INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD)
© 2026 IJNRD | Volume 11, Issue 5, May 2026 | ISSN: 2456-4184 | IJNRD.ORG

SUMO-Based Simulation Study on Dynamic Routing for Ambulance

Keertan BJ, Abhishek Gowda M, Ayush Y, Kushal M S
Department of Information Science and Engineering
P.E.S College of Engineering
Mandya, Karnataka, India

Prof. Nandish J, Dr. Bramesh S M
Department of Information Science and Engineering
P.E.S College of Engineering
Mandya, Karnataka, India

***Abstract*—When a major earthquake, flood, or industrial disaster strikes a city, the roads that ambulances depend on can collapse, flood, or be blocked within minutes. Conventional GPS navigation systems are designed for normal traffic conditions and simply cannot cope when three roads are simultaneously blocked and a thousand people need help at once.**

This paper asks a very practical question: *among the most well-known pathfinding algorithms used in computer science — A* Search, Dijkstra's Algorithm, Bellman-Ford, and Breadth-First Search — which one performs best when the road network is actively breaking down around a moving ambulance?*

To answer this question, we designed a controlled simulation using SUMO (Simulation of Urban MObility), an open-source traffic simulator, and a real map of Bengaluru, India. We built a virtual city in which we could deliberately block roads mid-route and precisely measure how quickly and how accurately each algorithm re-routed the ambulance around the obstruction.

Our results clearly show that A* Search, guided by a geographic heuristic, finds the best alternate routes in an average of just 2.4 milliseconds — roughly seven times faster than the classic Dijkstra algorithm and nearly thirty times faster than the older Bellman-Ford algorithm. More importantly, A* consistently selects the *shortest time path*, not just any route around a blockage.

These findings give us a concrete, evidence-based answer for engineers building real emergency dispatch systems: A* should be the routing engine of choice when lives depend on speed and accuracy under uncertainty.

***Index Terms*—Ambulance Routing, Disaster Response, SUMO, TraCI, A* Algorithm, Dijkstra, Bellman-Ford, BFS, NetworkX, Graph Theory, Emergency Medical Services, OpenStreetMap, Simulation.**

I. Introduction: The Problem With Routing an Ambulance in a Disaster

A. Why Routing Is Harder Than It Looks

Most of us have used a navigation app on our phones. We type in a destination, and within seconds a blue line appears on the map showing us the best route. If there is a traffic jam on the main road, the app re-routes us through side streets. This seems almost magical, but the mathematics behind it has been well understood for decades. What is less understood — and far harder to solve — is what happens to that navigation system when the roads themselves are being destroyed faster than the app can update.

In a disaster scenario, ambulances face a unique and brutal version of this problem. A bridge collapses, cutting off the fastest route to the hospital. A building facade falls into the street, blocking the next-best route. A gas main ruptures, making a third street impassable. All of this can happen within the same hour, and the ambulance must navigate around each new obstacle without any advance warning.

Unlike a tourist re-routing around traffic, an ambulance driver making wrong turns or taking unnecessarily long detours is not merely inconvenient — the delay can be fatal. Medical research has established the concept of the "golden hour" of trauma care: the roughly sixty-minute window after a serious injury during which rapid medical intervention can dramatically increase a patient's chance of survival [10]. Every unnecessary minute added to an ambulance's travel time reduces the probability of a good outcome.

This study is about that problem. Not about the ambulance dispatch software, not about building mobile apps, but specifically about the mathematical algorithms that compute *which roads to take* — and how those algorithms behave when the roads keep changing.

B. The Research Question

There are several classical graph search algorithms that computer scientists use to find the shortest path between two points on a map. They are taught in every algorithms course and are the foundation of modern navigation. But they were not all designed with the same priorities in mind. Some are fast. Some guarantee the optimal solution. Some work better when they have extra information about where the destination lies. Some become dangerously slow on large city maps.

Our research question is straightforward: **In a controlled simulation of a disaster — where roads are blocked unexpectedly mid-route — which pathfinding algorithm gives an ambulance the best combination of route quality and recalculation speed?**

We test four algorithms: A* Search, Dijkstra's Algorithm, Bellman-Ford, and Breadth-First Search. We run each one in four increasingly difficult scenarios on a real map of Bengaluru, from a simple baseline (no obstacles) to a cascading double-crash scenario designed to mimic the sequential infrastructure failures described in recent academic research on disaster routing [1].

C. Why This Matters Beyond Computer Science

The findings of this study are not just theoretical. Emergency services in Indian cities are being modernized rapidly, and decisions are being made right now about which software to embed in dispatch systems. If a city-scale dispatch server is evaluating routes for a hundred simultaneous ambulances and it is using Bellman-Ford, our data shows it would spend tens of seconds per update cycle just computing paths — time that those vehicles could be spending moving. If it uses A*, those same updates complete in under 3 seconds total for the entire fleet.

The stakes make precise, evidence-based algorithm comparison not an academic exercise but a genuine engineering obligation.

II. Background: What Do We Already Know About Ambulance Routing?

A. The Classic Approach: Offline Routing

For much of the last two decades, scholars studying ambulance routing have worked under a convenient but unrealistic assumption: that all the information you need is available before you send the ambulances out. You know how many victims there are, where they are, how badly hurt they are, how long each road segment takes to traverse, and which hospitals have available capacity. With this complete picture, you can run an optimization algorithm before anyone moves, find the mathematically perfect assignment of ambulances to victims to hospitals, and execute the plan.

This approach, called *offline optimization*, has produced sophisticated and elegant mathematical models. Talarico et al. [2] produced one of the most comprehensive of these, introducing two exact mathematical formulations and a large benchmark dataset of 1,296 test cases that researchers still use today. Rabbani et al. [3] extended the offline model to account for the disturbing reality that a victim's medical condition is not static — a mildly injured person ("green code") who is not reached in time will deteriorate to critically injured ("red code"), and ultimately may not survive ("black code"). Their model incorporated time thresholds for these deteriorations.

The offline approach is intellectually rich and has produced algorithms that perform excellently in simulations. The fundamental problem, as any disaster responder will point out, is that *none of that prior knowledge exists in a real disaster*.

B. The Honest Approach: Online Routing

The more recent and more realistic body of research treats ambulance routing as an *online* problem: decisions must be made with only the information available right now, knowing that more — and different — information will arrive in the future.

Shiri, Akbari, and Salman [1] produced the most rigorous formal study of this setting. They introduced the Online Ambulance Routing Problem (OARP), in which triage levels of victims, their required treatment times, and the travel times of roads are all initially unknown. This information is revealed incrementally: an ambulance learns the condition of a victim only when it physically arrives at their location, and it learns the true travel time of a road segment only when it is positioned at the start of that segment and can observe conditions in real time.

Through a type of mathematical analysis called *competitive analysis* — which compares the performance of an online algorithm (working with partial information) against an idealized offline optimal algorithm (working with complete information) — they proved a sobering theoretical result: *no online algorithm can guarantee a finite competitive ratio for the OARP*. In plain language, there exist worst-case disaster scenarios in which any online algorithm, no matter how clever, will perform arbitrarily worse than a hypothetical all-knowing system.

But their paper also showed something more encouraging: on real-world instance distributions, practical online heuristics achieve performance ratios very close to 1 (meaning they perform nearly as well as the theoretically perfect solution). Theory and practice diverge, and in practice, smart adaptive algorithms work well.

Our simulation study builds directly on this insight. We implement the routing philosophy of the OARP — incremental information revelation, online graph modification when roads are blocked — in a concrete simulation environment, and we empirically measure how four different algorithms behave under those conditions.

III. The Simulation Environment: Building a Virtual Disaster

A. Why Simulation?

Before describing the tools, it is worth explaining why simulation is necessary at all. The obvious alternative would be to run real ambulances through a real city and deliberately block roads to see how the algorithms perform. This is, of course, entirely impractical. Deliberately blocking roads in Bengaluru to observe algorithm latency would endanger real people, damage infrastructure, and be ethically indefensible.

What we need instead is a *controllable, reproducible, realistic virtual environment* in which:

- Every vehicle move can be traced and measured precisely.
- Roads can be blocked and unblocked at any moment with a single command.
- The entire experiment can be run dozens of times under identical conditions.
- The road network is a real city, not a simplified theoretical graph.
- Computational timing can be measured to the microsecond.

The combination of SUMO and the NetworkX graph library, controlled through the TraCI programming interface, provides exactly this environment. Let us describe each component in detail.

B. SUMO: A Virtual City in Software

SUMO, which stands for **Simulation of Urban MObility**, is an open-source traffic simulation tool developed by the Institute of Transportation Systems at DLR, the German Aerospace Center [4]. It has been used in traffic research for over two decades and has become one of the standard tools for studying vehicle behavior in road networks.

The key thing that makes SUMO genuinely useful for this study — rather than merely impressive — is that it does not simulate traffic in abstract or symbolic terms. It simulates individual vehicles as discrete physical agents. Each vehicle in the simulation has:

- A specific body length, maximum speed, acceleration profile, and braking distance.
- A reaction time, modelling the delay between a driver noticing a signal and responding to it.
- Awareness of other vehicles around it, maintaining safe following distances automatically.
- A planned route expressed as an ordered sequence of road segment identifiers.

The road network itself is represented with similar fidelity. Individual lanes have widths and curvatures. Intersections encode right-of-way rules. Traffic signal timings are specified per intersection. Roads have access restrictions (some lanes do not permit emergency vehicles; some roads are pedestrian-only).

For our study, we used real Bengaluru road data. The road network was exported from OpenStreetMap using a tool called JOSM and converted into a SUMO-compatible format using a utility called `netconvert`. The resulting simulation network contains thousands of directed road segments and junctions faithfully reflecting actual street connectivity in Bengaluru, including wide arterial highways, narrow residential lanes, and one-way streets.

C. TraCI: The Control Bridge Between Python and the Simulator

SUMO on its own runs a simulation and shows you vehicles moving on a map. To conduct our experiments, we needed to be able to control what happens in the simulation *from a Python program* — to spawn vehicles, measure positions, block roads, re-route ambulances mid-journey, and record timing data, all programmatically.

This is what the Traffic Control Interface, or **TraCI**, provides. TraCI is a library that opens a network connection to a running SUMO instance and exposes a rich set of commands that let an external program query and modify the simulation in real time.

Think of it as a remote control for the virtual city. Our Python script holds the remote and uses it to:

- **Add vehicles to the simulation:** We can spawn an ambulance on any road segment at any time, assign it a starting route, and give it a visual appearance (color, vehicle shape) for easy identification in the simulation viewer.
- **Re-assign a vehicle's route in real time:** At any simulation step, we can give a moving ambulance a completely new ordered list of road segments to follow, effectively re-routing it instantly. This is how we implement on-the-fly detour calculation.
- **Query a vehicle's current position:** At every step, we can ask "which road segment is ambulance X currently on?" This lets our algorithm always know where the vehicle is before computing its next route.
- **Measure total distance traveled:** Each vehicle's odometer can be read at any time, giving us the exact number of metres the ambulance has driven since the simulation started.
- **Query a vehicle's planned route:** We can read the full list of upcoming road segments a vehicle intends to follow. This is critical for our scenario design: to block the road in front of an ambulance, we first read its planned route and then block the road a few segments ahead.
- **Add visual markers:** We can place colored circular markers on the map to represent hospitals, crashed vehicles, or other points of interest. This makes the simulation visually understandable when watching it run.
- **Detect when vehicles arrive:** The simulation can report which vehicles reached their destination in the most recent step, allowing us to record precise arrival times.

The interaction loop in our experiment follows a simple rhythm. Every simulation step, our Python script sends a few queries to SUMO, receives the answers, and then decides whether to re-route any vehicles or inject any new events (such as a road blockage). This tight loop repeats thousands of times during each scenario run.

D. NetworkX: Building the Mathematical Map

SUMO handles the visual and physical simulation of vehicles. But SUMO itself does not have a built-in system for running advanced graph search algorithms. To find paths, we use **NetworkX**, a Python library specifically designed for working with graphs [11].

A graph, in mathematical terms, is simply a collection of nodes (points) connected by edges (lines between points). In our case, each node in the NetworkX graph represents a road segment in the SUMO simulation, and each directed edge represents a valid connection from one road segment to another (meaning a vehicle can physically travel from one to the other without turning illegally).

The weight attached to each edge represents how long it takes to drive that road segment: concretely, the weight is the length of the road in metres divided by the speed limit in metres per second. A longer road with a lower speed limit has a higher weight than a short road with a high speed limit,

meaning the algorithm correctly treats it as "more expensive" to traverse.

Before we run any experiments, we apply two important filters to this graph:

1) We remove any road segments that do not permit emergency vehicles. Ambulances have legal access to most roads, but certain pedestrian zones or restricted areas must be excluded.
2) We ensure the graph is *strongly connected*, meaning there is a valid path from every node to every other node. Without this check, an algorithm might try to plan a route to a destination that is topologically unreachable, producing an error rather than a path.

This NetworkX graph is where all four routing algorithms operate. When a road is blocked in our scenario, we do not modify anything in SUMO directly — instead, we set the weight of that road's corresponding node in the NetworkX graph to infinity. The next time any of the four algorithms searches for a path, they will naturally avoid the infinite-weight node and route around the blockage.

E. How Road Blockages Are Simulated

One of the most important mechanisms in our study is the simulation of a road blockage. This is worth describing in detail because it is the core of what makes the disaster routing scenarios realistic.

When we want to block a road at a particular moment in the simulation, our Python controller performs four actions simultaneously:

Step 1 — Identify the road to block: We query the current planned route of the A* ambulance and select a road segment several steps ahead in its journey (specifically, the fifth upcoming segment in Scenario 2, and the seventh in the second crash of Scenario 4). By choosing a road *ahead* of the ambulance rather than under it, we ensure the blockage represents a genuinely unexpected obstacle encountered during travel, not something the ambulance knew about when it departed.

Step 2 — Mark the road as blocked in the routing graph: In the NetworkX graph, we set the weight of all edges leading into that road segment to positive infinity. This is a mathematical way of saying "this road no longer exists as a viable option." Every algorithm that subsequently searches for a path will automatically exclude it.

Step 3 — Visualize the blockage in SUMO: We spawn a stationary orange truck on the blocked road segment. This truck has a speed of zero and will never move. It serves no algorithmic purpose — the routing logic ignores it completely — but it makes the blockage *visible* in the SUMO graphical interface. When you watch the simulation, you can see the orange truck sitting on the road and watch the ambulances re-route around it in real time.

Step 4 — Prevent automatic clearance: SUMO has a built-in traffic jam resolution feature that can automatically teleport stuck vehicles away to unclog traffic. We disable this feature for our crash trucks by setting a configuration parameter to minus one, ensuring that once a road is blocked, it stays blocked for the remainder of the scenario.

The combination of these four steps creates a blockage that is both algorithmically real (the route can never go through that road again) and visually compelling (the orange truck sits there as a clear physical obstacle).

IV. The Four Routing Algorithms: How Each One Thinks

To understand why one algorithm outperforms another, we need to understand what each algorithm is actually doing when it searches for a path. This section describes each algorithm in plain language, explains its strengths and weaknesses, and then places it in the context of the ambulance routing problem.

A. Breadth-First Search: The Naive Wanderer

Breadth-First Search, or BFS, is one of the oldest and most fundamental graph search algorithms. It works by exploring a graph one "ring" at a time, moving outward from the starting point like ripples from a stone dropped in water.

Imagine you are at the starting road segment. BFS first examines all road segments directly reachable from your starting point (one step away). Then it examines all segments reachable from those (two steps away), and so on, spreading outward ring by ring until it finds the destination.

The key characteristic of BFS is that it treats *all road segments as equal*, regardless of how long they are or how fast vehicles can travel on them. In BFS's world, a ten-metre alley counts exactly the same as a kilometre-long arterial highway. It minimizes the number of road segments in the route, not the travel time.

This makes BFS very fast to compute — it explores the graph methodically and finds a route quickly. But in an urban road network, minimizing the number of road segments is almost never what you want. You might end up on a route that technically visits fewer roads but forces you to crawl along narrow residential streets for ten minutes, when you could have taken two fast highway segments and arrived in three minutes. In our study, BFS serves as a *negative control*: a baseline that lets us measure exactly how much penalty is incurred when you completely ignore road travel times.

B. Bellman-Ford: The Thorough But Exhausted Explorer

The Bellman-Ford algorithm, introduced by Richard Bellman and Lester Ford in the late 1950s [9], was designed to find shortest paths in graphs that might contain negative-weight edges. A negative-weight edge is one where traversing it actually reduces your total cost rather than increasing it — a concept that makes sense in some financial models but is physically impossible in a road network where driving always takes time.

For our purposes, negative weights never occur: roads always take some positive amount of time to traverse. But Bellman-Ford does not know this in advance, so it employs a brute-force approach. It repeatedly examines every single edge in the entire graph, updating cost estimates over many

iterations, until it is certain it has found the true shortest path. Specifically, it performs exactly (number of nodes minus 1) passes over every edge.

On a small graph with a few dozen nodes, this is perfectly fine. On a large urban road network with thousands of nodes and tens of thousands of directed edges, this becomes enormously expensive. In our experiments on the Bengaluru road network, Bellman-Ford recalculation takes between 34 and 103 milliseconds per invocation. If a dispatch server were managing hundreds of ambulances and needed to recalculate routes every 40 seconds for each vehicle, the cumulative computation time from Bellman-Ford would dominate the system's resources and leave no computational capacity for anything else.

Bellman-Ford is included in our study not because it is recommended for this application, but because it represents the upper bound of computational cost for a correct algorithm — a useful reference point for understanding how much faster smarter approaches are.

C. *Dijkstra's Algorithm: The Gold Standard*

Dijkstra's algorithm, published by Edsger Dijkstra in 1959 [7], is the classic answer to the shortest-path problem. It is what most GPS navigation systems are based on, either directly or through accelerated variants. Unlike Bellman-Ford, Dijkstra is designed specifically for graphs with non-negative weights — exactly like our road network.

Dijkstra works by maintaining a priority queue: a sorted list of nodes ordered by how cheaply they can be reached from the starting point. At each step, it takes the cheapest node from the queue, examines its neighbors, and updates their cost estimates if it has found a cheaper way to reach them. The process continues until the destination node is extracted from the queue, at which point the cheapest path has been found.

The elegance of Dijkstra is that it is *provably optimal*: when it says it has found the shortest path, it has. There is no risk of missing a cheaper route. For non-negative road networks, it is correct by design.

The limitation is that Dijkstra searches in all directions simultaneously, like a spreading flood of exploration. From the starting point, it gradually expands to cover all points in the graph that are reachable at low cost, regardless of which direction the destination lies in. On a city map where the destination is to the north, Dijkstra will spend a substantial fraction of its time exploring roads to the south, east, and west before it converges on the correct northward path. This directionless behavior is not a bug — it is what guarantees correctness — but it does mean Dijkstra examines far more of the map than necessary before finding its answer.

D. *A* Search: The Informed Navigator*

A* Search, developed by Hart, Nilsson, and Raphael at Stanford Research Institute in 1968 [8], addresses Dijkstra's main weakness in an elegant way. Instead of expanding in all directions equally, A* uses additional information — called a *heuristic* — to guide its search toward the destination.

The heuristic is a function that estimates, for any given point in the graph, "how far is this point from the destination?" If the estimate is never an overestimate (a property called *admissibility*), then A* is mathematically guaranteed to find the optimal path, just like Dijkstra. But it finds it much faster, because it prioritizes exploring nodes that are both cheap to reach *and* close to the destination.

Think of the difference this way. Dijkstra behaves like a person searching for a friend's house by methodically knocking on every door in a spiral pattern from their current position, starting with the closest. A* behaves like a person who knows the friend's house is roughly to the north and therefore focuses their search in that direction, only checking doors to the south if the northward search proves fruitless.

In our simulation, the heuristic for A* is geographic: for any road segment in the graph, we compute the straight-line Euclidean distance from that segment's position to the destination, and divide it by a reference speed representing the fastest possible travel through the network. This gives a lower-bound estimate of travel time — it can never be a overestimate because the straight-line distance is always shorter than any real road path, and the reference speed is never faster than actual road speeds. This satisfies the admissibility condition, meaning A* remains optimal.

The practical effect is dramatic: while Dijkstra might examine 5,000 road segments before finding the path to a destination 4 kilometres away, A* might examine only 800, because its geographic heuristic prevents it from wasting time on roads in the wrong direction. This translates directly to computation time, which is where A*'s 2.4 millisecond mean latency advantage becomes visible.

V. The Four Scenarios: From Simple to Catastrophic

We designed four simulation scenarios of increasing complexity. Each one tests a different aspect of algorithmic performance in the ambulance routing context. Together, they trace a path from the simplest possible routing task to a realistic approximation of a cascading infrastructure failure in a post-disaster city.

For all scenarios, we run four ambulances simultaneously through the Bengaluru road network — one navigated by each of the four algorithms. This allows direct comparison because all four vehicles experience the same road network state at every moment of the simulation.

A. *Scenario 1: Random Routes — Testing Algorithmic Robustness Under Uncertainty*

1) What this scenario tests: Real disasters do not allow the luxury of controlled conditions. An ambulance may be dispatched at any moment from any location to any destination. Scenario 1 tests how well each algorithm handles *completely random* starting and ending points, with no pre-cached information about the network and no predictable route geometry.

2) How it works: At the beginning of the simulation, our controller filters the Bengaluru road network to include only emergency-accessible segments within the largest connected component. From this filtered set, a starting road segment is selected uniformly at random. The destination is then determined by finding the road segment whose start point is geographically farthest (by straight-line distance) from the start — this maximizes the length and difficulty of the routing problem.

Four ambulances are then spawned at the starting segment simultaneously. Each runs its own algorithm to find a path to the destination. They begin navigating, and neither the network nor the scenario throws any surprises at them: no road blockages, no obstacles.

3) Why it matters: This scenario primarily validates that our simulation infrastructure works correctly across diverse geographic conditions. If any algorithm generates a route that SUMO cannot execute (for example, a sequence of road IDs that involves an illegal turn), the experiment fails and we must revisit our graph construction. The fact that all four algorithms complete this scenario without issue confirms the topological integrity of our NetworkX graph. It also provides useful baseline data about relative algorithmic performance when no adversarial conditions are present.

B. Scenario 2: Static Hospitals with a Single Road Blockage

1) What this scenario tests: This scenario directly simulates the single most common emergent failure in disaster ambulance routing: a previously valid route becomes suddenly impassable due to an accident, debris, or structural collapse. The ambulance is already moving toward its destination when it discovers the blockage and must immediately find an alternate path.

2) How it works: Unlike Scenario 1's random endpoints, Scenario 2 uses fixed, deterministic hospital positions. These are the two road segments in the Bengaluru network that lie at opposite geographic extremes: the northwesternmost accessible segment becomes the Start Hospital, and the southeasternmost accessible segment becomes the Destination Hospital. Both are marked with large colored visual markers in the simulation viewer. Using fixed positions makes the scenario fully reproducible and ensures we are comparing algorithms on identical route challenges every time.

All four ambulances depart from the Start Hospital simultaneously. For the first 50 simulation steps, they travel freely. At simulation step 50, our controller intervenes:

1) It reads the current planned route of the A* ambulance.
2) It identifies the 5th road segment ahead in that route.
3) It blocks that segment in the NetworkX graph (infinite weight) and spawns an orange crash truck on it in SUMO.

All four ambulances are recalculating their routes every 40 simulation steps as part of the normal operation of the experiment. The blockage event at step 50 therefore takes effect at the next recalculation cycle (step 80, approximately 30 seconds later in simulation time). At that point, all four algorithms detect that one of their planned road segments has infinite weight and independently compute new paths around it.

3) Why it matters: This scenario is the core test of adaptive routing. Algorithms that can quickly find high-quality detours under this single-obstacle constraint will perform well in real-world single-incident dispatch. The key metrics here are recalculation latency (how fast does each algorithm find the new route?) and route quality (how much longer is the detour compared to the unobstructed path?).

C. Scenario 3: Static Hospitals, No Obstacles — The Performance Ceiling

1) What this scenario tests: Before we can meaningfully discuss how much a road blockage degrades performance, we need to know what the *best achievable performance* looks like under ideal conditions. Scenario 3 establishes this baseline.

2) How it works: The setup is identical to Scenario 2: same Start Hospital, same Destination Hospital, same four ambulances departing simultaneously. The only difference is that no blockage event is ever triggered. The ambulances travel freely from start to finish with the road network in its initial, fully accessible state.

3) What it tells us: The arrival times and total distances recorded in this scenario define the performance ceiling for each algorithm. By subtracting Scenario 3 numbers from Scenario 2 and Scenario 4 numbers for each algorithm, we can precisely quantify the penalty that each road blockage imposes: how many additional simulation steps did the detour add, and how many additional metres did the ambulance have to drive?

This clean subtraction is only possible because the scenarios are designed to be identical except for the deliberate roadblock injections.

D. Scenario 4: Cascading Double Road Blockage

1) What this scenario tests: This is the most sophisticated and most realistic scenario in the study. Its design is directly inspired by a key observation in the OARP research literature [1]: in real post-disaster scenarios, road conditions do not fail once and stabilize. They fail *sequentially and recursively*. An ambulance re-routes around one blocked road and then discovers that the detour it just committed to also has a new obstacle on it.

2) How it works: The scenario begins identically to Scenario 2. The first crash is injected at step 50, blocking the 5th future road segment of the A* ambulance's route. All four ambulances then re-route around this first obstacle at step 80. But here is where the scenario diverges and intensifies: at simulation step 150 — after all four ambulances have already committed to their detour routes — a *second* crash is injected. This time, we read the *current* route of the A* ambulance after it has already detoured, and block the 7th segment of that new route.

The ambulances now face a situation where two different road segments in the network are inaccessible simultaneously.

Each algorithm must, for the second time, search through the already-constrained graph to find a new path to the destination. The graph has fewer viable routes now than it did when the first crash occurred.

3) Why it matters: Scenario 4 answers a question that the researchers who study OARP consider essential: does the practical advantage of a smart algorithm (like A*) hold up under repeated adversarial disruptions, or does it degrade? Does the second re-route incur a much larger efficiency penalty than the first? Can any algorithm find a route at all when the graph has been pruned twice?

This scenario also measures the computational *resilience* of each algorithm: does Bellman-Ford's already-high latency get even worse when the graph state is more complex (two infinite-weight nodes instead of one)? Does A*'s heuristic guidance remain effective when the geographic direction of the destination has not changed but two previously valid paths in that direction are now blocked?

VI. Results: What Actually Happened

A. Scenario 3: The Unobstructed Baseline

Table I shows how each algorithm performed in the ideal, no-obstacle scenario.

TABLE I
UNOBSTRUCTED (BASELINE) PERFORMANCE — SCENARIO 3

Algorithm	Arrival (simulation steps)	Total Distance (m)
A* Search	323	4,322.6
Dijkstra	306	4,305.0
Bellman-Ford	308	4,311.5
BFS	362	4,265.2

The three weight-aware algorithms — A*, Dijkstra, and Bellman-Ford — all converge on essentially the same route, with physical distances within 17.6 metres of each other. The minor differences in arrival times reflect small random variations in microscopic vehicle behavior within the SUMO simulation (following distance enforcement, minor speed fluctuations at intersections) rather than any meaningful algorithmic difference. In an unobstructed environment, all three weight-aware algorithms find the same optimal path, as expected.

The BFS result deserves particular attention. BFS drove 4,265.2 metres — actually the *shortest physical distance* of all four algorithms. Yet it arrived *last*, at step 362 compared to Dijkstra's 306. How can the vehicle that drove the least distance take the longest time?

The answer illustrates exactly why BFS is wrong for road routing. BFS chose a route with fewer road-segment transitions — that is what it was designed to minimize. But those fewer segments included slow residential streets with 30 km/h speed limits, where the ambulance crept along for many more seconds per metre compared to the faster arterial roads selected by the weight-aware algorithms. BFS found the path with the fewest turns, not the path with the least travel time. In emergency response, these are very different objectives, and BFS optimizes the wrong one.

B. Scenario 2: After a Single Road Blockage

Table II shows the recalculation latency recorded across all route update cycles in Scenario 2.

TABLE II
ROUTE RECALCULATION LATENCY — SCENARIO 2 (SINGLE ROAD BLOCKED)

Algorithm	Min (ms)	Max (ms)	Mean (ms)
A* Search	1.01	4.88	2.40
Dijkstra	0.71	67.10	16.20
Bellman-Ford	34.80	103.60	67.40
BFS	0.35	1.04	0.62

These numbers require careful interpretation. At a glance, BFS appears to win this table with a mean of only 0.62 ms. And we already know from Scenario 3 that BFS produces inferior routes. So speed of computation alone is not the correct metric — we need *fast computation of the optimal route*.

A* achieves exactly this. With a mean recalculation time of 2.40 ms, it is nearly as fast as BFS, but it produces the time-optimal route. Dijkstra's 16.20 ms mean is about 6.75 times slower than A*, and Bellman-Ford's 67.40 ms mean is 28 times slower.

To put these numbers in practical context: imagine a city-scale dispatch server managing 1,000 simultaneously active ambulances, performing route recalculations every 40 seconds. With A*, each recalculation cycle would complete in approximately 2.4 seconds total for the entire fleet. With Dijkstra, it would take 16.2 seconds. With Bellman-Ford, 67.4 seconds — larger than the 40-second recalculation window itself, meaning the system would always be behind, never catching up.

Bellman-Ford's maximum latency of 103.60 ms is particularly alarming. It represents a scenario — presumably when the graph is in a particularly complex connectivity state — where a single route recalculation for a single vehicle takes over a tenth of a second. At city scale, this is operationally unacceptable.

C. Scenario 4: After Two Sequential Road Blockages

Table III shows arrival times and total distances after the second crash was absorbed.

TABLE III
PERFORMANCE AFTER CASCADING DUAL ROAD BLOCKAGES — SCENARIO 4

Algorithm	Arrival (steps)	Distance (m)	Extra vs. Baseline (m)
A* Search	319	4,309.6	−13.0
Dijkstra	305	4,328.6	+23.6
Bellman-Ford	309	4,311.5	±0.0
BFS	323	4,403.8	+138.6

These numbers tell a reassuring story for the three weight-aware algorithms, and a cautionary tale for BFS.

A*, Dijkstra, and Bellman-Ford all arrived within 5 simulation steps of their Scenario 3 baselines — despite absorbing

two completely unexpected road blockages mid-journey. They drove slightly different distances than the baseline (A* actually drove 13 metres *less* than its baseline, which is a quirk of the specific alternate route found), but the absolute deviations are tiny. The algorithms recovered cleanly both times.

BFS, by contrast, suffered a route length increase of 138.6 metres (a 3.3% increase over its already-suboptimal baseline distance) and arrived 39 simulation steps later than its own baseline (an 12% time penalty). Each time a road was blocked, BFS was forced to take an alternate path based on minimizing hops rather than minimizing time, and each alternate hop-minimizing path happened to involve even slower roads than the previous one. The blockages compounded BFS's fundamental weakness.

The most important result from Scenario 4 is the resilience demonstrated by A*: under the most adversarial conditions we designed, requiring two separate complete re-routes of an already-detoured path, A* found high-quality solutions in under 5 ms each time. Its heuristic guidance remained effective even as the graph became progressively more constrained.

D. Putting It All Together

Table IV provides a consolidated comparison across all dimensions studied.

TABLE IV
CONSOLIDATED ALGORITHM COMPARISON FOR DISASTER ROUTING

Metric	A*	Dijkstra	BF	BFS
Finds time-optimal path	Yes	Yes	Yes	No
Mean recalc. latency (ms)	2.40	16.20	67.40	0.62
Detour penalty (scenario 4, m)	−13	+24	±0	+139
Suitable for city-scale fleet	Yes	Limited	No	No

VII. DISCUSSION: WHAT THESE RESULTS MEAN IN PRACTICE

A. The Theory and the Simulation Agree

The academic study by Shiri, Akbari, and Salman [1] proved theoretically that online algorithms for ambulance routing — algorithms that work with partial and incrementally revealed information — can achieve near-optimal performance on real-world problem distributions, even though worst-case performance is theoretically unbounded. Our simulation results provide concrete empirical confirmation of this claim.

Even after two completely unannounced road blockages — a direct analogue of the incremental information revelation described in the OARP — our three weight-aware algorithms achieved performance within a few percent of their unobstructed baselines. If we define the "competitive ratio" for our Scenario 4 as the ratio of post-disaster performance to the ideal baseline, all three weight-aware algorithms achieve ratios of approximately 1.01 to 1.06. This aligns closely with the near-optimal experimental competitive ratios reported by Shiri et al. for their best-performing online heuristics.

BFS, which ignores edge weights and is therefore unable to incorporate the dynamic information (changing road availability and travel times) that makes adaptive routing effective, consistently underperforms. It can be interpreted as a metaphor for an offline plan that was drawn up before the disaster and then executed without any subsequent adaptation — exactly the approach that the OARP literature argues against.

B. Clear Engineering Recommendations

Based on our simulation results, we can make four concrete recommendations for engineers building emergency dispatch routing systems:

Use A* Search as your primary routing engine. Its combination of provably optimal path quality and fast recalculation latency (2.40 ms mean in our experiments) makes it uniquely well-suited to city-scale real-time routing. The geographic heuristic is straightforward to implement and naturally admissible on any road network.

Do not use Bellman-Ford in real-time systems. Its O(V × E) time complexity means that latency grows quadratically as the road network grows. On city-scale graphs, recalculation times of 34–103 ms per vehicle are operationally incompatible with real-time dispatch requirements. Bellman-Ford is an excellent algorithm for its intended purpose (graphs with negative weights), but road network routing is not that purpose.

Dijkstra is acceptable for small fleets with preprocessing. Dijkstra's 16.20 ms mean latency is substantially better than Bellman-Ford's and it produces optimal routes. For dispatch systems managing at most 50–100 simultaneously active vehicles, this may be acceptable. However, production systems should use Dijkstra with Contraction Hierarchy preprocessing (as in the OSRM routing engine), which reduces effective latency by orders of magnitude.

Never use BFS for travel-time routing. This point cannot be overstated. BFS consistently selected paths that minimized the number of road segment transitions rather than travel time, arriving last in every scenario despite sometimes driving shorter physical distances. Any emergency routing system based on BFS would consistently produce routes that feel intuitively short on a map but take longer to drive than weight-aware alternatives. The 56-step delay compared to Dijkstra in the baseline scenario alone represents approximately one minute of additional travel time — a potentially fatal delay in a mass casualty event.

C. Limitations of This Study

We are transparent about the constraints of our experimental design:

First, our simulation runs ambulances in isolation with no background civilian traffic. Real urban roads in Bengaluru at rush hour have very different travel time characteristics than our static speed-limit-based weights suggest. Integrating real-time traffic congestion into the edge weights would significantly change the absolute numbers, though we do not expect it to alter the relative ranking of the algorithms.

Second, our study uses a single city's road network. The relative performance of the algorithms might differ in cities with different topological characteristics — Manhattan's grid layout, for example, gives much less directional guidance

to A*'s geographic heuristic than Bengaluru's radial-arterial pattern, which could reduce A*'s latency advantage somewhat.

Third, this study does not model victim triage deterioration, multi-ambulance coordination, or hospital capacity constraints — all of which are central to the full OARP formulation. A more complete future study would integrate these elements, turning the simulation into a comprehensive multi-objective dispatch environment rather than a single-ambulance routing benchmark.

VIII. Conclusion

This paper asked a specific, practical question: which pathfinding algorithm should an emergency dispatch system use when routing ambulances through a road network that is being disrupted by an ongoing disaster?

We answered that question through a controlled simulation built on SUMO's microscopic traffic model, TraCI's real-time programmatic control interface, and NetworkX's graph algorithm library, operating on real Bengaluru road topology. Four algorithms — A*, Dijkstra, Bellman-Ford, and BFS — were tested across four scenarios of increasing adversarial complexity, from a clean baseline to a cascading dual-crash scenario inspired by the sequential infrastructure failures described in the OARP research literature.

The results are clear:

1) **A* Search** is the optimal choice for real-time emergency routing. It finds time-accurate optimal paths with a mean recalculation latency of 2.40 ms, recovers cleanly from multiple sequential road blockages, and scales to city-wide fleet management.
2) **Dijkstra** is correct and acceptable for smaller deployments, but at 16.20 ms mean latency it becomes a bottleneck at scale. With Contraction Hierarchy preprocessing it can be made production-viable.
3) **Bellman-Ford** is computationally unsuitable for any real-time routing application at city scale. Its mean latency of 67.40 ms and maximum of 103.60 ms would cause any large dispatch server to fall permanently behind.
4) **BFS** produces systematically slower routes due to its disregard for edge weights, arriving last in every scenario despite sometimes driving shorter distances. It must not be used in any time-sensitive routing context.

The core contribution of this study is the empirical validation, in a realistic simulation environment, of the theoretical insight from the OARP literature: adaptive, information-responsive routing algorithms can achieve near-optimal performance in practice, even under theoretically unpredictable disaster conditions. When roads break, A* finds another way — and it does so fast enough that the ambulance barely notices.

Future work will focus on three extensions: (1) incorporating dynamic traffic congestion into edge weights using real-time GPS probe data; (2) extending the simulation to multi-ambulance dispatch with hospital capacity constraints and victim triage deterioration timers; and (3) replacing the static Euclidean heuristic with a learned travel-time predictor trained on historical traffic data, potentially improving A*'s accuracy in congested conditions while maintaining its admissibility guarantee.

Acknowledgment

The authors gratefully acknowledge the SUMO development team at DLR (German Aerospace Center), whose open-source traffic simulation platform made this study possible; the OpenStreetMap contributors whose freely available geographic data provided the Bengaluru road network; and the NetworkX development community for their comprehensive graph algorithm library. The authors also thank the faculty and staff of the Department of Computer Science and Engineering for their support and guidance throughout this research.

References

[1] D. Shiri, V. Akbari, and F. S. Salman, "Online algorithms for ambulance routing in disaster response with time-varying victim conditions," *OR Spectrum*, vol. 46, pp. 785–819, Feb. 2024.
[2] P. Talarico, G. Meisel, and A. Soriano, "Ambulance routing for disaster response," *European Journal of Operational Research*, vol. 242, no. 3, pp. 1045–1056, 2015.
[3] M. Rabbani, H. Farrokhi-Asl, and B. Aghraei, "Dynamic ambulance routing with variable triage levels," *Computers & Industrial Engineering*, vol. 164, 2022.
[4] P. A. Lopez *et al.*, "Microscopic traffic simulation using SUMO," in *Proc. 21st IEEE ITSC*, Maui, HI, USA, 2018, pp. 2575–2582.
[5] D. Luxen and C. Vetter, "Real-time routing with OpenStreetMap data," in *Proc. 19th ACM SIGSPATIAL Int. Conf. Advances in Geographic Information Systems*, Chicago, IL, USA, 2011, pp. 513–516.
[6] M. Haklay and P. Weber, "OpenStreetMap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, Oct.–Dec. 2008.
[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.
[9] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.
[10] R. Adams, "Improving trauma care: The golden hour and beyond," *Emerg. Med. Clin. North Am.*, vol. 11, no. 1, pp. 1–13, 1993.
[11] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. 7th Python in Science Conference (SciPy 2008)*, Pasadena, CA, USA, 2008, pp. 11–15.
[12] V. Pillac, M. Gendreau, C. Gue´ret, and A. L. Medaglia, "A review of dynamic vehicle routing problems," *European Journal of Operational Research*, vol. 225, no. 1, pp. 1–11, 2013.
[13] F. S. Salman and E. Gu¨l, "Deployment of field hospitals in mass casualty incidents," *Computers & Industrial Engineering*, vol. 74, pp. 37–51, 2014.
[14] J. Yoon and L. Albert, "The dynamic ambulance routing problem with two vehicle types," *Int. J. Production Economics*, vol. 225, 2020.
[15] T. Tlili, S. Faiz, and S. Krichen, "An ambulance routing problem based on the open vehicle routing problem," *Procedia Computer Science*, vol. 112, pp. 1050–1059, 2017.
[16] M. Schilde, K. Doerner, R. Hartl, and G. Kiechle, "Metaheuristics for the dynamic stochastic dial-a-ride problem," *Computers & Operations Research*, vol. 38, no. 12, pp. 1719–1730, 2011.
[17] M. K. Oksuz and S. I. Satoglu, "A two-stage stochastic model for location planning of temporary medical centers for disaster response," *Int. J. Disaster Risk Reduction*, vol. 44, 2020.
[18] N. Geroliminis and C. F. Daganzo, "Existence of urban-scale macroscopic fundamental diagrams: Some experimental findings," *Transportation Research Part B: Methodological*, vol. 42, no. 9, pp. 759–770, 2008.