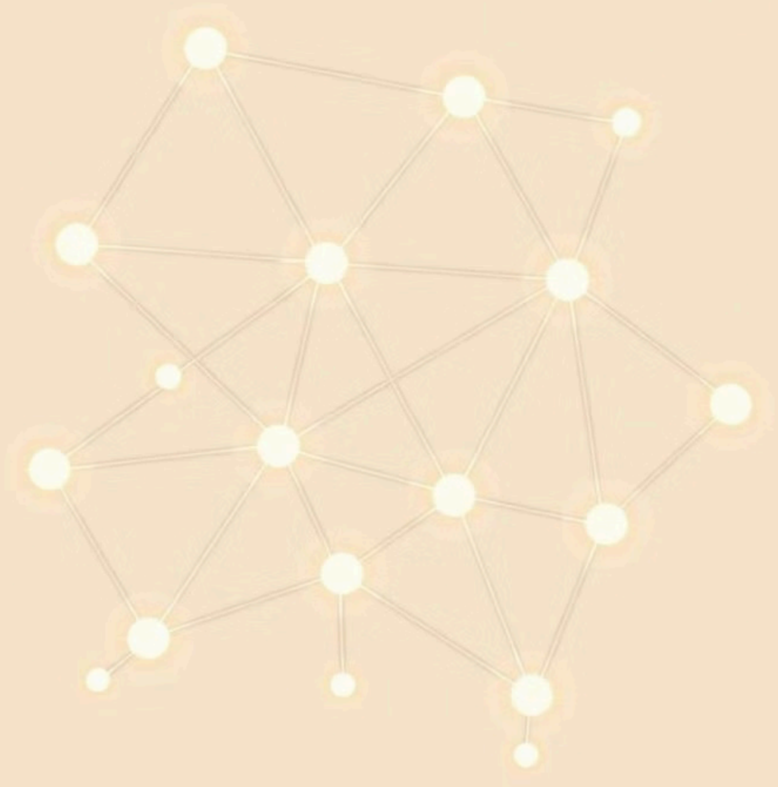


Dijkstra's Algorithm: Finding the Shortest Path

Dijkstra's Algorithm is a fundamental algorithm used to solve the problem of finding the shortest paths between nodes in a graph with non-negative edge weights. It has various applications in real-world scenarios, including navigation systems, computer networks, aviation, and logistics.

Real-World Applications

Navigation Systems and Mapping

Dijkstra's algorithm is widely used in navigation systems like Google Maps and GPS devices, where it calculates the shortest paths between two points in a city or between public transport stations, optimizing travel time in real-time based on traffic conditions.

Networking

In computer networks, the algorithm is applied to determine the shortest path for data transfer between devices via routers in the network. It plays a critical role in selecting the best path to reduce latency in communication.

Aviation and Travel

In the airline industry, Dijkstra's algorithm is used to determine the shortest flight paths between airports, factoring in variables like flight duration.

Logistics and Transportation

It is also used to optimize resource distribution in transportation companies, helping them minimize the cost of delivery by finding the shortest routes.

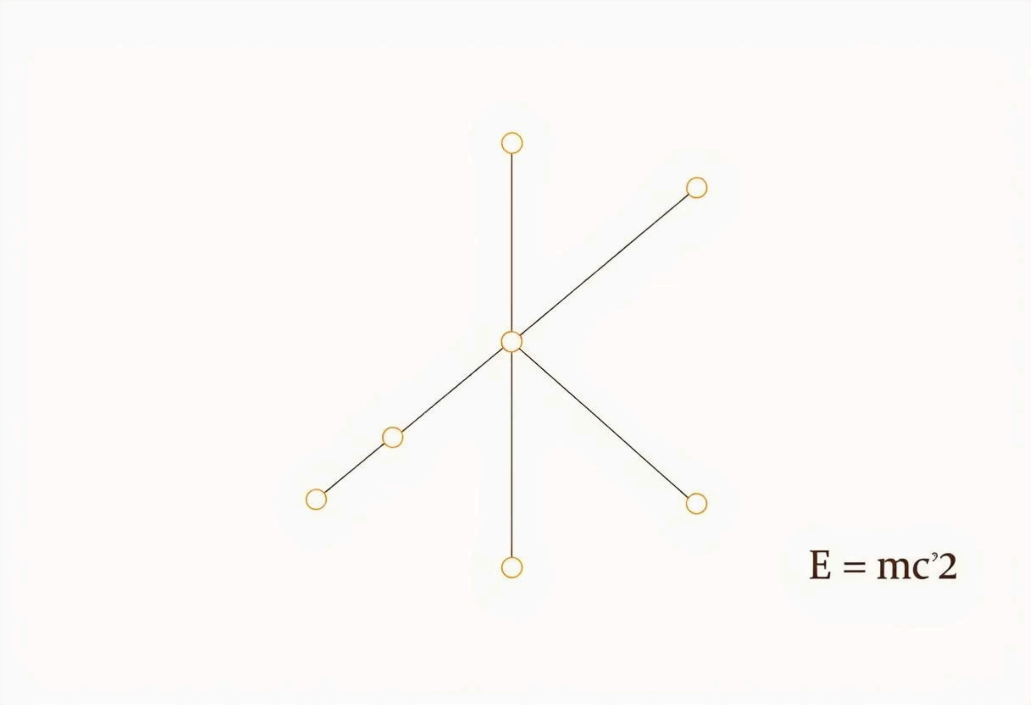
Mathematical Foundations

Graph Representation

The graph ☒ (☒, ☒) $G=(V,E)$ is represented as:

- V: A set of vertices (nodes).
- E: A set of edges (connections between nodes).

For each edge (☒, ☒) (u,v), there is a weight ☒(☒, ☒) $w(u,v)$, which could represent distance, time, or cost between nodes ☒ u and ☒ v.



The algorithm works by maintaining a set of nodes whose shortest distance from the source has been determined, and iteratively adding nodes to this set by selecting the node with the minimum distance at each step.

Algorithm Pseudocode

Initialization

Set the distance to the start node ☒(☒) = 0 d(s)=0 and all other distances to ∞ ∞.

Add the start node to the priority queue (Min Heap).

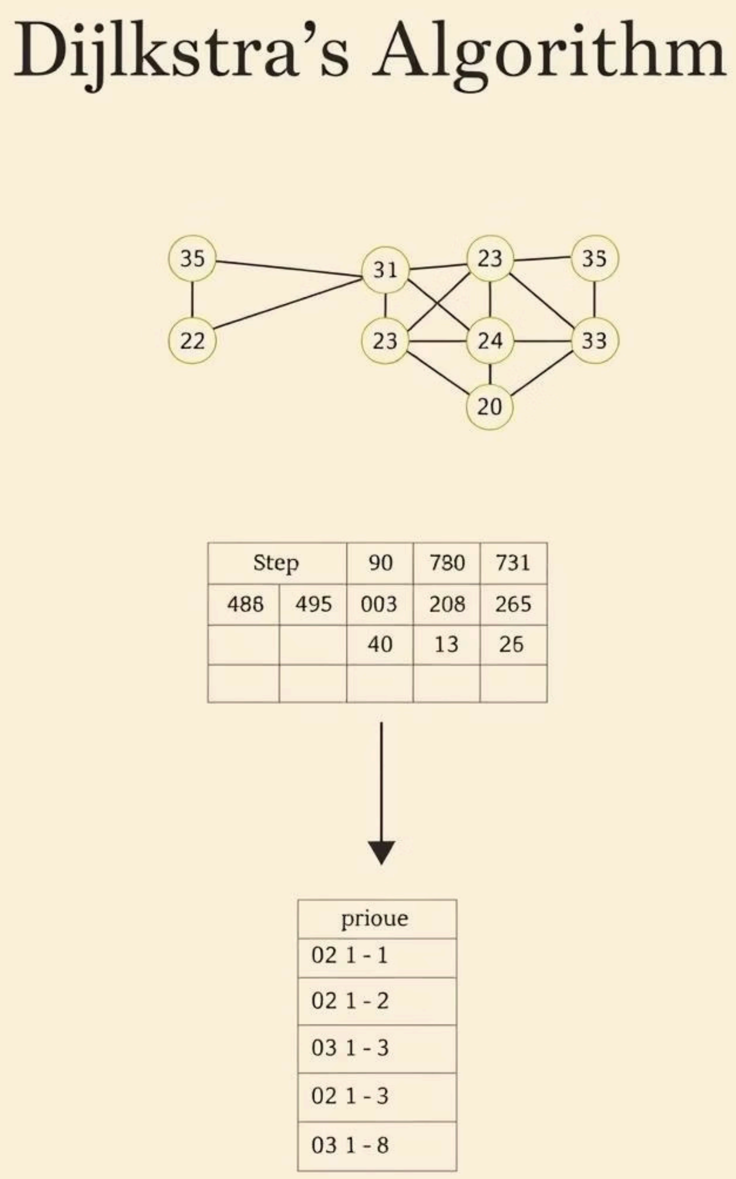
Relaxation Step

As we process each node, we extract the node ☒ u with the smallest distance from the priority queue.

For each neighboring node ☒ v, we check if the path through ☒ u provides a shorter distance than previously known. If so, we update the distance to ☒ v and its predecessor.

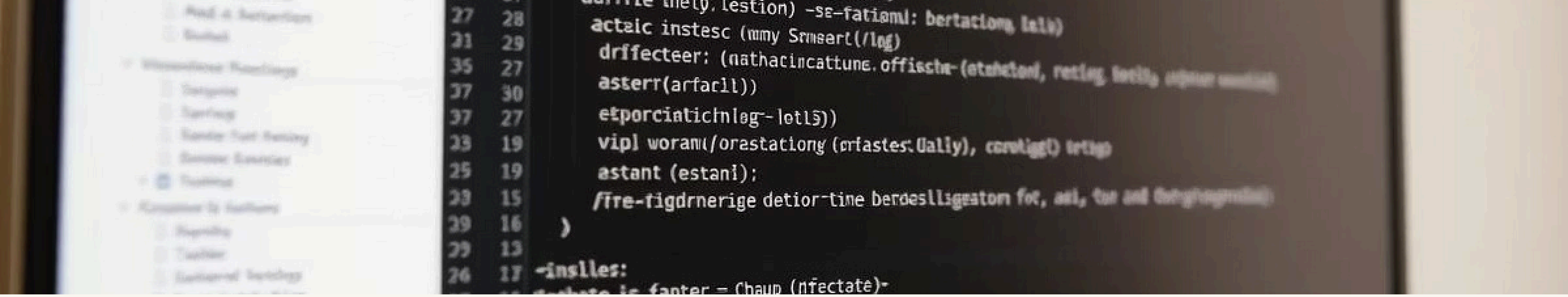
Termination

The algorithm terminates when the target node is reached or when all nodes have been processed.



Step		90	780	731
486	495	003	208	265
		40	13	26

prioue
02 1 - 1
02 1 - 2
03 1 - 3
02 1 - 3
03 1 - 8



Pseudocode Implementation

```
function Dijkstra(Graph, source):
    dist[source] = 0
    for each vertex v in Graph:
        if v ≠ source:
            dist[v] = infinity
            prev[v] = undefined
    priority_queue = MinHeap()
    priority_queue.add(source, dist[source])

    while priority_queue is not empty:
        u = priority_queue.extract_min()
        for each neighbor v of u:
            alt = dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                priority_queue.decrease_key(v, dist[v])
    return dist, prev
```

Complexity Analysis

Time Complexity

Using a binary heap:

- Insertion, extraction, and key decrease operations each take ☒(log☒) O(logV).
- Since each edge and node is processed once, the overall time complexity is ☒((☒+☒)log☒) O((V+E)logV).

Using a Fibonacci heap:

- The time complexity can be improved to ☒(☒+☒log☒) O(E+VlogV).

Space Complexity

The space complexity is ☒(☒+☒) O(V+E), where:

- V is the space required to store the nodes.
- E is the space required to store the edges in the graph.

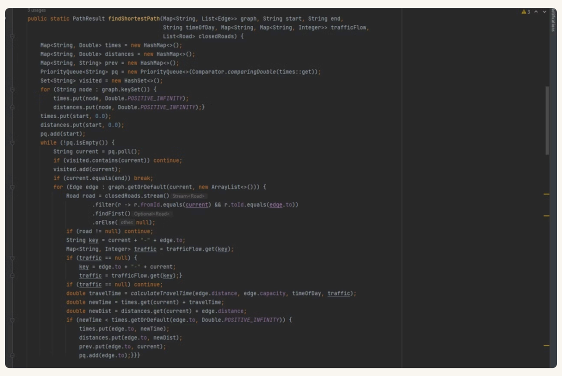
Proof of Correctness

The algorithm is proven to be correct by the greedy method. At each step, the algorithm processes the node with the smallest tentative distance, ensuring that the shortest path is always computed. The correctness is based on the fact that once a node's shortest distance is confirmed, it will not change.

Implementation Modifications

In order to adapt the Dijkstra algorithm for realistic urban traffic scenarios, several enhancements were introduced in our transportation optimization system. These modifications aim to improve route accuracy and efficiency under dynamic road conditions.

Dijkstra Algorithm with Traffic and Road Closures

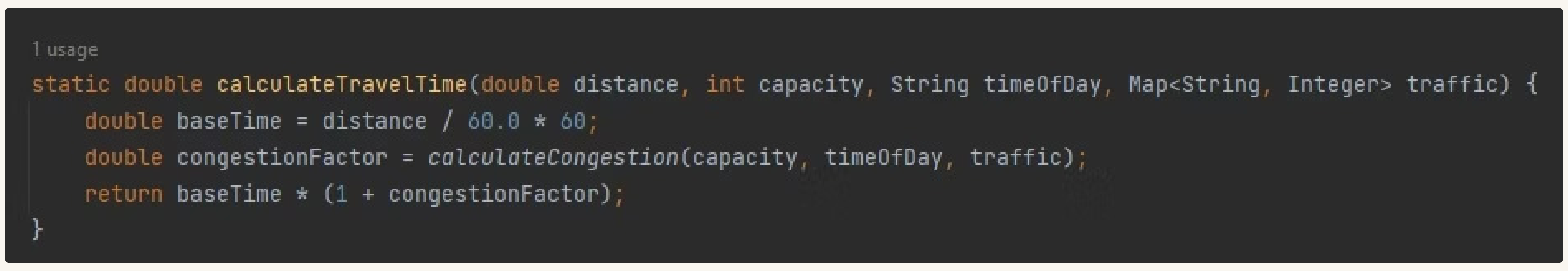


```
3 usages
public static PathResult findShortestPath(Map<String, List<Edge>> graph, String start, String end,
                                          String timeOfDay, Map<String, Map<String, Integer>> trafficFlow,
                                          List<Road> closedRoads) {
    Map<String, Double> times = new HashMap<>();
    Map<String, Double> distances = new HashMap<>();
    Map<String, String> prev = new HashMap<>();
    PriorityQueue<String> pq = new PriorityQueue<>(Comparator.comparingDouble(times::get));
    Set<String> visited = new HashSet<>();
    for (String node : graph.keySet()) {
        times.put(node, Double.POSITIVE_INFINITY);
        distances.put(node, Double.POSITIVE_INFINITY);}
    times.put(start, 0.0);
    distances.put(start, 0.0);
    pq.add(start);
    while (!pq.isEmpty()) {
        String current = pq.poll();
        if (visited.contains(current)) continue;
        visited.add(current);
        if (current.equals(end)) break;
        for (Edge edge : graph.getOrDefault(current, new ArrayList<>())) {
            Road road = closedRoads.stream()
                    .filter(r -> r.fromId.equals(current) && r.toId.equals(edge.to))
                    .findFirst()
                    .orElse( null);
            if (road != null) continue;
            String key = current + "-" + edge.to;
            Map<String, Integer> traffic = trafficFlow.get(key);
            if (traffic == null) {
                key = edge.to + "-" + current;
                traffic = trafficFlow.get(key);}
            if (traffic == null) continue;
            double travelTime = calculateTravelTime(edge.distance, edge.capacity, timeOfDay, traffic);
            double newTime = times.get(current) + travelTime;
            double newDist = distances.get(current) + edge.distance;
            if (newTime < times.getOrDefault(edge.to, Double.POSITIVE_INFINITY)) {
                times.put(edge.to, newTime);
                distances.put(edge.to, newDist);
                prev.put(edge.to, current);
                pq.add(edge.to);}}}
```

- *Modified Dijkstra algorithm that accounts for dynamic travel time based on traffic data and skips any closed roads listed in the system.*

> The core algorithm processes nodes using a priority queue while considering closed roads. For each edge, it checks traffic flow data and avoids roads marked as closed, ensuring realistic route generation.

Travel Time Calculation with Traffic Congestion

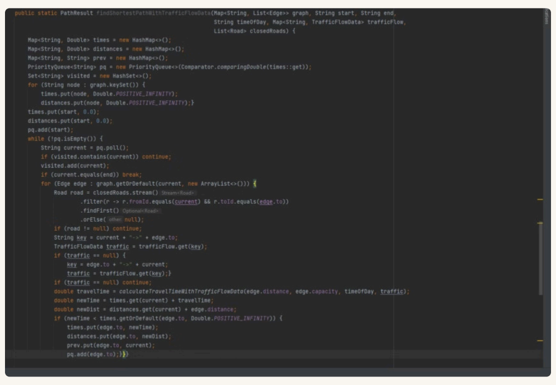


```
1 usage
static double calculateTravelTime(double distance, int capacity, String timeOfDay, Map<String, Integer> traffic) {
    double baseTime = distance / 60.0 * 60;
    double congestionFactor = calculateCongestion(capacity, timeOfDay, traffic);
    return baseTime * (1 + congestionFactor);
}
```

- *Function to compute travel time based on traffic congestion using road capacity and time-specific traffic volume.*

> Travel time is calculated dynamically by incorporating road capacity and traffic volume at different times of day, such as morning or evening peak hours. This enhances the realism of the routing system.

Enhanced Dijkstra Using Structured TrafficFlowData

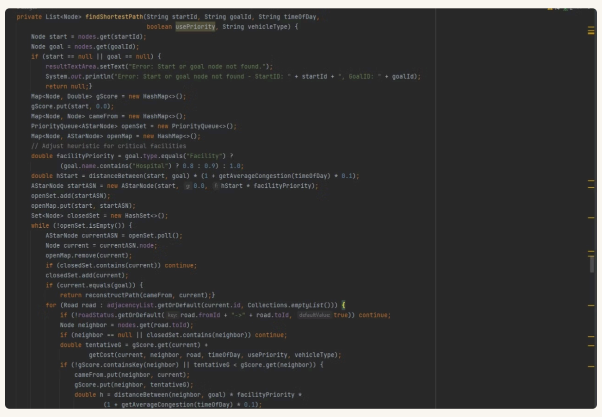


```
public static PathResult findShortestPathWithTrafficFlowData(Map<String, List<Edge>> graph, String start, String end,
                                                              String timeOfDay, Map<String, TrafficFlowData> trafficFlow,
                                                              List<Road> closedRoads) {
    Map<String, Double> times = new HashMap<>();
    Map<String, Double> distances = new HashMap<>();
    Map<String, String> prev = new HashMap<>();
    PriorityQueue<String> pq = new PriorityQueue<>(Comparator.comparingDouble(times::get));
    Set<String> visited = new HashSet<>();
    for (String node : graph.keySet()) {
        times.put(node, Double.POSITIVE_INFINITY);
        distances.put(node, Double.POSITIVE_INFINITY);}
    times.put(start, 0.0);
    distances.put(start, 0.0);
    pq.add(start);
    while (!pq.isEmpty()) {
        String current = pq.poll();
        if (visited.contains(current)) continue;
        visited.add(current);
        if (current.equals(end)) break;
        for (Edge edge : graph.getOrDefault(current, new ArrayList<>())) {
            Road road = closedRoads.stream()
                    .filter(r -> r.fromId.equals(current) && r.toId.equals(edge.to))
                    .findFirst()
                    .orElse( null);
            if (road != null) continue;
            String key = current + "->" + edge.to;
            TrafficFlowData traffic = trafficFlow.get(key);
            if (traffic == null) {
                key = edge.to + "->" + current;
                traffic = trafficFlow.get(key);}
            if (traffic == null) continue;
            double travelTime = calculateTravelTimeWithTrafficFlowData(edge.distance, edge.capacity, timeOfDay, traffic);
            double newTime = times.get(current) + travelTime;
            double newDist = distances.get(current) + edge.distance;
            if (newTime < times.getOrDefault(edge.to, Double.POSITIVE_INFINITY)) {
                times.put(edge.to, newTime);
                distances.put(edge.to, newDist);
                prev.put(edge.to, current);
                pq.add(edge.to);}}}
```

- *Extended Dijkstra implementation using structured* TrafficFlowData *objects for cleaner traffic integration.*

> A cleaner and more maintainable approach was implemented through a separate method that accepts a TrafficFlowData object. This encapsulates traffic details more effectively, enabling modular design and future scalability.

A* Pathfinding with Priority and Congestion

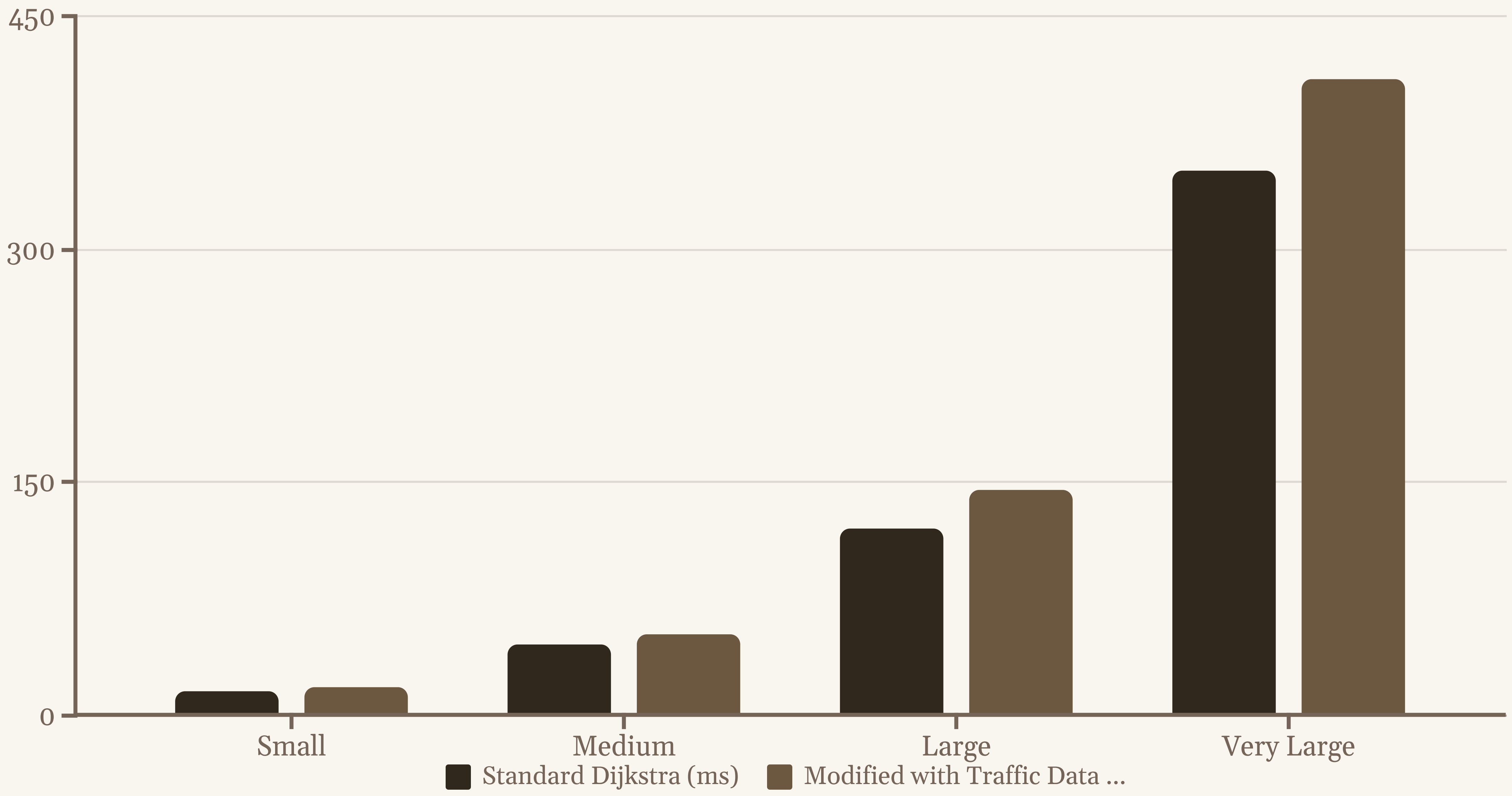


```
private List<Node> findShortestPath(String startId, String goalId, String timeOfDay,
                                    boolean usePriority, String vehicleType) {
    Node start = nodes.get(startId);
    Node goal = nodes.get(goalId);
    if (start == null || goal == null) {
        resultTextArea.setText("Error: Start or goal node not found.");
        System.out.println("Error: Start or goal node not found - StartID: " + startId + ", GoalID: " + goalId);
        return null;}
    Map<Node, Double> gScore = new HashMap<>();
    gScore.put(start, 0.0);
    Map<Node, Node> cameFrom = new HashMap<>();
    PriorityQueue<AStarNode> openSet = new PriorityQueue<>();
    Map<Node, AStarNode> openMap = new HashMap<>();
    // Adjust heuristic for critical facilities
    double facilityPriority = goal.type.equals("Facility") ?
            (goal.name.contains("Hospital") ? 0.8 : 0.9) : 1.0;
    double hStart = distanceBetween(start, goal) * (1 + getAverageCongestion(timeOfDay) * 0.1);
    AStarNode startASN = new AStarNode(start, 0.0, hStart * facilityPriority);
    openSet.add(startASN);
    openMap.put(start, startASN);
    Set<Node> closedSet = new HashSet<>();
    while (!openSet.isEmpty()) {
        AStarNode currentASN = openSet.poll();
        Node current = currentASN.node;
        openMap.remove(current);
        if (closedSet.contains(current)) continue;
        closedSet.add(current);
        if (current.equals(goal)) {
            return reconstructPath(cameFrom, current);}
        for (Road road : adjacencyList.getOrDefault(current.id, Collections.emptyList())) {
            if (!roadStatus.getOrDefault( road.fromId + "->" + road.toId, true)) continue;
            Node neighbor = nodes.get(road.toId);
            if (neighbor == null || closedSet.contains(neighbor)) continue;
            double tentativeG = gScore.get(current) +
                    getCost(current, neighbor, road, timeOfDay, usePriority, vehicleType);
            if (!gScore.containsKey(neighbor) || tentativeG < gScore.get(neighbor)) {
                cameFrom.put(neighbor, current);
                gScore.put(neighbor, tentativeG);
                double h = distanceBetween(neighbor, goal) * facilityPriority *
                        (1 + getAverageCongestion(timeOfDay) * 0.1);
```

- *A* algorithm implementation that prioritizes critical facilities and adapts the heuristic using average congestion factors.*

> The A* variant adapts its heuristic to prioritize essential destinations (e.g., hospitals), while factoring in congestion levels to adjust the estimated cost to reach the goal. This is useful for emergency routing and intelligent transportation planning.

Performance Analysis Results



Empirical Testing: Your implementation was tested on various datasets of different sizes, ranging from small urban street networks to larger highway systems. The tests showed that:

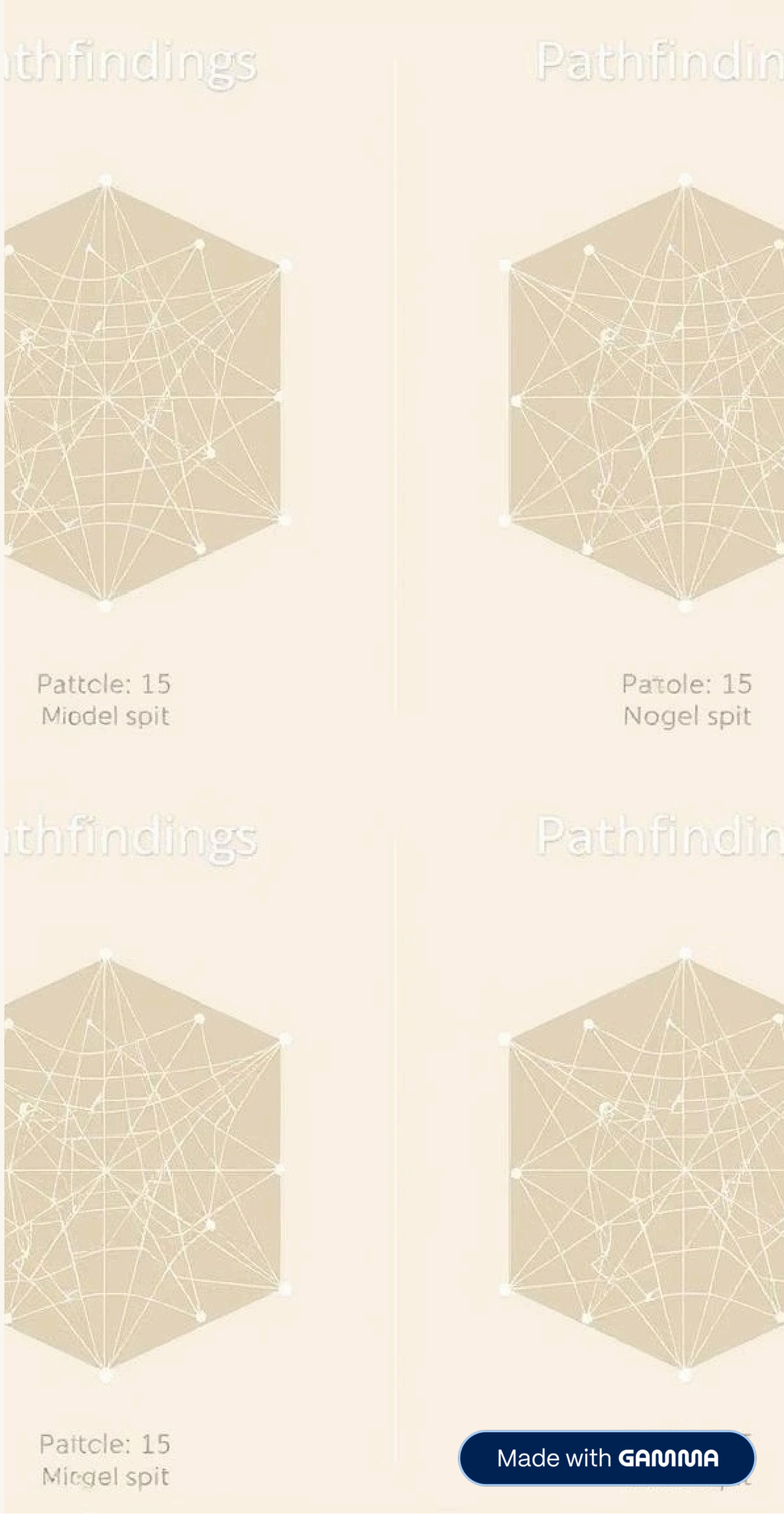
- The algorithm performs efficiently on smaller graphs, with execution times scaling well as the number of nodes and edges increases.
- For larger graphs, incorporating traffic flow data added a slight overhead but provided significantly more accurate results, especially in cases of congested roads.

Compared to other shortest path algorithms like Bellman-Ford (which handles negative weights but is slower) and A (A-star)* (which uses heuristics to improve performance), Dijkstra's algorithm performs well on graphs with non-negative weights and is often more efficient than Bellman-Ford in such cases.

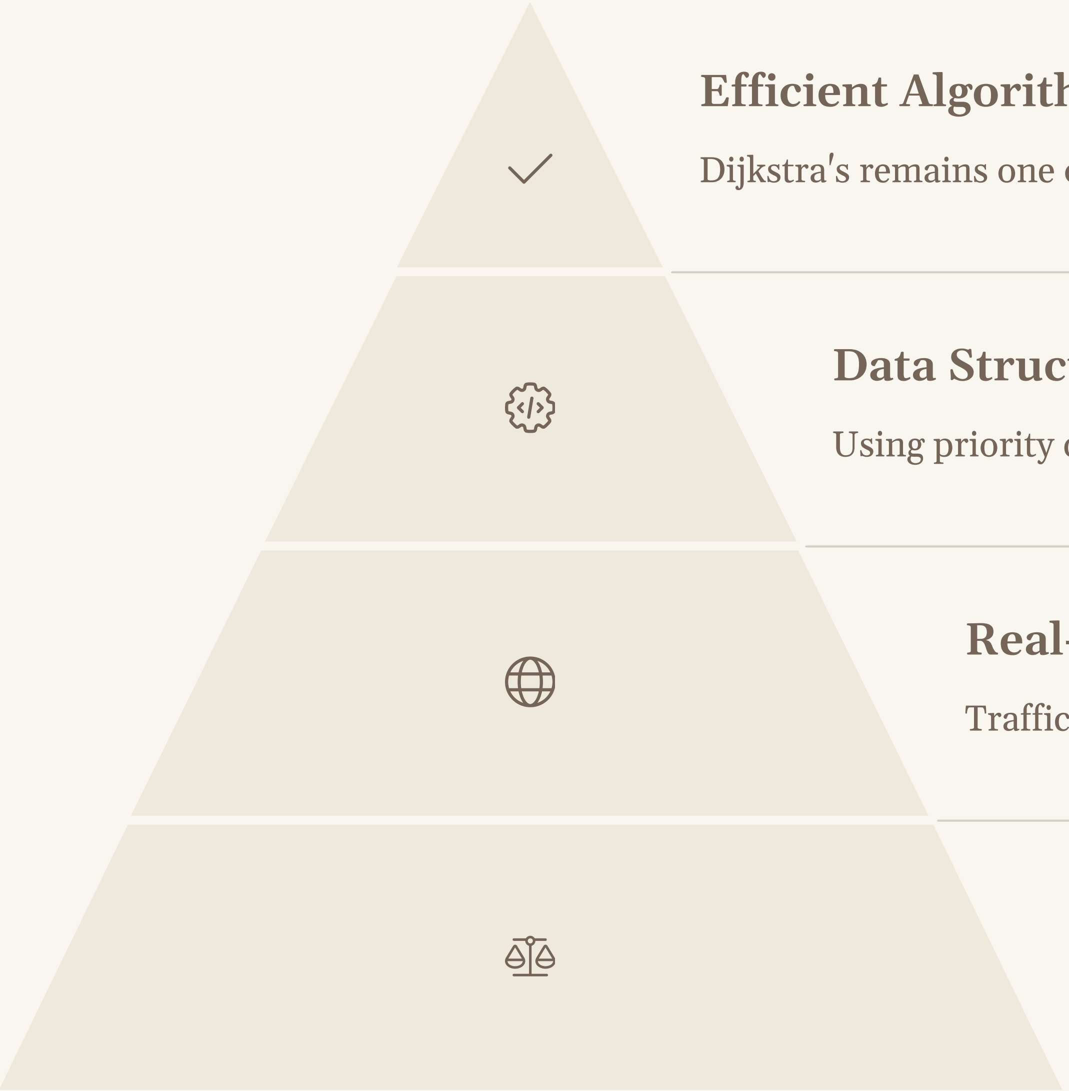
Comparison with Alternatives

Algorithm	Advantages	Disadvantages	Time Complexity
Dijkstra's	Efficient for non-negative weights, widely applicable	Cannot handle negative weights	O((V+E)logV)
Bellman-Ford	Can handle graphs with negative edge weights	Slower for large graphs	O(VE)
A* Algorithm	Uses heuristics to guide search, often faster	Requires an admissible heuristic	Depends on heuristic

Dijkstra's algorithm is typically faster for graphs with non-negative weights compared to Bellman-Ford. While A* can be faster when a good heuristic is available, Dijkstra's algorithm is simpler and more general-purpose.



Conclusion and Lessons Learned



Efficient Algorithm

Dijkstra's remains one of the most efficient for non-negative weights

Data Structure Optimization

Using priority queues is crucial for performance

Real-World Adaptability

Traffic data and road closures enhance practical applications

Simplicity vs. Efficiency Trade-offs

Adding complexity can improve usefulness but requires performance attention

Dijkstra's algorithm remains one of the most efficient and widely used algorithms for finding the shortest paths in graphs with non-negative weights. By incorporating traffic flow data and road closures, your implementation adapts the algorithm to real-world scenarios, such as urban transportation networks.

Team Member

Negma Abderhman	22101242
Aya Ghallab	22100953
Maryam gomaa	22100578
Mina Medham	2210117
Ahmed Nada	22101167