Multi-Taxi Routing with Capacity-Constrained Roads

Assignment 1 - CSL304 AI

1 Problem Overview

Problem Statement

Objective: Plan optimal routes for multiple taxis in a city with capacity-constrained roads to minimize total completion time.

Key Constraints:

- Each taxi carries exactly one passenger
- At most 2 taxis per road simultaneously
- 30-minute wait penalty for congestion
- Constant speed of 40 km/h

1.1 Problem Formulation

Given:

- Weighted undirected graph $G = (V, E)$ with $|V| = N = 8$ nodes
- Edge weights represent distances in kilometers
- P taxi-passenger pairs with pickup and drop-off locations
- Speed $S = 40$ km/h, Wait time $W = 30$ minutes

Travel Time Formula:

$$\text{Travel Time (minutes)} = \frac{\text{Distance (km)} \times 60}{\text{Speed (km/h)}} = \frac{d \times 60}{40} = 1.5 \times d$$

2 Solution Methodology

Solution Approach

Three-Phase Approach:

1. **Preprocessing:** Compute optimal heuristics using Dijkstra's algorithm
2. **Path Planning:** Use A* search with precomputed heuristics
3. **Simulation:** Event-driven scheduling with congestion handling

2.1 Phase 1: Heuristic Preprocessing

Algorithm Details

Dijkstra-based Heuristic Computation
For each unique destination d, we compute the shortest path from every node to d using Dijkstra's algorithm. This provides an **admissible and consistent heuristic** for A* search.

Algorithm 1 Precompute Heuristics

```
unique_goals ← {d|(_, d) ∈ trips}
heuristics ← {}
for each goal ∈ unique_goals do
    heuristics[goal] ← dijkstra_all_sources(goal, G, time_per_km)
end for
return heuristics
```

Why this heuristic works:

- **Admissible:** Never overestimates the actual shortest path cost
- **Consistent:** Triangle inequality holds: $h(n) \leq c(n, n') + h(n')$
- **Optimal:** Provides exact shortest path distances as heuristic values

2.2 Phase 2: A* Path Planning

```python
def a_star_with_precomputed_heuristic(start, goal, graph, time_per_km, h):
    open_heap = []
    g_scores = {start: 0.0}
    f0 = g_scores[start] + h.get(start, float('inf'))
    heapq.heappush(open_heap, (f0, 0.0, start, [start]))

    while open_heap:
        f, g, node, path = heapq.heappop(open_heap)

        # Skip if we've found a better path to this node
        if g > g_scores.get(node, float('inf')) + 1e-9:
            continue

        if node == goal:
            return path, g

        for neighbor, distance in graph[node]:
            travel_time = distance * time_per_km
            tentative_g = g + travel_time

            if tentative_g + 1e-9 < g_scores.get(neighbor, float('inf')):
                g_scores[neighbor] = tentative_g
                f_score = tentative_g + h.get(neighbor, float('inf'))
                heapq.heappush(open_heap, (f_score, tentative_g, neighbor, path + [neighbor]))

    return None, float('inf')  # No path found
```

Listing 1: A* Implementation with Precomputed Heuristic

2.3 Phase 3: Event-Driven Simulation

Algorithm Details

Congestion-Aware Scheduling
The simulation uses a priority queue to process taxi movement events chronologically:

Algorithm 2 Event-Driven Simulation

Initialize priority queue PQ with all taxis at time $t = 0$
Initialize edge reservations $R = \{\}$ for each edge
while PQ is not empty **do**
  $(t, taxi\_id) \leftarrow PQ.\text{pop}()$
  $taxi \leftarrow taxis[taxi\_id]$
  if taxi is at destination **then**
    Mark taxi as finished
  else
    $(u, v) \leftarrow$ current edge to traverse
    $overlaps \leftarrow \text{count\_overlaps}(R[(u, v)], t, t + \text{edge\_time})$
    if $overlaps \leq 1$ **then**
      Reserve edge: $R[(u, v)].\text{append}((t, t + \text{edge\_time}))$
      Move taxi to next node
      $PQ.\text{push}((t + \text{edge\_time}, taxi\_id))$
    else
      Add wait event: taxi waits W minutes
      $PQ.\text{push}((t + W, taxi\_id))$
    end if
  end if
end while

3 Code Implementation

3.1 Key Functions

```python
def count_overlaps(reservations, start, end):
    """Count how many existing reservations overlap with [start,
       end)"""
    count = 0
    for (a, b) in reservations:
        # Check if intervals [start, end) and [a, b) overlap
        if not (end <= a or start >= b):
            count += 1
    return count
```

Listing 2: Overlap Detection for Congestion Control

```python
def parse_input(text):
    tokens = text.strip().split()
    iterator = iter(tokens)
```

```python

    N = int(next(iterator))  # nodes
    M = int(next(iterator))  # edges
    P = int(next(iterator))  # passengers
    W = int(next(iterator))  # wait time
    S = float(next(iterator))  # speed

    # Parse coordinates
    coords = {}
    for i in range(1, N+1):
        x, y = float(next(iterator)), float(next(iterator))
        coords[i] = (x, y)

    # Parse graph edges
    graph = {i: [] for i in range(1, N+1)}
    for _ in range(M):
        u, v, d = int(next(iterator)), int(next(iterator)),
           float(next(iterator))
        graph[u].append((v, d))
        graph[v].append((u, d))  # undirected

    # Parse passenger trips
    trips = []
    for _ in range(P):
        a, b = int(next(iterator)), int(next(iterator))
        trips.append((a, b))

    return N, M, P, W, S, coords, graph, trips
```

Listing 3: Input Parsing

4 Sample Execution & Results

Results & Analysis

Input: 8 nodes, 9 edges, 3 passengers, 30min wait, 40km/h speed
Passenger Assignments:

- Taxi 1: Node 2 $\rightarrow$ Node 7
- Taxi 2: Node 1 $\rightarrow$ Node 8
- Taxi 3: Node 3 $\rightarrow$ Node 4

4.1 Detailed Execution Trace

Time per km: $\frac{60 \text{ min}}{40 \text{ km/h}} = 1.5$ minutes/km

Taxi	Route	Congestion	Total Time
1	$2 \to 3 \to 5 \to 7$	None	165.0 min
2	$1 \to 2 \to 3 \to 5 \to 7 \to 8$	Wait at edge 2→3	315.0 min
3	$3 \to 2 \to 4$	None	165.0 min

Congestion Analysis:

- Edge 2→3 has overlapping reservations from Taxi 1 [0, 75) and Taxi 3 [0, 75)
- Taxi 2 attempts to use edge 2→3 at $t = 45$ minutes
- Congestion detected! Taxi 2 waits 30 minutes at node 2
- Taxi 2 resumes at $t = 75$ minutes when edge becomes available

4.2 Performance Metrics

$$\text{Total Completion Time} = \sum_{i=1}^{P} T_i = 165 + 315 + 165 = \boxed{645.0 \text{ minutes}} \tag{1}$$

$$\text{Makespan} = \max_{i=1}^{P} T_i = \max(165, 315, 165) = \boxed{315.0 \text{ minutes}} \tag{2}$$

5 Algorithm Complexity Analysis

Algorithm Details

Time Complexity Analysis

- **Preprocessing (Dijkstra):** $O(|D| \cdot (|V| + |E|) \log |V|)$ where $|D|$ is number of unique destinations
- **A* Search:** $O(P \cdot |E| \log |V|)$ for P taxi paths
- **Simulation:** $O(P \cdot L \cdot |E|)$ where L is average path length

Space Complexity: $O(|V|^2 + |E| \cdot T)$ for storing heuristics and reservations

6 Key Algorithmic Insights

1. **Optimal Heuristic:** Using Dijkstra's algorithm to precompute exact shortest path distances provides the best possible heuristic for A*, ensuring optimal pathfinding.
2. **Event-Driven Simulation:** Processing taxi movements chronologically allows accurate modeling of congestion without complex scheduling conflicts.
3. **Greedy Scheduling:** The first-come-first-served approach for edge reservations is simple but effective for this problem size.

7 Conclusion

This solution effectively combines classical shortest path algorithms (Dijkstra, A*) with discrete event simulation to handle the multi-taxi routing problem with capacity constraints. The precomputed heuristic approach ensures optimal pathfinding while the event-driven simulation accurately models real-world congestion scenarios.