 3): 2, (4, 4): 0
}

g = Graph(graph, heuristic)
print("Path:", g.a_star((0, 0), (4, 4)))
```

[11/25, 12:19] Krishnendu Samanta (SNU): from collections import deque

```
def bfs(graph, start):
    visited = set()  # To keep track of visited nodes
    queue = deque([start])  # Initialize a queue with the starting node

    while queue:

        current = queue.popleft()

        if current not in visited:
            print(current)  # Process the current node
            visited.add(current)

            # Enqueue all unvisited neighbors
            for neighbor in graph[current]:
                if neighbor not in visited:
```

```
            queue.append(neighbor)

    return visited

graph = {
    'A': ['B', 'F'],
    'B': ['D', 'E'],
    'F': ['C'],
    'C': [],
    'D': [],
    'E': []
}


visited_nodes = bfs(graph, 'A')
print("Visited nodes are: ", visited_nodes)
```

[11/25, 12:19] Krishnendu Samanta (SNU): Eyita AO* er ta

[11/25, 12:19] Krishnendu Samanta (SNU): class AONode:

```
    def __init__(self, name, cost=0):
        self.name = name
        self.cost = cost
        self.children = []  # List of child sets (AND groups)
        self.solution = False

    def add_child(self, children):
        self.children.append(children)  # Add a group of children as an AND relationship

def ao_star(node):
```

```
    # If the node is already marked as a solution, return its cost
    if node.solution:
        return node.cost

    # If the node has no children (a leaf node), return its cost
    if not node.children:
        return node.cost

    # Evaluate each child set (AND group) and calculate the cost
    for child_set in node.children:
        total_cost = 0
        for child in child_set:
            total_cost += ao_star(child)  # Recursively compute the cost of each child

        # Update the node's cost and mark it as a solution if it's better
        if not node.solution or total_cost < node.cost:
            node.cost = total_cost
            node.solution = True  # Mark this node as a part of the optimal solution

    return node.cost

# Example of using AO* algorithm on an And-Or tree
A = AONode("A")
B = AONode("B", 5)
C = AONode("C", 2)
D = AONode("D", 3)
E = AONode("E", 6)
```

Define the structure of the And-Or tree

A.add_child([B, C]) # A is solved by B AND C

A.add_child([D])   # OR A is solved by D

C.add_child([E])   # C is solved by E

Execute the AO* algorithm

print("Minimum cost to solve:", ao_star(A))

[11/25, 12:19] Krishnendu Samanta (SNU): ` ` ` BFS in python

from collections import deque

def bfs(graph, start, goal):

 visited = set() # Keep track of visited nodes to avoid cycles

 queue = deque([(start, [])]) # Initialize the queue with the starting node and an empty path

 while queue:

  node, path = queue.popleft()

  if node == goal:

   return path + [goal] # Return the path with the goal node appended

  if node not in visited:

   visited.add(node)

   for neighbor in graph[node]:

    queue.append((neighbor, path + [node])) # Add the neighbor to the queue with the updated path

 return None # If the goal is not reached, no path exists

Example usage:

graph = {

 'A': ['B', 'C'],

 'B': ['D', 'E'],

 'C': ['F'],

 'D': ['E'],

'E': ['F'],

'F': []

}

start_node = 'A'

goal_node = 'F'

path = bfs(graph, start_node, goal_node)

print("BFS in Python")

if path:

print("Shortest path:", path)

else:

print("No path found.")```

[11/25, 12:19] Krishnendu Samanta (SNU): Hill Climbing in Python

```import random

def hill_climbing(objective_function, initial_state, max_iterations=1000):

current_state = initial_state

best_state = current_state

best_value = objective_function(current_state)

for _ in range(max_iterations):

neighbors = generate_neighbors(current_state)

best_neighbor = None

best_neighbor_value = None

for neighbor in neighbors:

neighbor_value = objective_function(neighbor)

if neighbor_value > best_value:

best_neighbor = neighbor

best_neighbor_value = neighbor_value

if best_neighbor is not None:

current_state = best_neighbor
```

```
best_value = best_neighbor_value
if best_value > best_value:
best_state = current_state
best_value = best_neighbor_value
return best_state, best_value
def generate_neighbors(state):
# Example for a simple 2D grid:
neighbors = []
for dx in [-1, 0, 1]:
for dy in [-1, 0, 1]:
if dx != 0 or dy != 0:
new_state = list(state)  # Create a copy of the state
new_state[0] += dx
new_state[1] += dy
neighbors.append(new_state)
return neighbors
# Example usage:
def objective_function(state):
return -(state[0] ** 2 + state[1] ** 2)  # Minimize the sum of squares
initial_state = [random.randint(-10, 10), random.randint(-10, 10)]
best_state, best_value = hill_climbing(objective_function, initial_state)
print("Best state:", best_state)
print("Best value:", best_value)` ` `
[11/25, 12:19] Krishnendu Samanta (SNU): ` ` `Water Jug Problem in Python
def water_jug_problem(jug1_capacity, jug2_capacity, target_quantity):
  visited = set()  # Keep track of visited states to avoid cycles
  queue = [(0, 0)]  # Initialize the queue with the initial state (both jugs empty)
  while queue:
```

```
        jug1, jug2 = queue.pop(0)
        if jug1 == target_quantity or jug2 == target_quantity:
            return [(jug1, jug2)]
        # Generate new states based on possible actions
        new_states = [
            (jug1_capacity, jug2),  # Fill jug1
            (jug1, jug2_capacity),  # Fill jug2
            (0, jug2),  # Empty jug1
            (jug1, 0),  # Empty jug2
            (min(jug1 + jug2, jug1_capacity), max(0, jug1 + jug2 - jug1_capacity)),  # Pour jug2
into jug1
            (max(0, jug1 + jug2 - jug2_capacity), min(jug1 + jug2, jug2_capacity))  # Pour jug1
into jug2
        ]
        for state in new_states:
            if state not in visited:
                visited.add(state)
                queue.append(state)
    return None
# Example usage:
jug1_capacity = 4
jug2_capacity = 3
target_quantity = 2
solution = water_jug_problem(jug1_capacity, jug2_capacity, target_quantity)
print("Water Jug Problem in Python")
if solution:
    print("Solution:")
    for state in solution:
        print(state)
```

else:

    print("No solution found.")```

[11/25, 12:19] Krishnendu Samanta (SNU): import random

```
# Distance matrix representing distances between cities
# Replace this with the actual distance matrix for your problem
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

def total_distance(path):
    """Calculate the total distance traveled in the given path."""
    total = 0
    for i in range(len(path) - 1):
        total += distance_matrix[path[i]][path[i + 1]]
    total += distance_matrix[path[-1]][path[0]]  # Return to starting city
    return total

def hill_climbing_tsp(num_cities, max_iterations=10000):
    """Solve the TSP using the Hill Climbing algorithm."""
    current_path = list(range(num_cities))  # Initial solution: visiting cities in order
    current_distance = total_distance(current_path)

    for _ in range(max_iterations):
        # Generate a neighboring solution by swapping two random cities
```

```
        neighbor_path = current_path.copy()
        i, j = random.sample(range(num_cities), 2)
        neighbor_path[i], neighbor_path[j] = neighbor_path[j], neighbor_path[i]
        neighbor_distance = total_distance(neighbor_path)

        # If the neighbor solution is better, move to it
        if neighbor_distance < current_distance:
            current_path = neighbor_path
            current_distance = neighbor_distance

    return current_path, current_distance

# Example usage
num_cities = 4  # Number of cities in the TSP
optimal_path, optimal_distance = hill_climbing_tsp(num_cities)
print("Optimal path:", optimal_path)
print("Total distance:", optimal_distance)
[11/25, 12:19] Krishnendu Samanta (SNU): ` ` ` DFS in python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited
```

```
# Example usage
graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': [],
  'E': ['F'],
  'F': []
}

visited_nodes = dfs(graph, 'A')```
print("Visited nodes:", visited_nodes)
```

[11/25, 12:19] Krishnendu Samanta (SNU): ```AO* in python

```
class Node:
  def __init__(self, name, cost=0):
    self.name = name
    self.cost = cost
    self.children = []
    self.and_node = False
    self.solved = False

def ao_star(node):
  if node.solved:
    return node.cost
  if not node.children:
    return node.cost

  min_cost = float('inf')
```

```
    for children in node.children:
        if node.and_node:
            cost = sum(ao_star(child) for child in children)
        else:
            cost = min(ao_star(child) for child in children)
        if cost < min_cost:
            min_cost = cost
            node.best_children = children

    node.cost += min_cost
    node.solved = True
    return node.cost

# Example usage
a = Node('A')
b = Node('B', 1)
c = Node('C', 2)
d = Node('D', 4)
e = Node('E', 1)

a.children = [[b, c], [d, e]]
a.and_node = True

cost = ao_star(a)` ` `
print(f"Minimum cost to solve: {cost}")
[11/25, 12:19] Krishnendu Samanta (SNU): ` ` ` TOH in python
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
```

```python
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)

# Example usage
n = 3  # Number of disks```
tower_of_hanoi(n, 'A', 'B', 'C')
```

[11/25, 12:19] Krishnendu Samanta (SNU): ``` A* in python

```python
import heapq

class Node:
    def __init__(self, position, g, h, parent=None):
        self.position = position
        self.g = g  # Cost from start to the current node
        self.h = h  # Heuristic cost to the goal
        self.f = g + h  # Total cost
        self.parent = parent

    def __lt__(self, other):
        return self.f < other.f

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(grid, start, end):
    open_list = []
```

```
closed_list = set()
start_node = Node(start, 0, heuristic(start, end))
heapq.heappush(open_list, start_node)

while open_list:
  current_node = heapq.heappop(open_list)
  closed_list.add(current_node.position)

  if current_node.position == end:
    path = []
    while current_node:
      path.append(current_node.position)
      current_node = current_node.parent
    return path[::-1]  # Return reversed path

  neighbors = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # 4 possible movements
  for dx, dy in neighbors:
    neighbor_pos = (current_node.position[0] + dx, current_node.position[1] + dy)
    if (0 <= neighbor_pos[0] < len(grid) and
      0 <= neighbor_pos[1] < len(grid[0]) and
      grid[neighbor_pos[0]][neighbor_pos[1]] == 0 and
      neighbor_pos not in closed_list):

      g_cost = current_node.g + 1
      h_cost = heuristic(neighbor_pos, end)
      neighbor_node = Node(neighbor_pos, g_cost, h_cost, current_node)

      if neighbor_pos not in [node.position for node in open_list]:
```

```
                heappq.heappush(open_list, neighbor_node)
            else:
                for node in open_list:
                    if neighbor_pos == node.position and g_cost < node.g:
                        node.g = g_cost
                        node.f = g_cost + node.h
                        node.parent = current_node
                        heapq.heapify(open_list)
                        break

    return None  # No path found

# Example usage
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)
end = (4, 4)
path = astar(grid, start, end)```
print("Path:", path)
```