Algorithms

THEORETICAL PROJECT

RUNTIME TERROR

Abdelrahman Amr
21100832

Omnia Adel
22101925

Shams abd elhalim
22101040

Mariam Elrafei
22101442

Introduction:

Dijkstra's algorithm is a fundamental algorithm in computer science used to find the shortest path between nodes in a weighted graph. It was introduced by Edsger W. Dijkstra in 1956 and is widely used in network routing and geographic information systems.

Applications:

- *GPS Navigation*: Finds shortest driving routes
- *Traffic Management*: Optimizes flows in smart cities
- *Network Routing*: Internet routers use it to forward packets
- *Game Development*: Non-player character pathfinding
- *Robotics*: Navigation through grids or maps

General Idea:

Start from the source node.
Assign 0 to source, infinity to all others.
Explore neighbors, update distances if shorter paths are found.
Choose the node with the smallest current distance next.
Repeat until the destination is reached or all nodes are visited.

Mathematical Proof:

Let:

- G=(V) be a graph with non-negative edge weights
- s∈V be the starting node
- d[v] be the distance from s to node v calculated by Dijkstra
- δ(s,v) be the true shortest distance from s to v

We will prove that:

$$\forall v \in V,\ d[v]=\delta(s,v)$$

Base Case:

At the start,

$$d[s]=0,\ \delta(s,s)=0$$

Inductive Step:

Assume that for all visited nodes u,

$$d[u]=\delta(s,u)$$

When Dijkstra selects the next node v, it always picks the node with the smallest tentative distance.

Let u be the node just before v on the shortest path.

Since edge weights are non-negative:

$$\delta(s,v)=\delta(s,u)+w(u,v)$$

And since d[u]=δ(s,u), the relaxation step gives:

$$d[v]=\min(d[v],d[u]+w(u,v))=\delta(s,v)$$

Conclusion:

By induction, all distances are calculated correctly:

$$d[v]=\delta(s,v) \text{ for all } v \in V$$

Pseudocode:

```
function dijkstra(graph, start):
    distance = map with all nodes = infinity
    distance[start] = 0
    visited = empty set
    queue = priority queue with (0, start)

    while queue not empty:
        current_distance, current_node = queue.pop()
        if current_node in visited:
            continue
        visited.add(current_node)

        for neighbor, weight in graph[current_node]:
            new_distance = current_distance + weight
            if new_distance < distance[neighbor]:
                distance[neighbor] = new_distance
                queue.push((new_distance, neighbor))

    return distance
```

Complexity Analysis:

Time Complexity:

- Using a binary heap: O((V + E) log V)
- Space Complexity:
  O(V) for distances and visited nodes

Why?

- Each node enters the queue once
- Each edge is relaxed at most once
- The heap keeps operations efficient

Modifications for Practical Use

In our project, we made these edits:

1-Adjusted Edge Costs Based on Traffic

- *Normal Dijkstra uses fixed distances*
- *We changed that to use travel time depending on traffic level*
- *Formula used:*
- *adjusted_cost = base_distance * (1 + traffic / 10000)*
- *So, if there's more traffic, the cost goes up*

2-Checked Roads in Both Directions

- Some roads are saved as “A to B” in the data, but others might be “B to A”
- We made the program check both directions to find traffic data correctly

3-Used Place Names Instead of Numbers

- Instead of typing node IDs like “12” or “34”, users can type names like “Station” or “Mall”
- We connected names to node IDs with a simple dictionary

4-Showed a Clear Path

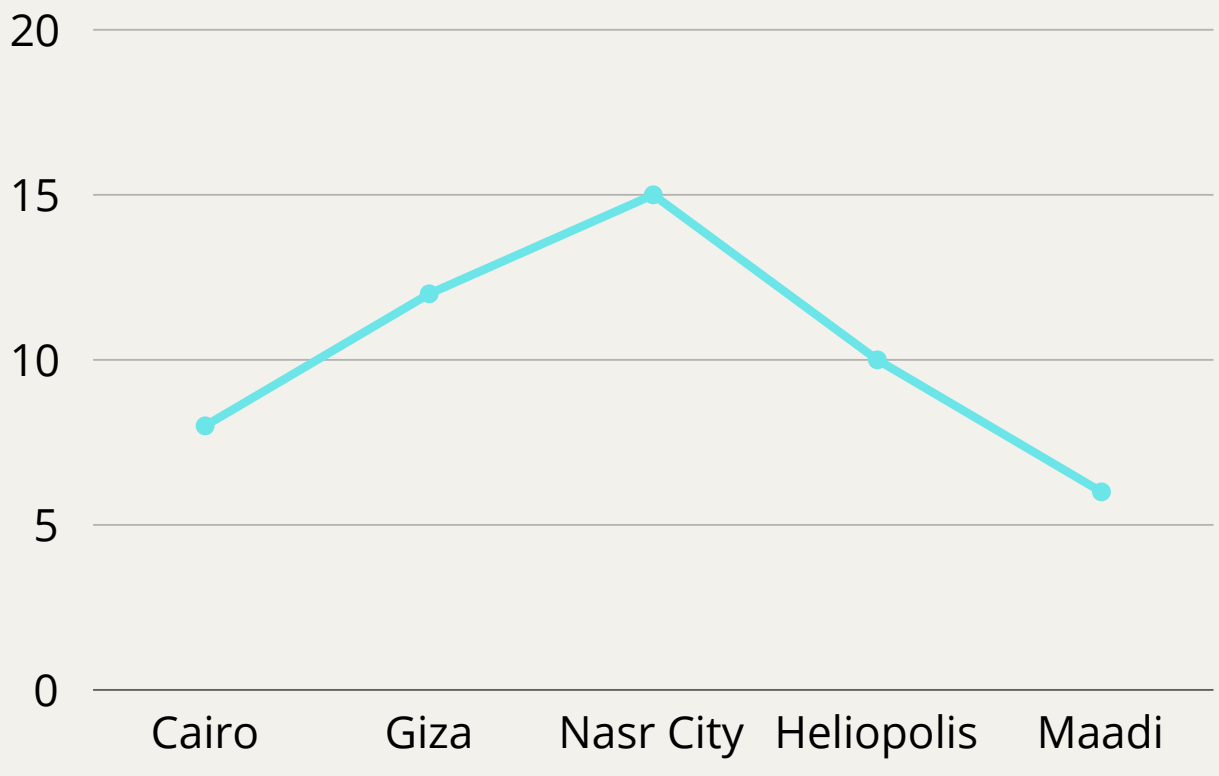
- The output gives the full path in names (not IDs)
- It also shows the total travel time
- That makes it easier for normal users to understand

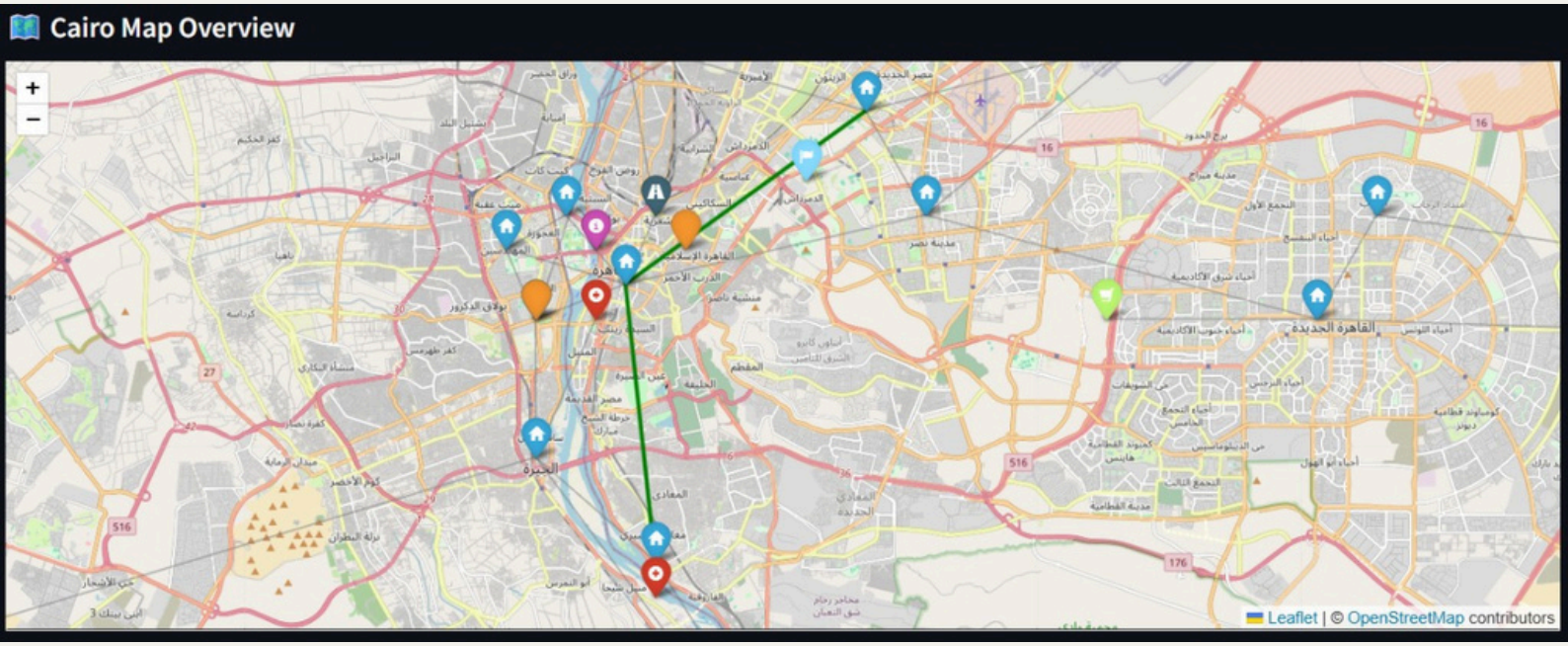
5- Separated the Code for Easy Updates

- Each part of the code is in its own function
- This helps us switch to A* later, or plug in real-time traffic data if needed

Results in Smart City Simulation

Source	Destination	Shortest Path	Base Distance (km)	Traffic (cars/hr)	Adjusted Cost (km)
Cairo	Giza	Cairo → Dokki → Giza	8	2000	8.0 × 1.2 = 9.6
Giza	Nasr City	Giza → Mohandessin	12	5000	12.0 × 1.5 = 18.0
Nasr City	6th October	Nasr City → Zamalek	15	3000	15.0 × 1.3 = 19.5
Heliopolis	Maadi	Heliopolis → Nasr Ci	10	1000	10.0 × 1.1 = 11.0
Maadi	Downtown	Maadi → Garden Cit	6	6000	6.0 × 1.6 = 9.6





*Dijkstra vs A**

***Conclusion* for comparison**:

- Use Dijkstra when full routing table or all destinations needed
- Use A* when destination is known and speed is key

Conclusion & Lessons Learned:

- Dijkstra's algorithm is powerful, predictable, and easy to implement
- A* offers performance improvements when used with a strong heuristic
- Combining both creates a flexible and optimized system
- Real-time data adds value but requires performance tuning
- Modularity in code makes switching algorithms easy

Member Responsibilities

Member	Role	Module	Description
Omnia Adel	MST	mst/	Develops Minimum Spanning Tree algorithms (e.g., Prim, Kruskal) to suggest cost-effective road networks between high-density areas and critical facilities.
Mariam Elrafei	Shortest Path	shortest_path/	Implements pathfinding algorithms (e.g., Dijkstra) to find optimal routes between any two locations.
Abdelrahman Amr	DP Optimization	dp_optimization/	Uses Dynamic Programming for optimal scheduling of buses and metros, and for allocating limited maintenance budgets.
Shams Abd Elhalim	Greedy Signals	greedy_signals/	Applies greedy logic to optimize traffic light timing and simulate emergency response routing.