Assignment - 1

Name	ID Number	Email ID
Rajeev Kumar	12341700	rajeevk@iitbhilai.ac.in
Rahul Raj	12341680	rahulr@iitbhilai.ac.in
Aditya P. Rehpade	12340120	adityapr@iitbhilai.ac.in
Aditya Saini	12340130	adityasai@iitbhilai.ac.in

Course Code: CSL304
Course Name: Artificial Intelligence

Contents

1 Multi-Taxi Routing with Capacity-Constrained Roads

1.1 1. Problem Overview

We are given a city represented as a weighted undirected graph with 8 intersections (nodes) and roads (edges). Multiple taxis must pick up and drop off passengers (one passenger per taxi). Travel time is computed from distances and constant speed. Each road can have at most two taxis simultaneously; a third taxi must wait a fixed penalty before entering.

1.2 2. Rules & Cost Function

- Travel time (minutes): $\text{time} = \dfrac{\text{distance (km)}}{\text{speed (km/h)}} \times 60$.
- Congestion rule: at most 2 taxis on an edge at the same time. If a third arrives, it waits W minutes (e.g. 30).
- Objective: minimize total completion time (sum of all taxi times) and report the makespan (maximum taxi time).

1.3 3. Initial Setup

- Graph: $N = 8$, $M = 9$ (see assignment input).
- Example passenger trips (sample): $(2 \to 7)$, $(1 \to 8)$, $(3 \to 4)$.
- Constants: $W = 30$ minutes, $S = 40$ km/h.

1.4 4. Algorithm

1. Use A* (or Dijkstra) to compute shortest paths between pickups and drops. Heuristic for A*: Euclidean distance / speed (converted to minutes).
2. Simulate each taxi's timeline along chosen path; for each edge check for overlapping intervals with previously scheduled taxis.
3. If an entering taxi would create 3 simultaneous users on an edge, insert a wait at the node of arrival equal to W minutes.
4. Record per-edge intervals to enforce capacity.
5. Compute total times for each taxi; sum for total completion time and max for makespan.

1.5 5. Code Structure

- `TaxiRouter` class: holds nodes, edges, passengers.
- `astar / dijkstra` function: returns path and distance.
- `simulate_timeline`: enforces congestion and inserts waits.
- `solve & print_results`: top-level orchestration and formatted output.

1.6 6. Python Code

```
import heapq
import math

class TaxiRouter:
    def __init__(self):
        self.nodes = {}  # id -> (x, y)
        self.edges = {}  # (u, v) -> distance
        self.graph = {}  # adjacency list
        self.passengers = []  # [(pickup, dropoff), ...]
        self.speed = 40  # km/h
        self.wait_time = 30  # minutes
        self.time_per_km = 60 / 40  # 1.5 minutes per km

    def dijkstra_shortest_path(self, start, goal):
        """Find shortest path using Dijkstra's algorithm"""
        # Priority queue: (distance, current_node, path)
        pq = [(0, start, [start])]
        distances = {start: 0}
        visited = set()

        while pq:
            current_dist, current, path = heapq.heappop(pq)

            if current in visited:
                continue

            visited.add(current)

            # Found the goal!
            if current == goal:
                return path, current_dist

            # Explore neighbors
            if current in self.graph:
                for neighbor, edge_dist in self.graph[current]:
                    if neighbor not in visited:
                        new_dist = current_dist + edge_dist

                        if neighbor not in distances or new_dist
                           < distances[neighbor]:
                            distances[neighbor] = new_dist
                            new_path = path + [neighbor]
                            heapq.heappush(pq, (new_dist,
                               neighbor, new_path))

        return [], float('inf')  # No path found

    def check_congestion(self, edge, start_time, end_time,
       all_routes):
```

```
        """Check if edge would have 3+ taxis (congestion)"""
        u, v = edge
        count = 0

        for route_info in all_routes:
            for edge_data in route_info['edge_times']:
                route_u, route_v = edge_data['edge']
                route_start, route_end = edge_data['start'],
                    edge_data['end']

                # Same edge (either direction)?
                if (route_u == u and route_v == v) or (route_u
                    == v and route_v == u):
                    # Time intervals overlap?
                    if not (end_time <= route_start or
                        start_time >= route_end):
                        count += 1

        return count >= 2  # True if would be 3rd taxi

    def plan_route(self, taxi_id, pickup, dropoff,
        existing_routes):
        """Plan route for one taxi considering traffic"""
        # Get shortest path using Dijkstra
        path, total_distance =
            self.dijkstra_shortest_path(pickup, dropoff)

        if not path:
            print(f"ERROR: No path from {pickup} to {dropoff}")
            return None

        current_time = 0
        edge_times = []
        waits = []

        # Go through each edge in the path
        for i in range(len(path) - 1):
            u, v = path[i], path[i + 1]
            distance = self.edges[(u, v)]
            travel_time = distance * self.time_per_km

            # When would we start/end on this edge?
            start_time = current_time
            end_time = current_time + travel_time

            # Check for congestion
            if self.check_congestion((u, v), start_time,
                end_time, existing_routes):
                # Must wait!
                waits.append({'at_node': u, 'to_edge': v,
                    'wait_minutes': self.wait_time})
```

```
                current_time += self.wait_time
                start_time = current_time
                end_time = current_time + travel_time

            # Record this edge usage
            edge_times.append({
                'edge': (u, v),
                'start': start_time,
                'end': end_time,
                'distance': distance
            })

            current_time = end_time

        return {
            'taxi_id': taxi_id,
            'pickup': pickup,
            'dropoff': dropoff,
            'path': path,
            'total_time': current_time,
            'edge_times': edge_times,
            'waits': waits
        }

    def solve(self):
        """Main solver - plan routes for all taxis"""
        all_routes = []

        # Process each taxi/passenger pair
        for i, (pickup, dropoff) in enumerate(self.passengers):
            taxi_id = i + 1
            route = self.plan_route(taxi_id, pickup, dropoff,
                all_routes)

            if route:
                all_routes.append(route)

        return all_routes

    def print_results(self, routes):
        """Print results in the required format"""
        total_time = 0
        max_time = 0

        for route in routes:
            print(f"Taxi {route['taxi_id']}: Passenger
                {route['pickup']}->{route['dropoff']}")
            print(f"Route: {' -> '.join(map(str,
                route['path']))}")

            # Print any waits
```

```
            for wait in route['waits']:
                print(f"WAIT on edge ({wait['at_node']}->{wait['to_edge']}): {wait['wait_minutes']} minutes")

            print(f"Total time = {route['total_time']:.1f} minutes")
            print()

            total_time += route['total_time']
            max_time = max(max_time, route['total_time'])

        print(f"Total Completion Time = {total_time:.1f} minutes (span = {max_time:.1f} minutes)")
```

1.7 7. Results

```
Multi-Taxi Routing Solution
============================
Verifying shortest paths:
2->7: [2, 3, 5, 7], distance: 110
1->8: [1, 2, 3, 5, 7, 8], distance: 190
3->4: [3, 2, 4], distance: 110

Taxi 1: Passenger 2->7
Route: 2 -> 3 -> 5 -> 7
Total time = 165.0 minutes

Taxi 2: Passenger 1->8
Route: 1 -> 2 -> 3 -> 5 -> 7 -> 8
Total time = 285.0 minutes

Taxi 3: Passenger 3->4
Route: 3 -> 2 -> 4
WAIT on edge (3->2): 30 minutes
Total time = 195.0 minutes

Total Completion Time = 645.0 minutes (span = 285.0 minutes)
```

```
Multi-Taxi Routing Solution
============================
Choose mode: 1=Sample Test, 2=Custom Input: 2
Enter number of nodes: 8
Enter node coordinates as: id x y
1 0 0
2 0 5
3 5 1
4 5 2
5 7 3
6 1 8
7 2 9
8 2 7
Enter number of edges: 9
Enter edges as: u v distance
1 2 25
1 4 60
2 3 45
2 5 70
3 6 55
4 5 40
5 6 30
5 7 50
6 8 65
Enter number of passengers: 3
Enter passengers as: pickup dropoff
3 7
1 8
4 6
Enter taxi speed (km/h, default=40): 40
Enter wait time (minutes, default=30): 30

Solving routes with your custom input...

Taxi 1: Passenger 3->7
Route: 3 -> 6 -> 5 -> 7
Total time = 202.5 minutes

Taxi 2: Passenger 1->8
Route: 1 -> 2 -> 3 -> 6 -> 8
Total time = 285.0 minutes

Taxi 3: Passenger 4->6
Route: 4 -> 5 -> 6
Total time = 105.0 minutes

Total Completion Time = 592.5 minutes (span = 285.0 minutes)
```

1.8 8. Notes & Improvements

- Use A* with admissible heuristic for larger graphs.
- Consider simultaneous multi-agent planning rather than sequential planning for better makespan.
- Introduce probabilistic wait modeling for realistic traffic.

2 Reservoir Distribution with Valves

2.1 1. Problem Overview

Three reservoirs connected in a mesh. Each reservoir has capacity and initial/target amounts. Opening valves transfers water until destination full or source reaches safety threshold (20% capacity). Only one valve can be opened at a time. Find a shortest sequence of valve openings to reach target distribution.

2.2 2. Rules & Cost Function

- At each operation open valve $(i \to j)$ and transfer: $\min(\text{src} - 0.2C_i,\ C_j - \text{dst})$.
- State-space search; operations count is cost to minimize.

2.3 3. Initial Setup

- Capacities: (8.0, 5.0, 3.0)
- Initial: (8.0, 0.0, 0.0)
- Target: (2.4, 5.0, 0.6)

2.4 4. Algorithms

- **BFS**: explores by increasing number of operations; guarantees minimum valve openings.
- **DFS**: may find solution but not guaranteed minimal.
- **A***: uses heuristic $h(s) = \sum_i |s_i - t_i|$ (or scaled) to guide search.

2.5 5. Code Structure

- `State` representation: tuple of three floats.
- `transfers` generator: yields possible next states and operation labels.
- Search wrappers: `bfs`, `dfs`, `astar`.
- `reconstruct_path` for printing sequence of valve opens.

2.6 6. Python Code

```
from collections import deque
import heapq

def norm(state):
    return tuple(round(x, 4) for x in state)

def successors(state, capacities):
    C1, C2, C3 = capacities
    mins = [0.2*C1, 0.2*C2, 0.2*C3]  # safety thresholds
```

```
    succs = []
    S = list(state)
    for i in range(3):
        for j in range(3):
            if i == j:
                continue
            src = S[i]
            dst = S[j]
            src_min = mins[i]
            dst_cap = capacities[j]
            can_give = src - src_min
            can_accept = dst_cap - dst
            transfer = min(can_give, can_accept)
            if transfer > 1e-9:
                newS = S.copy()
                newS[i] -= transfer
                newS[j] += transfer
                action = (i, j, transfer)  # (src, dst, amount)
                succs.append((action, norm(newS)))
    return succs

def is_goal(state, target):
    return all(abs(state[i] - target[i]) < 1e-3 for i in
       range(3))

def bfs(start, target, capacities):
    start, target = norm(start), norm(target)
    q = deque()
    q.append((start, []))
    visited = set([start])
    while q:
        state, path = q.popleft()
        if is_goal(state, target):
            return path
        for action, ns in successors(state, capacities):
            if ns not in visited:
                visited.add(ns)
                q.append((ns, path + [action]))
    return None

def dfs(start, target, capacities, max_depth=20):
    start, target = norm(start), norm(target)
    stack = [(start, [])]
    visited = set([start])
    while stack:
        state, path = stack.pop()
        if is_goal(state, target):
            return path
        if len(path) >= max_depth:
            continue
        for action, ns in successors(state, capacities):
```

```
            if ns not in visited:
                visited.add(ns)
                stack.append((ns, path + [action]))
    return None

def heuristic(state, target):
    # Simple heuristic: total absolute difference
    return sum(abs(state[i] - target[i]) for i in range(3))

def astar(start, target, capacities):
    start, target = norm(start), norm(target)
    pq = []
    heapq.heappush(pq, (heuristic(start, target), 0, start, []))
    visited = set()
    while pq:
        est_cost, g, state, path = heapq.heappop(pq)
        if is_goal(state, target):
            return path
        if state in visited:
            continue
        visited.add(state)
        for action, ns in successors(state, capacities):
            if ns not in visited:
                new_g = g + 1
                f = new_g + heuristic(ns, target)
                heapq.heappush(pq, (f, new_g, ns, path +
                   [action]))
    return None

def print_solution(name, start, capacities, path, target):
    print(f"\n==== {name} with step calculations ====")
    if path is None:
        print("No solution found.")
        return

    state = list(start)
    for step, (i, j, transfer) in enumerate(path, 1):
        state[i] -= transfer
        state[j] += transfer
        print(f"Operation {step}     Open valve ({i+1}->{j+1})")
        print(f"  Transfer = {transfer:.2f}")
        print(f"  New state = ({state[0]:.2f}, {state[1]:.2f},
           {state[2]:.2f})\n")

    print(f"Reached target = {tuple(round(x,2) for x in state)}")
    print(f"Number of valve operations = {len(path)}")
```

2.7 7. Results

```
==== BFS with step calculations ====
Operation 1 — Open valve (1->2)
  Transfer = 5.00
  New state = (3.00, 5.00, 0.00)

Operation 2 — Open valve (1->3)
  Transfer = 1.40
  New state = (1.60, 5.00, 1.40)

Operation 3 — Open valve (3->1)
  Transfer = 0.80
  New state = (2.40, 5.00, 0.60)

Reached target = (2.4, 5.0, 0.6)
Number of valve operations = 3
```

```
==== DFS with step calculations ====
Operation 1 — Open valve (1->3)
  Transfer = 3.00
  New state = (5.00, 0.00, 3.00)

Operation 2 — Open valve (3->2)
  Transfer = 2.40
  New state = (5.00, 2.40, 0.60)

Operation 3 — Open valve (1->2)
  Transfer = 2.60
  New state = (2.40, 5.00, 0.60)

Reached target = (2.4, 5.0, 0.6)
Number of valve operations = 3
```

```
==== A* with step calculations ====
Operation 1 — Open valve (1->2)
  Transfer = 5.00
  New state = (3.00, 5.00, 0.00)

Operation 2 — Open valve (1->3)
  Transfer = 1.40
  New state = (1.60, 5.00, 1.40)

Operation 3 — Open valve (3->1)
  Transfer = 0.80
  New state = (2.40, 5.00, 0.60)

Reached target = (2.4, 5.0, 0.6)
Number of valve operations = 3
```

2.8 8. Notes & Improvements

- Heuristic design for A* can greatly reduce nodes expanded.
- Use exact rational arithmetic or rounding tolerance to avoid floating-point branching issues.

3 The Game of Tic-Tac-Toe

3.1 1. Problem Overview

Given a partially filled 3x3 board where X (MAX) plays next, build the game tree, apply Minimax and Alpha-Beta pruning using the provided evaluation function:

$$\text{Evaluation}(s) = 8X_3(s) + 3X_2(s) + X_1(s) - (8O_3(s) + 3O_2(s) + O_1(s))$$

Count lines with exactly n pieces and no opponent pieces where $X_n(s)$ and $O_n(s)$.

3.2 2. Rules & Cost Function

- Terminal states: X wins → large positive score, O wins → large negative score, else evaluate using formula above.
- Minimax explores full game tree; alpha-beta prunes branches using bounds.

3.3 3. Initial Setup

Initial board (rows 0..2):

```
_  X  _
O  _  _
X  _  O
```

3.4 4. Algorithm

- Implement Minimax (complete), track nodes visited.
- Implement Alpha-Beta with identical evaluation but prune where possible, track nodes visited and pruning occurrences.
- Compare best moves and node counts.

3.5 5. Code Structure

- Helper functions: `print_board`, `get_empty_positions`, `check_winner`, `evaluate_board`
- Search functions: `minimax`, `alpha_beta`, and wrapper functions to report best move and stats.

3.6 6. Python Code (Provided)

```
def print_board(board):
    """Helper function to display the board"""
    for i in range(3):
        row = ""
        for j in range(3):
            if board[i][j] == 'X':
                row += "X"
            elif board[i][j] == 'O':
```

```
                row += "O"
            else:
                row += "_"
            if j < 2:
                row += " | "
        print(row)
        if i < 2:
            print("---------")
    print()

def get_empty_positions(board):
    """Find all empty positions on the board"""
    positions = []
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                positions.append((i, j))
    return positions

def check_winner(board):
    """Check if someone has won the game"""
    # Check rows
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]

    # Check columns
    for j in range(3):
        if board[0][j] == board[1][j] == board[2][j] != ' ':
            return board[0][j]

    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]

    return None

def is_game_over(board):
    """Check if game is finished"""
    if check_winner(board) is not None:
        return True
    # Check if board is full
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                return False
    return True

def count_lines_with_n_pieces(board, player, n):
```

```
60      """Count lines with exactly n pieces of player and no
            opponent pieces"""
61      count = 0
62      opponent = 'O' if player == 'X' else 'X'
63
64      # Check all 8 lines (3 rows, 3 columns, 2 diagonals)
65      lines = []
66
67      # Add rows
68      for i in range(3):
69          lines.append([board[i][0], board[i][1], board[i][2]])
70
71      # Add columns
72      for j in range(3):
73          lines.append([board[0][j], board[1][j], board[2][j]])
74
75      # Add diagonals
76      lines.append([board[0][0], board[1][1], board[2][2]])
77      lines.append([board[0][2], board[1][1], board[2][0]])
78
79      # Count lines with exactly n player pieces and no opponent
            pieces
80      for line in lines:
81          player_count = line.count(player)
82          opponent_count = line.count(opponent)
83          if player_count == n and opponent_count == 0:
84              count += 1
85
86      return count
87
88  def evaluate_board(board):
89      """Evaluation function from the assignment"""
90      # Check for terminal states first
91      winner = check_winner(board)
92      if winner == 'X':
93          return 1000  # X wins
94      elif winner == 'O':
95          return -1000  # O wins
96
97      # Calculate evaluation using the given formula
98
99      X3 = count_lines_with_n_pieces(board, 'X', 3)
100     X2 = count_lines_with_n_pieces(board, 'X', 2)
101     X1 = count_lines_with_n_pieces(board, 'X', 1)
102
103     O3 = count_lines_with_n_pieces(board, 'O', 3)
104     O2 = count_lines_with_n_pieces(board, 'O', 2)
105     O1 = count_lines_with_n_pieces(board, 'O', 1)
106
107     evaluation = (8*X3 + 3*X2 + X1) - (8*O3 + 3*O2 + O1)
108     return evaluation
```

```
109
110 def minimax(board, depth, is_maximizing):
111     """Basic minimax algorithm"""
112     global minimax_nodes_visited
113     minimax_nodes_visited += 1
114
115     # Base case - terminal node
116     if is_game_over(board):
117         return evaluate_board(board)
118
119     if is_maximizing:  # X's turn (MAX player)
120         best_value = float('-inf')
121         empty_positions = get_empty_positions(board)
122
123         for pos in empty_positions:
124             i, j = pos
125             board[i][j] = 'X'  # Make move
126             value = minimax(board, depth + 1, False)
127             board[i][j] = ' '  # Undo move
128             best_value = max(best_value, value)
129
130         return best_value
131
132     else:  # O's turn (MIN player)
133         best_value = float('inf')
134         empty_positions = get_empty_positions(board)
135
136         for pos in empty_positions:
137             i, j = pos
138             board[i][j] = 'O'  # Make move
139             value = minimax(board, depth + 1, True)
140             board[i][j] = ' '  # Undo move
141             best_value = min(best_value, value)
142
143         return best_value
144
145 def minimax_with_move(board):
146     """Find best move using minimax"""
147     best_value = float('-inf')
148     best_move = None
149     empty_positions = get_empty_positions(board)
150
151     print("Evaluating possible moves for X:")
152     for pos in empty_positions:
153         i, j = pos
154         board[i][j] = 'X'  # Try this move
155         value = minimax(board, 0, False)  # Get minimax value
156         board[i][j] = ' '  # Undo move
157
158         print(f"Move ({i},{j}): Value = {value}")
159
```

```
160             if value > best_value:
161                 best_value = value
162                 best_move = pos
163 
164     return best_move, best_value
165 
166 def alpha_beta(board, depth, alpha, beta, is_maximizing):
167     """Minimax with alpha-beta pruning"""
168     global alpha_beta_nodes_visited, pruning_count
169     alpha_beta_nodes_visited += 1
170 
171     # Base case - terminal node
172     if is_game_over(board):
173         return evaluate_board(board)
174 
175     if is_maximizing:  # X's turn (MAX player)
176         best_value = float('-inf')
177         empty_positions = get_empty_positions(board)
178 
179         for pos in empty_positions:
180             i, j = pos
181             board[i][j] = 'X'  # Make move
182             value = alpha_beta(board, depth + 1, alpha, beta,
                    False)
183             board[i][j] = ' '  # Undo move
184 
185             best_value = max(best_value, value)
186             alpha = max(alpha, best_value)
187 
188             if beta <= alpha:
189                 pruning_count += 1
190                 break  # Beta cutoff (pruning)
191 
192         return best_value
193 
194     else:  # O's turn (MIN player)
195         best_value = float('inf')
196         empty_positions = get_empty_positions(board)
197 
198         for pos in empty_positions:
199             i, j = pos
200             board[i][j] = 'O'  # Make move
201             value = alpha_beta(board, depth + 1, alpha, beta,
                    True)
202             board[i][j] = ' '  # Undo move
203 
204             best_value = min(best_value, value)
205             beta = min(beta, best_value)
206 
207             if beta <= alpha:
208                 pruning_count += 1
```

```
                break  # Alpha cutoff (pruning)

        return best_value

def alpha_beta_with_move(board):
    """Find best move using alpha-beta pruning"""
    best_value = float('-inf')
    best_move = None
    empty_positions = get_empty_positions(board)
    alpha = float('-inf')
    beta = float('inf')

    print("Evaluating possible moves for X with Alpha-Beta:")
    for pos in empty_positions:
        i, j = pos
        board[i][j] = 'X'  # Try this move
        value = alpha_beta(board, 0, alpha, beta, False)
        board[i][j] = ' '  # Undo move

        print(f"Move ({i},{j}): Value = {value}")

        if value > best_value:
            best_value = value
            best_move = pos

    return best_move, best_value

# Main solution
def solution():
    """Solve the tic-tac-toe assignment"""
    global minimax_nodes_visited, alpha_beta_nodes_visited,
       pruning_count

    board = [
        [' ', 'X', ' '],  # Row 0
        ['O', ' ', ' '],  # Row 1
        ['X', ' ', 'O']   # Row 2
    ]

    print("=== TIC-TAC-TOE ASSIGNMENT SOLUTION ===")
    print("\nInitial Board State:")
    print_board(board)

    print(f"Initial evaluation: {evaluate_board(board)}")

    # Reset counters
    minimax_nodes_visited = 0
    alpha_beta_nodes_visited = 0
    pruning_count = 0

    print("\n--- MINIMAX ALGORITHM ---")
```

```
    best_move_minimax, best_value_minimax =
        minimax_with_move(board)
    print(f"\nMinimax Result:")
    print(f"Best move for X: Row {best_move_minimax[0]}, Column
        {best_move_minimax[1]}")
    print(f"Best value: {best_value_minimax}")
    print(f"Nodes visited: {minimax_nodes_visited}")

    print("\n--- ALPHA-BETA PRUNING ALGORITHM ---")
    best_move_ab, best_value_ab = alpha_beta_with_move(board)
    print(f"\nAlpha-Beta Result:")
    print(f"Best move for X: Row {best_move_ab[0]}, Column
        {best_move_ab[1]}")
    print(f"Best value: {best_value_ab}")
    print(f"Nodes visited: {alpha_beta_nodes_visited}")
    print(f"Pruning occurred: {pruning_count} times")

    print(f"\nEfficiency: Alpha-beta visited
        {minimax_nodes_visited - alpha_beta_nodes_visited} fewer
        nodes")
    if minimax_nodes_visited > 0:
        savings = ((minimax_nodes_visited -
            alpha_beta_nodes_visited) / minimax_nodes_visited) *
            100
        print(f"Space savings: {savings:.1f}%")

    # Show final board with optimal move
    print(f"\nBoard after optimal move ({best_move_minimax[0]},
        {best_move_minimax[1]}):")
    board[best_move_minimax[0]][best_move_minimax[1]] = 'X'
    print_board(board)

# Global counters for analysis
minimax_nodes_visited = 0
alpha_beta_nodes_visited = 0
pruning_count = 0

# Run the solution
if __name__ == "__main__":
    solution()
```

3.7 7. Results

```
--- MINIMAX ALGORITHM ---
Evaluating possible moves for X:
Move (0,0): Value = 0
Move (0,2): Value = 1000
Move (1,1): Value = 1000
Move (1,2): Value = 0
Move (2,1): Value = -1000

Minimax Result:
Best move for X: Row 0, Column 2
Best value: 1000
Nodes visited: 225
```

(a) MinMax Algorithm.

```
--- ALPHA-BETA PRUNING ALGORITHM ---
Evaluating possible moves for X with Alpha-Beta:
Move (0,0): Value = 0
Move (0,2): Value = 1000
Move (1,1): Value = 1000
Move (1,2): Value = 0
Move (2,1): Value = -1000

Alpha-Beta Result:
Best move for X: Row 0, Column 2
Best value: 1000
Nodes visited: 135
Pruning occurred: 39 times
```

(b) Alpha-Beta Pruning Algorithm.

3.8 8. Notes & Improvements

- Alpha-Beta pruning reduces nodes visited; ordering moves (best-first) improves pruning.
- For larger games, add transposition tables and iterative deepening.

4 Chip Placement Optimization using Integer Descent (Column-wise Restricted Movement)

4.1 1. Problem Overview

Place rectangular chips on a discrete 10×10 grid. Chips are taller than wide and locked to rows (fixed y). Chips can move horizontally by integer steps only. Some chips are connected by a netlist. The objective is to minimize a conflict score defined as wiring cost + overlap cell count using integer descent.

4.2 2. Conflict Score

Wiring Cost for connected chips (c_i, c_j):

$$\text{WiringCost}(c_i, c_j) = \max\big(0,\ x_j - (x_i + w_i),\ x_i - (x_j + w_j)\big) + |y_i - y_j|$$

(horizontal gap if disjoint) + vertical row difference.

Overlap Cost: number of grid cells covered by more than one chip.

Total conflict = sum of wiring costs over connections + total overlap blocks.

4.3 3. Initial Setup

- Grid: 10×10.
- Chips:
  - c1: 2x4 @ row 0, x=0
  - c2: 2x5 @ row 1, x=1
  - c3: 1x3 @ row 0, x=1
  - c4: 2x5 @ row 4, x=2
  - c5: 2x4 @ row 3, x=3
  - c6: 1x4 @ row 2, x=2
  - c7: 2x5 @ row 5, x=0
  - c8: 1x3 @ row 6, x=4
- Connections: (1,2),(2,6),(2,3),(3,5),(4,5),(5,6),(1,6),(7,4),(7,2),(8,5)

4.4 4. Algorithm (Integer Descent)

1. Compute initial conflict score.
2. For each chip c_i, iterate all valid integer x positions (0..grid_width - w_i).
2. Compute conflict score for the tentative position.
3. Keep the single best improvement across all chips (greedy single-move).
4. Apply that move and repeat until no improvement found (convergence).

4.5 5. Code Structure

- `Chip` class: id, w, h, y, x.
- `ChipPlacementOptimizer`:
  - `calculate_wiring_cost`
  - `calculate_overlap_blocks`
  - `calculate_total_conflict`
  - `integer_descent_step` and `optimize`
  - Visualization utilities: grid plotting and convergence curve.

4.6 6. Python Code

```
"""
Q)4 Chip Placement Optimization using Integer Descent
Column-wise Restricted Movement

This program implements an integer descent algorithm to optimize chip placement
on a grid, minimizing wire lengths between connected chips and overlap penalties.
"""

import numpy as np
import matplotlib.pyplot as plt
from typing import List, Tuple, Dict
import copy

class Chip:
    """
    Represents a single chip/module to be placed on the grid.

    Attributes:
        id (int): Unique identifier for the chip
        width (int): Width of the chip
        height (int): Height of the chip
        locked_row (int): The row this chip is locked to
        x (int): Current x-coordinate (column) position
    """
    def __init__(self, chip_id: int, width: int, height: int, locked_row: int, initial_x: int):
        self.id = chip_id
        self.width = width
        self.height = height
        self.locked_row = locked_row
        self.x = initial_x

    def __repr__(self):
        return f"Chip{self.id}(w={self.width}, h={self.height}, row={self.locked_row}, x={self.x})"
```

```
34
35 class ChipPlacementOptimizer:
36     """
37     Implements integer descent optimization for chip placement
       problem.
38
39     The optimizer minimizes a conflict score consisting of:
40     1. Wiring cost: Manhattan distance between connected chips
41     2. Overlap cost: Number of grid cells occupied by multiple
       chips
42     """
43
44     def __init__(self, grid_width: int = 10, grid_height: int =
       10):
45         """
46         Initialize the chip placement optimizer.
47
48         Args:
49             grid_width (int): Width of the placement grid
50             grid_height (int): Height of the placement grid
51         """
52         self.grid_width = grid_width
53         self.grid_height = grid_height
54         self.chips = []
55         self.connections = []
56         self.conflict_history = []
57
58     def setup_initial_state(self):
59         """
60         Set up the initial chip configuration as specified in
           the problem.
61         """
62         # Create chips with their specifications (id, width,
           height, locked_row, initial_x)
63         chip_specs = [
64             (1, 2, 4, 0, 0),  # c1: 2 4  at row y=0, x=0
65             (2, 2, 5, 1, 1),  # c2: 2 5  at row y=1, x=1
66             (3, 1, 3, 0, 1),  # c3: 1 3  at row y=0, x=1
67             (4, 2, 5, 4, 2),  # c4: 2 5  at row y=4, x=2
68             (5, 2, 4, 3, 3),  # c5: 2 4  at row y=3, x=3
69             (6, 1, 4, 2, 2),  # c6: 1 4  at row y=2, x=2
70             (7, 2, 5, 5, 0),  # c7: 2 5  at row y=5, x=0
71             (8, 1, 3, 6, 4),  # c8: 1 3  at row y=6, x=4
72         ]
73
74         self.chips = []
75         for chip_id, width, height, locked_row, initial_x in
           chip_specs:
76             chip = Chip(chip_id, width, height, locked_row,
               initial_x)
77             self.chips.append(chip)
```

```
78
79          # Define connections (netlist) as pairs of chip IDs
80          self.connections = [
81              (1, 2), (2, 6), (2, 3), (3, 5), (4, 5),
82              (5, 6), (1, 6), (7, 4), (7, 2), (8, 5)
83          ]
84
85          print("Initial chip configuration:")
86          for chip in self.chips:
87              print(f"  {chip}")
88
89      def get_chip_by_id(self, chip_id: int) -> Chip:
90          """Get a chip object by its ID."""
91          for chip in self.chips:
92              if chip.id == chip_id:
93                  return chip
94          return None
95
96      def calculate_wiring_cost(self, c_i: Chip, c_j: Chip) -> int:
97          """
98          Calculate the wiring cost between two connected chips.
99
100         The cost is the Manhattan distance between chip
              boundaries plus
101         the vertical distance between their locked rows.
102
103         Args:
104             c_i (Chip): First chip
105             c_j (Chip): Second chip
106
107         Returns:
108             int: Wiring cost
109         """
110         # Calculate horizontal distance between chips
111         x_i_right = c_i.x + c_i.width
112         x_j_right = c_j.x + c_j.width
113
114         # Manhattan distance formula as specified
115         horizontal_dist = max(0, c_j.x - x_i_right, c_i.x -
              x_j_right)
116         vertical_dist = abs(c_i.locked_row - c_j.locked_row)
117
118         return horizontal_dist + vertical_dist
119
120     def get_grid_representation(self) -> np.ndarray:
121         """
122         Create a grid representation showing which chips occupy
              which cells.
123
124         Returns:
```

```
125             np.ndarray: Grid where each cell contains the chip
                    ID (0 if empty)
126         """
127         grid = np.zeros((self.grid_height, self.grid_width),
                dtype=int)
128 
129         for chip in self.chips:
130             # Mark all cells occupied by this chip
131             for dy in range(chip.height):
132                 for dx in range(chip.width):
133                     y = chip.locked_row + dy
134                     x = chip.x + dx
135                     if 0 <= y < self.grid_height and 0 <= x <
                            self.grid_width:
136                         if grid[y, x] == 0:
137                             grid[y, x] = chip.id
138                         else:
139                             # Mark overlap with negative value
140                             grid[y, x] = -1
141 
142         return grid
143 
144     def calculate_overlap_blocks(self) -> int:
145         """
146         Calculate the number of grid blocks occupied by more
                than one chip.
147 
148         Returns:
149             int: Number of overlapping grid cells
150         """
151         overlap_count = 0
152         grid_occupancy = {}  # Dictionary to track which chips
                occupy each cell
153 
154         for chip in self.chips:
155             for dy in range(chip.height):
156                 for dx in range(chip.width):
157                     y = chip.locked_row + dy
158                     x = chip.x + dx
159                     if 0 <= y < self.grid_height and 0 <= x <
                            self.grid_width:
160                         cell = (y, x)
161                         if cell not in grid_occupancy:
162                             grid_occupancy[cell] = []
163                         grid_occupancy[cell].append(chip.id)
164 
165         # Count cells with more than one chip
166         for cell, chip_ids in grid_occupancy.items():
167             if len(chip_ids) > 1:
168                 overlap_count += 1
169 
```

```
170         return overlap_count
171
172     def calculate_total_conflict(self) -> int:
173         """
174         Calculate the total conflict score (wiring cost +
                overlap cost).
175
176         Returns:
177             int: Total conflict score
178         """
179         # Calculate wiring cost for all connections
180         total_wiring_cost = 0
181         for c_i_id, c_j_id in self.connections:
182             c_i = self.get_chip_by_id(c_i_id)
183             c_j = self.get_chip_by_id(c_j_id)
184             if c_i and c_j:
185                 total_wiring_cost +=
                        self.calculate_wiring_cost(c_i, c_j)
186
187         # Calculate overlap cost
188         overlap_blocks = self.calculate_overlap_blocks()
189
190         total_conflict = total_wiring_cost + overlap_blocks
191
192         return total_conflict
193
194     def get_valid_moves(self, chip: Chip) -> List[int]:
195         """
196         Get all valid integer x-positions for a chip (within
                grid bounds).
197
198         Args:
199             chip (Chip): The chip to move
200
201         Returns:
202             List[int]: List of valid x-positions
203         """
204         valid_positions = []
205         for x in range(self.grid_width - chip.width + 1):
206             valid_positions.append(x)
207         return valid_positions
208
209     def integer_descent_step(self) -> bool:
210         """
211         Perform one step of integer descent optimization.
212
213         Returns:
214             bool: True if improvement was found, False otherwise
215         """
216         current_conflict = self.calculate_total_conflict()
217         best_conflict = current_conflict
```

```
        best_move = None

        # Try moving each chip to all possible integer positions
        for chip in self.chips:
            original_x = chip.x
            valid_positions = self.get_valid_moves(chip)

            for new_x in valid_positions:
                if new_x == original_x:
                    continue

                # Try the move
                chip.x = new_x
                new_conflict = self.calculate_total_conflict()

                # Check if this is the best move so far
                if new_conflict < best_conflict:
                    best_conflict = new_conflict
                    best_move = (chip, new_x)

                # Restore original position
                chip.x = original_x

        # Apply the best move if found
        if best_move:
            chip, new_x = best_move
            print(f"Moving Chip {chip.id} from x={chip.x} to
                x={new_x} (conflict: {current_conflict} ->
                {best_conflict})")
            chip.x = new_x
            return True

        return False

    def optimize(self, max_iterations: int = 100) -> Dict:
        """
        Run the integer descent optimization algorithm.

        Args:
            max_iterations (int): Maximum number of optimization
                iterations

        Returns:
            Dict: Optimization results including final positions
                and conflict score
        """
        print("\n" + "="*60)
        print("Starting Integer Descent Optimization")
        print("="*60)

        # Record initial conflict
```

```
        initial_conflict = self.calculate_total_conflict()
        self.conflict_history = [initial_conflict]
        print(f"Initial conflict score: {initial_conflict}")

        # Run integer descent
        iteration = 0
        while iteration < max_iterations:
            iteration += 1

            # Perform one descent step
            improved = self.integer_descent_step()

            # Record conflict score
            current_conflict = self.calculate_total_conflict()
            self.conflict_history.append(current_conflict)

            # Check for convergence
            if not improved:
                print(f"\nConverged at iteration {iteration}")
                break

        # Get final results
        final_conflict = self.calculate_total_conflict()

        print("\n" + "="*60)
        print("Optimization Complete")
        print("="*60)
        print(f"Final conflict score: {final_conflict}")
        print(f"Improvement: {initial_conflict - final_conflict}
            ({(initial_conflict -
            final_conflict)/initial_conflict*100:.1f}%)")
        print(f"Total iterations: {iteration}")

        # Print final chip positions
        print("\nFinal chip positions:")
        for chip in sorted(self.chips, key=lambda c: c.id):
            print(f"  Chip {chip.id}: x={chip.x},
                y={chip.locked_row} (row-locked)")

        return {
            'initial_conflict': initial_conflict,
            'final_conflict': final_conflict,
            'iterations': iteration,
            'conflict_history': self.conflict_history,
            'final_positions': [(chip.id, chip.x,
                chip.locked_row) for chip in self.chips]
        }

    def visualize_placement(self, title: str = "Chip Placement"):
        """
        Visualize the current chip placement on the grid.
```

```
312
313         Args:
314             title (str): Title for the visualization
315         """
316         fig, ax = plt.subplots(1, 1, figsize=(10, 10))
317
318         # Draw grid
319         for i in range(self.grid_height + 1):
320             ax.axhline(y=i, color='lightgray', linewidth=0.5)
321         for i in range(self.grid_width + 1):
322             ax.axvline(x=i, color='lightgray', linewidth=0.5)
323
324         # Color map for chips
325         colors = plt.cm.Set3(np.linspace(0, 1, len(self.chips)))
326
327         # Draw chips
328         for idx, chip in enumerate(self.chips):
329             rect = plt.Rectangle((chip.x, chip.locked_row),
330                                  chip.width, chip.height,
331                                  facecolor=colors[idx],
332                                  edgecolor='black',
333                                  linewidth=2,
334                                  alpha=0.7)
335             ax.add_patch(rect)
336
337             # Add chip label
338             ax.text(chip.x + chip.width/2, chip.locked_row +
                        chip.height/2,
339                     f'C{chip.id}', ha='center', va='center',
340                     fontsize=12, fontweight='bold')
341
342         # Draw connections
343         for c_i_id, c_j_id in self.connections:
344             c_i = self.get_chip_by_id(c_i_id)
345             c_j = self.get_chip_by_id(c_j_id)
346             if c_i and c_j:
347                 # Connection from center of chips
348                 x1 = c_i.x + c_i.width/2
349                 y1 = c_i.locked_row + c_i.height/2
350                 x2 = c_j.x + c_j.width/2
351                 y2 = c_j.locked_row + c_j.height/2
352                 ax.plot([x1, x2], [y1, y2], 'r--', alpha=0.3,
                            linewidth=1)
353
354         ax.set_xlim(0, self.grid_width)
355         ax.set_ylim(0, self.grid_height)
356         ax.set_aspect('equal')
357         ax.invert_yaxis()  # Invert y-axis to match matrix
                representation
358         ax.set_xlabel('X (Column)')
359         ax.set_ylabel('Y (Row)')
```

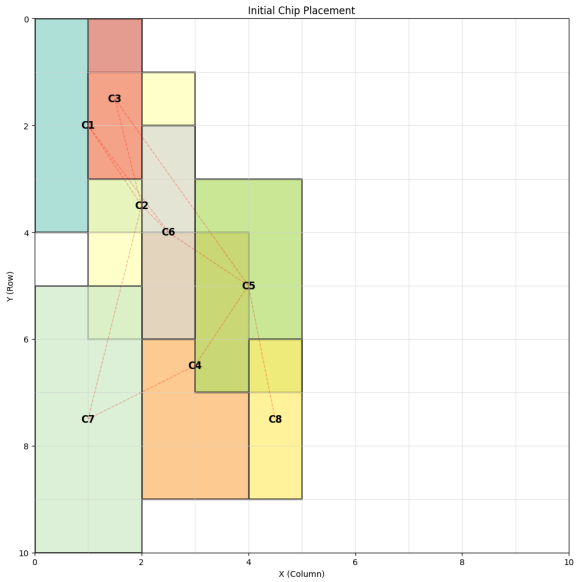
```
360         ax.set_title(title)
361
362         plt.tight_layout()
363         plt.show()
364
365     def plot_convergence(self):
366         """Plot the convergence of the conflict score over
                iterations."""
367         plt.figure(figsize=(10, 6))
368         plt.plot(self.conflict_history, marker='o', linewidth=2,
                markersize=6)
369         plt.xlabel('Iteration')
370         plt.ylabel('Conflict Score')
371         plt.title('Integer Descent Convergence')
372         plt.grid(True, alpha=0.3)
373         plt.tight_layout()
374         plt.show()
375
376
377 def main():
378     """
379     Main function to run the chip placement optimization.
380     """
381     # Create optimizer instance
382     optimizer = ChipPlacementOptimizer(grid_width=10,
            grid_height=10)
383
384     # Set up initial state
385     optimizer.setup_initial_state()
386
387     # Visualize initial placement
388     print("\nVisualizing initial placement...")
389     optimizer.visualize_placement("Initial Chip Placement")
390
391     # Run optimization
392     results = optimizer.optimize(max_iterations=100)
393
394     # Visualize final placement
395     print("\nVisualizing final placement...")
396     optimizer.visualize_placement("Optimized Chip Placement")
397
398     # Plot convergence
399     print("\nPlotting convergence...")
400     optimizer.plot_convergence()
401
402     # Print detailed overlap analysis
403     print("\n" + "="*60)
404     print("Detailed Analysis")
405     print("="*60)
406
407     overlap_blocks = optimizer.calculate_overlap_blocks()
```

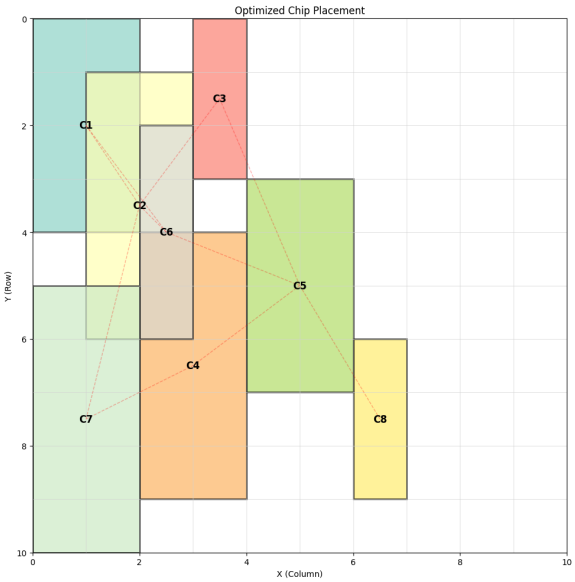
```
    print(f"Number of overlapping blocks: {overlap_blocks}")

    # Calculate and print wiring costs for each connection
    print("\nIndividual wiring costs:")
    total_wiring = 0
    for c_i_id, c_j_id in optimizer.connections:
        c_i = optimizer.get_chip_by_id(c_i_id)
        c_j = optimizer.get_chip_by_id(c_j_id)
        if c_i and c_j:
            cost = optimizer.calculate_wiring_cost(c_i, c_j)
            total_wiring += cost
            print(f"  Connection ({c_i_id}, {c_j_id}): {cost}")

    print(f"\nTotal wiring cost: {total_wiring}")
    print(f"Total overlap cost: {overlap_blocks}")
    print(f"Total conflict score: {total_wiring +
       overlap_blocks}")

    return results


if __name__ == "__main__":
    results = main()
```

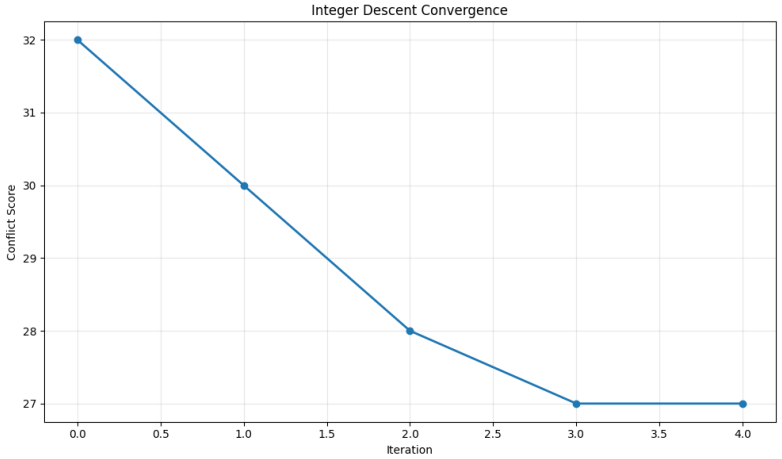
4.7 7. Results

```
============================================================
Starting Integer Descent Optimization
============================================================
Initial conflict score: 32
Moving Chip 3 from x=1 to x=3 (conflict: 32 -> 30)
Moving Chip 5 from x=3 to x=4 (conflict: 30 -> 28)
Moving Chip 8 from x=4 to x=6 (conflict: 28 -> 27)

Converged at iteration 4

============================================================
Optimization Complete
============================================================
Final conflict score: 27
Improvement: 5 (15.6%)
Total iterations: 4

Final chip positions:
  Chip 1: x=0, y=0 (row-locked)
  Chip 2: x=1, y=1 (row-locked)
  Chip 3: x=3, y=0 (row-locked)
  Chip 4: x=2, y=4 (row-locked)
  Chip 5: x=4, y=3 (row-locked)
  Chip 6: x=2, y=2 (row-locked)
  Chip 7: x=0, y=5 (row-locked)
  Chip 8: x=6, y=6 (row-locked)
```

```
============================================================
Detailed Analysis
============================================================
Number of overlapping blocks: 8

Individual wiring costs:
  Connection (1, 2): 1
  Connection (2, 6): 1
  Connection (2, 3): 1
  Connection (3, 5): 3
  Connection (4, 5): 1
  Connection (5, 6): 2
  Connection (1, 6): 2
  Connection (7, 4): 1
  Connection (7, 2): 4
  Connection (8, 5): 3

Total wiring cost: 19
Total overlap cost: 8
Total conflict score: 27
```







4.8 8. Notes & Improvements

- Integer descent is fast and guaranteed to converge to a local minimum.
- To escape local minima, consider random restarts or simulated annealing.
- Introduce weights to balance wiring vs overlap costs if necessary.