

AI END LAB EXAM

NAME: G. OMKAR

HT.NO: 2403A52039

BATCH NO: AIB03

TASK 1:

PROMPT:

Implement dynamic routing for public transit using AI-enhanced Dijkstra and A* algorithms with live fluctuating costs, then simulate and measure average travel times.

CODE:

The screenshot shows a code editor window with the file 'exam_end_1.py' open. The code implements the A* search algorithm. It includes functions for A* search, a heuristic calculation (Euclidean distance), and a LiveCost class. The code uses a priority queue (heapq) to manage nodes based on their f-score. The interface includes standard file operations like File, Edit, Selection, View, Go, Run, and a status bar at the bottom.

```
38 def a_star(graph, start, goal, heuristic):
39     queue = []
40     heapq.heappush(queue, (0, start))
41     distances = {start: 0}
42     previous = {start: None}
43     while queue:
44         current_f_score, current_node = heapq.heappop(queue)
45         if current_node == goal:
46             break
47         for neighbor, weight in graph.neighbors(current_node):
48             tentative_g_score = distances[current_node] + weight()
49             if neighbor not in distances or tentative_g_score < distances[neighbor]:
50                 distances[neighbor] = tentative_g_score
51                 f_score = tentative_g_score + heuristic(neighbor, goal)
52                 previous[neighbor] = current_node
53                 heapq.heappush(queue, (f_score, neighbor))
54     path = []
55     node = goal
56     while node:
57         path.append(node)
58         node = previous[node]
59     path.reverse()
60     return path, distances.get(goal, float('inf'))
61
62 def heuristic(node, goal):
63     x1, y1 = node
64     x2, y2 = goal
65     return ((x1 - x2)**2 + (y1 - y2)**2)**0.5
66
67 class LiveCost:
68     def __init__(self, base_cost):
69         self.base_cost = base_cost
70     def __call__(self):
71         return self.base_cost * random.uniform(0.8, 1.2)
72
73 G = Graph()
74 G.add_edge((0, 0), (1, 0), LiveCost(5))
75 G.add_edge((1, 0), (2, 0), LiveCost(10))
76 G.add_edge((0, 0), (0, 1), LiveCost(2))
77 G.add_edge((0, 1), (1, 1), LiveCost(3))
78 G.add_edge((1, 1), (2, 0), LiveCost(2))
79 G.add_edge((2, 0), (3, 0), LiveCost(1))
80
81 def simulate_routing(algorithm, graph, start, goal, trials=100):
82     total_time = 0
83     total_cost = 0
84     for _ in range(trials):
85         start_time = time.time()
86         if algorithm == dijkstra:
87             path, cost = algorithm(graph, start, goal)
88         else:
89             path, cost = algorithm(graph, start, goal, heuristic)
90         end_time = time.time()
91         total_time += (end_time - start_time)
92         total_cost += cost
93     return total_time / trials, total_cost / trials
94
95 start_node = (0, 0)
96 goal_node = (3, 0)
97 avg_time_dijkstra, avg_cost_dijkstra = simulate_routing(dijkstra, G, start_node, goal_node)
98 avg_time_astar, avg_cost_astar = simulate_routing(a_star, G, start_node, goal_node)
99
100 print(f'Dijkstra avg time: {avg_time_dijkstra:.6f}s, avg cost: {avg_cost_dijkstra:.2f}')
101 print(f'A* avg time: {avg_time_astar:.6f}s, avg cost: {avg_cost_astar:.2f}')
102
```

The screenshot shows a code editor window with the file 'exam_end_1.py' open. The code defines a 'LiveCost' class that multiplies a base cost by a random factor between 0.8 and 1.2. It also contains a 'simulate_routing' function that runs a specified algorithm (either Dijkstra's or A*) multiple times to calculate average time and cost. The code uses the 'time' module for timing. The interface includes standard file operations like File, Edit, Selection, View, Go, Run, and a status bar at the bottom.

```
71         return self.base_cost * random.uniform(0.8, 1.2)
72
73 G = Graph()
74 G.add_edge((0, 0), (1, 0), LiveCost(5))
75 G.add_edge((1, 0), (2, 0), LiveCost(10))
76 G.add_edge((0, 0), (0, 1), LiveCost(2))
77 G.add_edge((0, 1), (1, 1), LiveCost(3))
78 G.add_edge((1, 1), (2, 0), LiveCost(2))
79 G.add_edge((2, 0), (3, 0), LiveCost(1))
80
81 def simulate_routing(algorithm, graph, start, goal, trials=100):
82     total_time = 0
83     total_cost = 0
84     for _ in range(trials):
85         start_time = time.time()
86         if algorithm == dijkstra:
87             path, cost = algorithm(graph, start, goal)
88         else:
89             path, cost = algorithm(graph, start, goal, heuristic)
90         end_time = time.time()
91         total_time += (end_time - start_time)
92         total_cost += cost
93     return total_time / trials, total_cost / trials
94
95 start_node = (0, 0)
96 goal_node = (3, 0)
97 avg_time_dijkstra, avg_cost_dijkstra = simulate_routing(dijkstra, G, start_node, goal_node)
98 avg_time_astar, avg_cost_astar = simulate_routing(a_star, G, start_node, goal_node)
99
100 print(f'Dijkstra avg time: {avg_time_dijkstra:.6f}s, avg cost: {avg_cost_dijkstra:.2f}')
101 print(f'A* avg time: {avg_time_astar:.6f}s, avg cost: {avg_cost_astar:.2f}')
102
```

OUTPUT:



The screenshot shows a Windows terminal window within the Visual Studio Code interface. The terminal output is as follows:

```
Microsoft Windows [Version 10.0.26200.7171]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nalla\OneDrive\Desktop\AI assisted coding>C:\Users\nalla\anaconda3\Scripts\activate

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>conda activate base

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding> cmd /C "c:\Users\nalla\anaconda3\python.exe c:\Users\nalla\vscode\extensions\ms-python.debugpy-2025.16.0-win32-x64\bundled\libs\debugpy\launcher 51793 -- "c:\Users\nalla\OneDrive\Desktop\AI assisted coding\exam end 1.py"
Dijkstra avg time: 0.000026s, avg cost: 7.94
A* avg time: 0.000018s, avg cost: 8.05

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>
```

OBSERVATION:

Dijkstra's and A* algorithms to quickly find the cheapest path between two places while adapting to changing conditions. It repeats these searches many times, using live, fluctuating costs to imitate real traffic or delays, and prints out how quickly and cheaply each method guides you through the network, just like a navigation app does in real life

TASK 2:

PROMPT:

Optimize traffic light timings using an AI-driven heuristic loop to maximize traffic throughput, then simulate and evaluate the results.

CODE:

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, ...
- Search Bar:** All assisted coding
- Toolbar:** Includes icons for file operations, search, and help.
- Code Cell:** The code is written in Python 3. It defines a function `optimize_traffic_lights` which uses simulated annealing to find the best traffic light timings. It also includes a helper function `simulate_throughput` to evaluate the throughput based on the given traffic light timings.

```
1 import random
2 import numpy as np
3
4 # AI-driven heuristic optimization loop (simulated annealing style)
5 def optimize_traffic_lights(initial_timings, evaluate_func, iterations=1000, temp=100, cooling=0.95):
6     current_timings = initial_timings[:]
7     current_score = evaluate_func(current_timings)
8     best_timings = current_timings[:]
9     best_score = current_score
10
11     for i in range(iterations):
12         new_timings = current_timings[:]
13         idx = i % len(new_timings)
14         # Slightly adjust one traffic light timing (+1 or -1), with minimum 5 seconds
15         new_timings[idx] = max(5, new_timings[idx] + (1 if i % 2 == 0 else -1))
16         new_score = evaluate_func(new_timings)
17
18         # Accept new timings if improvement or with probability related to temperature
19         if new_score > current_score or random.random() < temp / 100:
20             current_timings = new_timings
21             current_score = new_score
22             if new_score > best_score:
23                 best_timings = new_timings
24                 best_score = new_score
25
26             temp *= cooling
27     return best_timings, best_score
28
29 # Simulation of traffic throughput as evaluation function
30 def simulate_throughput(timings):
31     # Throughput is sum of green light durations minus penalty for uneven timings (variance)
32     throughput = sum(timings) - np.var(timings)
33     return throughput
34
35 # Initial traffic light timings (seconds) for 4 phases
```

OUTPUT:



The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** On the left, showing icons for Problems, Output, Debug Console, Terminal (highlighted), File, Selection, View, Go, Run, and others.
- Terminal:** The central area displays the following text:

```
Microsoft Windows [Version 10.0.26200.7171]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nalla\OneDrive\Desktop\AI assisted coding>C:/Users/nalla/anaconda3/Scripts/activate

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>conda activate base

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding> cmd /C "c:\Users\nalla\anaconda3\python.exe c:\Users\nalla\vscode\extensions\ms-python.debugpy-2025.16.0-win32-x64\bundled\libs\debugpy\launcher 57669 -- "c:\Users\nalla\OneDrive\Desktop\AI assisted coding\lab end exam 2.py"
Optimized traffic light timings (seconds): [31, 30, 30, 30]
Estimated throughput score: 120.8125

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>
```
- Status Bar:** At the bottom right, it says "Python Debug Console".

OBSERVATION:

The AI heuristic gently adjusts traffic light durations to balance green light times, improving overall traffic flow. Simulation shows that small timing tweaks can significantly increase throughput by reducing stop-and-go inefficiencies. This approach adapts dynamically, making intersections more efficient without complex modelling.