A* Implementation and Analysis for Emergency Vehicle Routing

Sean Ando (ando.se@northeastern.edu)
Ngoc Khanh Vy Le (le.ngoc@northeastern.edu)
Divit Pratap Singh (singh.divit@northeastern.edu)
Aayushi Saraswat (saraswat.a@northeastern.edu)
Zuzu Nyirakanyange (nyirakanyange.z@northeastern.edu)

INTRODUCTION

In a life-critical situation, emergency vehicles need to find the fastest route to save lives. These vehicles utilize GPS and standard navigation applications, which commonly use variations of algorithms such as Dijkstra's algorithm or rely on historical and average traffic data. Therefore, when there are unexpected traffic conditions like traffic jams or the road is closed, the route they picked as "fastest" might actually end up being much slower than an alternative, causing critical delays. Emergency vehicles need dynamic systems which not only give the best optimal shortest path in the current situation but also continually adapt and change as unexpected road changes happen. While traditional navigation systems may have a few minutes delay when updating to a new path and sometimes the drivers need to restart the app in order to get the updated paths, our approach will ingest the live road updates and recalculate the new routes simultaneously. This will ensure that they will never commit to a stale plan when a faster alternative emerges.

The goal is to cut down every extra minute in an emergency. To address this problem, we propose utilizing the A* search algorithm. We use A* because it is renowned for its reliability in quickly identifying optimal paths even under dynamically changing conditions and it always finds the shortest route and can re-route instantly when new jams or closures pop up. We will test it on simple grid maps with random blockages to see how quickly it finds a good path by analyzing its running time and comparing it to a regular GPS algorithm such as Dijkstra's.

SCENARIO DEFINITION

In our experimental scenario, we have an 8×8 grid representing simplified city streets. Each cell will represent a segment of roadway that can be either accessible or blocked to simulate dynamic and realistic traffic conditions. We will test the performance of the A* algorithm across varying

obstacle densities, at 0%, 20%, and 50%. Different conditions will allow us to evaluate the algorithm's effectiveness and efficiency in adapting to different levels of traffic disruptions. Each scenario systematically assesses the algorithm's runtime, the number of expanded nodes, and the optimality of the path produced.

Subsequently, the implementation is scaled up and integrated into a larger, more realistic grid environment, incorporating animated visualizations. This enhanced implementation offers real-time interactive capabilities, allowing dynamic selection of start and goal points. Additionally, it considers more complex, realistic urban scenarios, including evolving road conditions, and varying degrees of blockage severity. The integration aims to comprehensively demonstrate the algorithm's robustness and practicality for real-world application scenarios faced by emergency response systems.

IMPLEMENTATION

1. A* Algorithm

Our standard is A* is a base algorithm that will search an optimal path from s to goal t. The algorithm will find the shortest path by selecting the next node to explore by its total cost and by keeping track of the exact cost to reach each node from the start. For each discovered node, it will maintain:

- The exact cost g(n) of the lowest-cost path from the starting node s
- A heuristic h(n) of the remaining cost from current node to the goal t. In our implementation, h(n) is the Manhattan distance between n and t

Starting with the starting node s, the algorithm will repeatedly select and expand the node n with the smallest estimated total cost. We combine them into:

$$f(n) = g(n) + h(n)$$

And we keep all discovered nodes in a closed list of smallest f. Repeatedly, we:

- Remove the node n with smallest f from the open list
- Generate its neighbor
- For each neighbor we compute a new cost via the node n. If it is better than before, we update the g'(n), set it back-pointers and then we reinsert it into the open list with its update.
- Once all of its neighbors are visited, we move node n into the closed list.

a. Correctness:

- Prove the admissibility of Manhattan heuristic:

  We have:

$$h(n) = |x - x_{goal}| + |y - y_{goal}|$$

  be the Manhattan distance from node $n = (x, y)$ to the goal.

  - Every move can reduce the Manhattan distance by at most 1 while costing exactly 1, so no path from n to the goal can cost less than h(n). Therefore for every node n, so we have $h(n) \leq h^*(n)$where $h^*(n)$ is the true cost of the shortest path.
  - For any neighbor n' of n, we have $h(n) \leq 1 + h'(n)$ and the step from n' to n costs 1 $\Rightarrow h(n) \leq cost(n', n) + h'(n) \Rightarrow$ f = g + h is nonincreasing along any paths.

  Therefore, h(n) is admissible.

- Prove the optimality of A*:

  Claim: If h(n) is admissible, then A* always returns the shortest path from start to goal.

  Prove by contradiction: Assume that the algorithm does not return a true optimal path. There are three cases:

  - Case 1: Terminates at non-goal node ⇒ Does not hold because the code will only return if current_node is goal so the algorithm wont stop at other notes.
  - Case 2: The algorithm will run infinitely ⇒ Does not hold because every move costs 1 and the grid is finite which means we can just run a finite step. Hence after many expansions we will exhaust all the nodes in open_set and eventually reach the goal.
  - Case 3: The algorithm will terminate at the goal node but it does not achieve the minimum-cost g*:
    + Suppose the algorithm terminates at goal node t and there is an actual optimal path P, and let's assume there is a node n_j in path P where j is between node 0 and node k that the algorithm does not expand to when it picks node t:

$$P = [n_0, \dots, n_j, \dots, n_k]$$

+ We have all nodes between $[n_0, \dots, n_{j-1}]$ must be visited and in the closed_set but node j is still in the open_set. Then n_{j-1} was expanded, so when relaxing its neighbors A* inserted n_j with $g(n_j) = g^*(n_j)$
+ By admissibility of the Manhattan heuristic we know that: $f(n_j) = g(n_j) + h(n_j) \leq$ cost + $h^*(n_j)$ = cost + k - j = g* meanwhile, at the goal node t $f(n_t) = g(n_t) + h(n_t) > g^* \Rightarrow f(n_j) < f(n_t)$ => since A* always chooses the open-set node of smallest f, it would have expanded n_j before n_t. Therefore it has to visit j before it visits and stops at t ⇒ The claim does not hold.

Since all 3 cases proved are false ⇒ the assumption is false ⇒ The algorithm does return an optimal path.

b. Time Complexity:

Since we run the algorithm in a n x n grid. Grid size: V = n × n = n^2 cells

Each vertex has at most 4 edges leading to its neighbors so E = 4V

- **Best case:** the Manhattan distance heuristic which we use in the algorithm also runs perfectly in a grid with no obstacles
  - Since there are no obstacles, the algorithm only expands nodes on the shortest path. The length l will be at most 2n -> O(2n) = O(n)
  - Pop one heap: O(logV) each node
  - Push one heap: O(logV) each node, up to 4 nodes

  Therefore, running time: O(n) x O(logV) = O(n.logn^2) = O(n.logn)
- **Average case:** Since there are no obstacles, the average case will run closely to its best performance and the Manhattan distance heuristic also runs perfectly in a grid with no obstacles.
  - Manhattan heuristic gives the same f = 2(n–1) for every cell since there are no obstacles, the algorithm will expand every node in the diamond of radius (n – 1) → O(V) expansions

- Pop one heap: O(logV) each node
- Push one heap cost O(logV) each node, up to 4 heap per node: O(4V.logV) = O(E.logV)

Therefore, the running time: O(V.logV + E.logV) = O(5V.logV) = O(V.logV) = O(n^2.log(n^2)) = O(n^2.logn)

- **Worst case:** In the worst case A* will expand every reachable cell, doing one heap-pop and up to 4 heap-pushes per cell. Each of V node expansion does:
  - Pop one heap: O(logV), which can pop up to V times.
  - Check 4 neighbors: O(1) each
  - Push each heap: O(logV), which applies for each at most once, up to E edges.

  Therefore, the running time:

$$\begin{aligned} O(V.\log V + E.\log V) &= O(5V.\log V) \\ &= O(V.\log V) \\ &= O(n^2.\log(n^2)) \\ &= O(n^2 \log n) \end{aligned}$$

c. Space Complexity:

In the worst case scenario, A* has to check every vertex in the grid and it also needs to store a graph:

- O(V) for an open and close list
- O(V + E) to hold the graph's adjacent structure

The overall space complexity will be O(V+E). However, in this project, we run in an n x n grid, where $V = n^2$ and $E \approx 2n(n-1) = 2n^2 - 2n$. Therefore, the final space complexity is: $O(n^2)$

2. A* Algorithm with obstacles

a. Modification:

The initialization will be the same:

- Input: n x n grid, s is a start node and t is the goal node
- Open list is always ordered by smallest f
- The closed list contains the node that we fully visited all of its neighbors.

- Randomly spawn the obstacles in the grid where 1 is the node that has an obstacle and 0 when it does not contain an obstacle.

Except in the main while loop:

- Insert s into the open list with

$$g(s) = 0,\ h(s) = heuristic(s,t),\ f(s) = g + h$$

- Repeatedly remove from the open list the node n with the smallest f
- If n = t, we stop the loop.
- Otherwise, we will explore its neighbors: Skip if the node contains obstacle, if it does not we proceed checking like in standard A* with no obstacles.
- After we checked all the neighbors of node n, we will put it into the closed list.

b. Time Complexity

Algorithm returns the path scenario:

- Let V is the number of free cells, E is the number of edges, p the probability that the cell will have obstacles, d is the length of the shortest path and R is the number of free cells that the algorithm actually visited.
- Similar to the standard A*, in the worst case the algorithm has to traverse all the free nodes to find the shortest path. Since in this case we have p chances that the node can have obstacles. So the final running time when we have path will be

$$\mathrm{O}((1-p)n^2\log n)$$

Algorithm could not find the path scenario: In the case where is no path existed, which mean all the obstacles block the path ⇒ A* will have to explore entire free nodes that are reachable from s before it returns no path, let's call it size C. The final running time will be:

$$O(C.logC) \text{ where } C < V$$

c. Space Complexity

Algorithm returns the path scenario:

- Open list (heap) → up to O(R)
- Closed set → up to O(R)
- Store g-scores → up to O(R)

- Costs to visit order → up to O(R)

Therefore the space complexity will be O(R)

Algorithm returns the path scenario: In the case where no paths exist, R will be the same size as C where C is the number of free nodes that is reachable from the start node s. So the space complexity in this case will be O(C).

3. Dijkstra's (GPS):

a. No obstacles:

Time Complexity: In standard Dijkstra's binary heap with no obstacles and the running time is O((V + E) lo V). Since we are running in a n x n grid where V = n^2 and E = 4V = 4.n^2. Therefore the final running time will be:

$$O((V + E) \log V) = O(5.V.\log V) = O(n^2.\log n^2) = O(n^2.\log n)$$

Space Complexity: In Dijkstra's, the space complexity is O(V) = O(n^2)

b. Obstacles:

Modification:

- The algorithm will take an input n x n grid, start node and goal node
- Consider every grid as a graph where each empty cell will be 0 and the cell with an obstacle will be 1. Each edge will connect with 4 neighbors. There will be no negative nodes.
- When running Dijkstra's, we will skip the neighbor that has value 1.
- We use min-heap keyed by the current best distance from the start
- When we pop a node from the heap, that distance is final, so we mark it closed.
- Stop when we pop the goal.

Time Complexity: Let V is number of free cells since this is n x n grid ⇒ $V < n^2$ and Algorithm returns the path scenario:

- A significant portion of the grid is explored, often O(V) nodes.
- Operations: Each node pop/push is O(log V).
- Total: O(V log V) = O(n^2 log n) ≈ O($(1-p)\, n^2 \log n$) where p is the probability that a node will have obstacle

Algorithm could not find the path scenario: In the case where is no path existed, which mean all the obstacles block the path:

- Pop min: O(log V) × V times → O(V log V)
- Neighbor checks: O(1) × E → O(V)
- Push/update: O(log V) × E → O(V log V)
- Total: O(V log V) = O(n^2log n) = $O(n^2 logn) \approx O((1-q)n^2 logn)$ where p is the probability that a node will have obstacle

Space Complexity: Similar to standard Dijkstra's:

- Store the distance → up to R
- Add into visited list → up to R
- Heap (open set) → up to O(R) at once
- The grid: $O(n^2)$
- So auxiliary space used by the algorithm's data structures, including the grid storage, is: $O(n^2)$

COMPARISON WITH DIJKSTRA'S

1. General analysis between A* and Dijkstra

Based on our analysis, both A* and Dijkstra use the same space in this grid settings which is $O(n^2)$ (for the grid plus the usual maps/sets/heap; worst case both store $\ominus(n^2)$ entries).

For time complexity it is based on different scenarios:

a. No obstacles:

- A*: expands roughly along the path only. If the shortest path length is and Final running time: O(n log n)
- Dijkstra: must fill the whole radius-ddd "diamond" from the start (about $\Theta(d^2)$ cells) and final running time is: O(n^2log n)

Conclusion: A* runs faster than Dijkstra without obstacles.

b. With obstacles (each cell blocked with probability of p)

Let $V_{free} \approx (1-p)n^2$ be the number of free cells.

- A*: Time: $O(R_A logR_A) \approx O(n^2 logn)$ (where RA$\ll V_{free}$).

- Dijkstra: $O(n^2 logn)$
- Worst case (both): explore essentially all free cells:$O((1 - q)n^2 logn)$

Conclusion: With obstacles, both share the same big-O worst case, but A* usually expands fewer nodes. We will future explore this aspect practically in the next section

c. No paths exist:

Let C be the size of the reachable free region from s($C \leq V_{free}$). A* and Dijkstra both end up exploring that whole region which is $O(C. logC)$

Conclusion: Same asymptotic bound but A* still tends to expand no more than Dijkstra.

2. Quantitative analysis of A* and Dijkstra

We conducted a practical evaluation using Python implementations of the A* and Dijkstra pathfinding algorithms. Our tests were structured to analyze the algorithms' performance and effectiveness in grid environments simulating various obstacle scenarios. To generalize the findings, we conducted an additional case of 30×30 grid to assess how increasing the grid environment affects the performance of the algorithms (A link to the detailed data and maps is provided for reference). The grid maps were generated randomly, representing three distinct conditions based on obstacle density:

- No Obstacles (0%): An optimal and ideal scenario with no obstacles.
- Moderate Obstacles (20%): Representing moderate conditions such as minor traffic disruptions or temporary blockages.
- High Obstacles (50%): Simulating severe conditions akin to widespread disruptions or emergencies, such as natural disasters or major incidents.

Scenario evaluations:

1. No Obstacles (0%) Scenarios:

a. Grid Composition: 0% obstacles, 100% free cells.

b. A* Algorithm Performance: Expanded 14 nodes with a computation time of 0.000021 seconds, indicating near-instantaneous computation and exceptional efficiency.

c. Dijkstra Algorithm Performance: Expanded significantly more nodes, 63 nodes, and required a computation time of 0.000109 seconds, showcasing the inefficiency of

exhaustive exploration compared to heuristic-based methods (this clearly illustrates Dijkstra's inherent limitation compared to A* due to the absence of heuristic guidance).

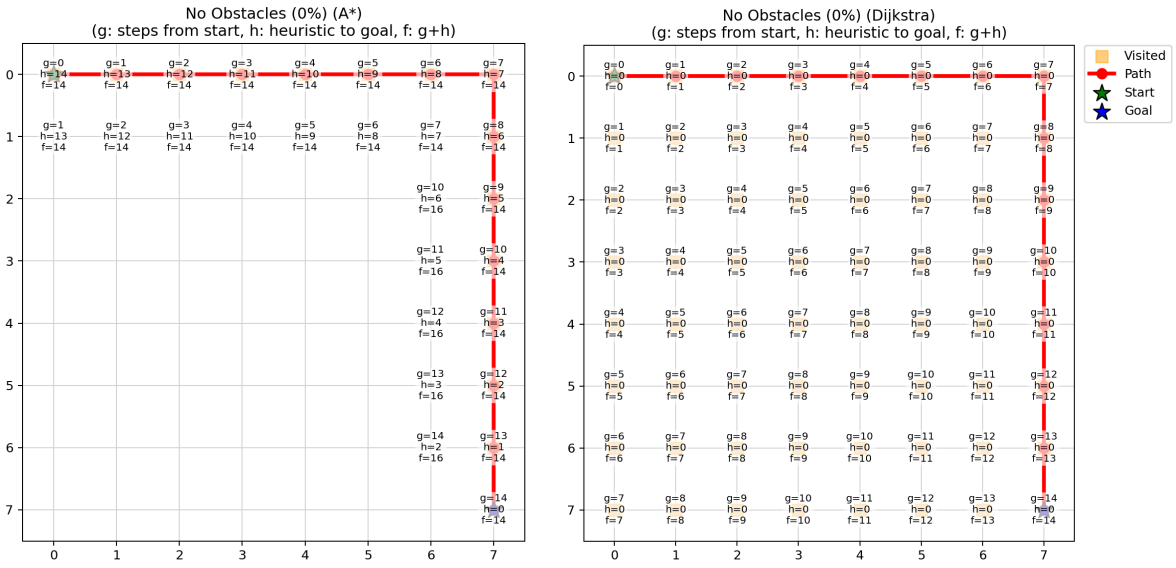

Figure 1: Visualization of A* and Dijkstra's when no obstacles in 8x8 grids

2. Moderate Obstacles (20%) Scenario:

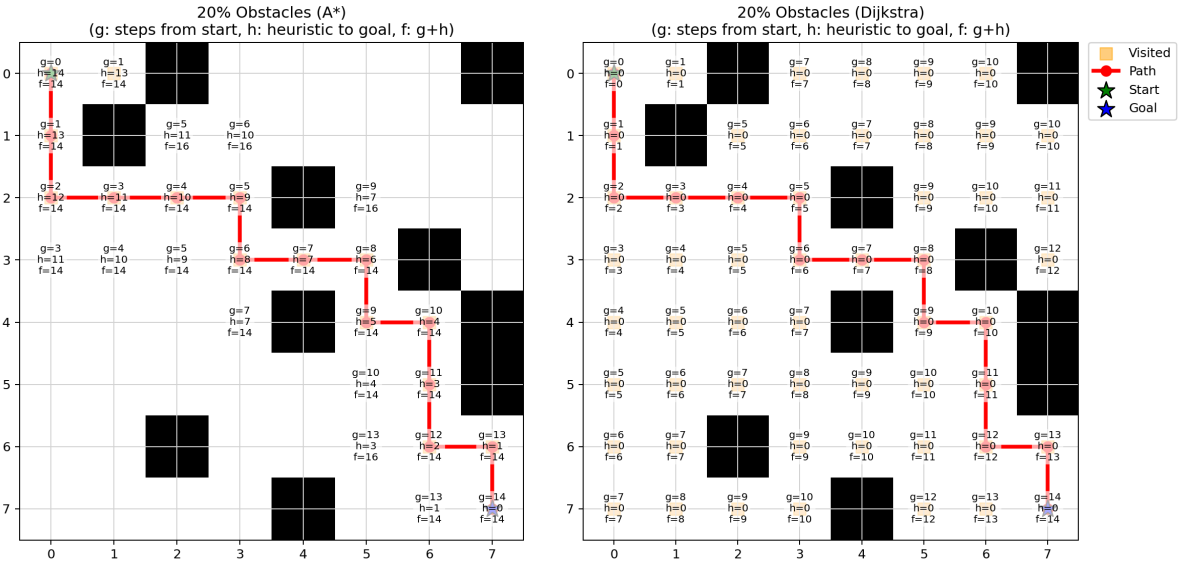

Figure 2: Visualization of A* and Dijkstra's when having 20% obstacles in 8x8 grids

a. Grid Composition: Approximately 15.6% obstacles, 84.4% free cells, effectively representing moderate traffic disruption scenarios.

b. A* Algorithm Performance: Expanded 15 nodes with a computation time of 0.000078 seconds. Despite encountering minor obstacles, A* effectively recalculated paths, demonstrating robust adaptability and maintaining high computational efficiency.

c. Dijkstra Algorithm Performance: Expanded 53 nodes with a computational duration of 0.000174 seconds, underscoring the additional computational resources required due to lack of heuristic guidance.

3. **High Obstacles (50%) Scenario:**

a. Grid Composition: 42.2% obstacles, 57.8% free cells, simulating severe traffic disruption scenarios.

b. A* Algorithm Performance: Expanded 25 nodes within a computation time of 0.000068 seconds, effectively identifying valid paths under complex conditions with minimal increase in computational demands.

c. Dijkstra Algorithm Performance: Expanded 34 nodes with a computation time in 0.000075 seconds. Although still reasonably efficient, this further underscores the computational overhead due to the algorithm's lack of heuristic-driven path optimization.

d.

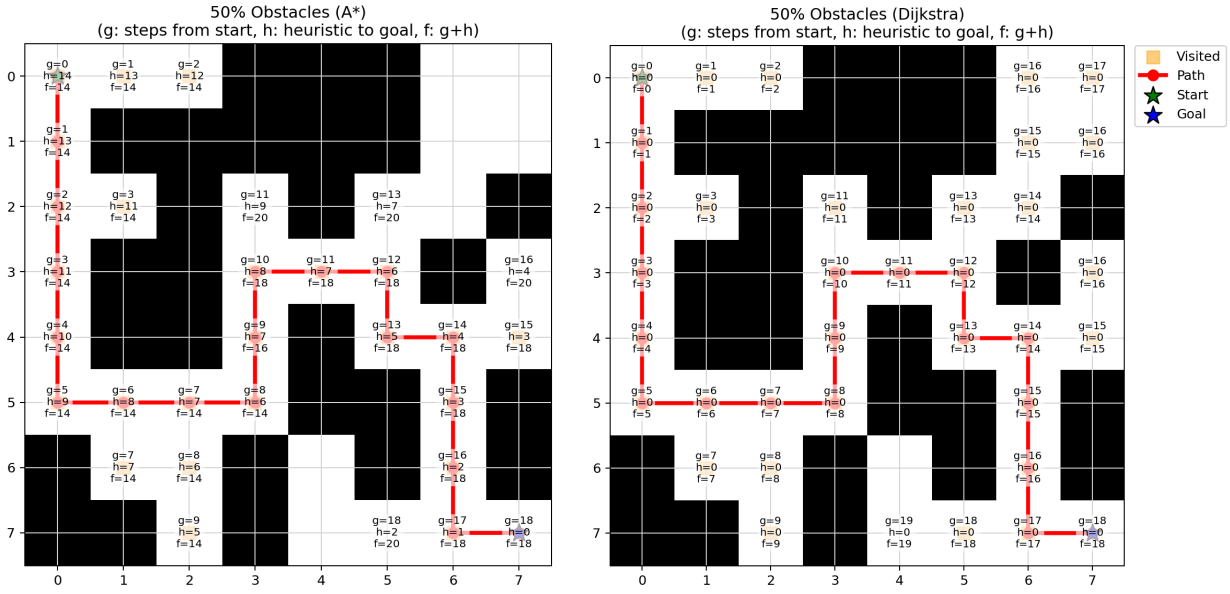

Figure 2: Visualization of A* and Dijkstra's when having 50% obstacles in 8x8 grids

As the grid size increases the search cost grows for both algorithms. However, across both grid sizes and all obstacle densities, A* returns optimal paths while consistently using less search effort and time than Dijkstra.

To justify this result, we need to evaluate 4 cases – per-scenario comparisons on the 8×8 grid, per-scenario comparisons on the 30×30 grid, and scale-up effect from 8×8 to 30×30 to show how much more work each algorithm does. And, the scale-up effect needs to examine the per-scenario scale factors (same algorithm and same scenario) and scenario-balanced scale-up (ratio of per-grid means) that assigns equal weight to each obstacle density to isolate grid-size effects. Then, we conduct a pooled (workload-weighted) comparison of A* vs Dijkstra across all six conditions (both grid sizes × three obstacle levels) by aggregating total nodes and runtime across conditions for each algorithm to yield a single overall effect size, such as how much less work and time A* uses more than Dijkstra.

For the per-scenario computation for both 8×8 and 30×30 girds, use the following calculation:

- Speedup in nodes: S_nodes = Dijkstra nodes / A* nodes
- Speedup in time: S_time = Dijkstra time) / A* time
- Percent reduction: R = 1 – (A* / Dijkstra) × 100 %

RESULT

8 × 8 Grid						
Scenarios (% Obstacles)	Obstacle (%)	Free Cells (%)	A*		Dijkstra	
			Expanded Nodes	Time (sec)	Expanded Nodes	Time (sec)
No (0%)	0.0%	100.0%	14	0.000021	63	0.000109
Moderate (20%)	15.6%	84.4%	15	0.000078	53	0.000174
High (50%)	42.2%	57.8%	25	0.000068	34	0.000075

30 × 30 Grid						
Scenarios (% Obstacles)	Obstacle (%)	Free Cells (%)	A*		Dijkstra	
			Expanded Nodes	Time (sec)	Expanded Nodes	Time (sec)
No (0%)	0.0%	100.0%	59	0.000108	899	0.001187
Moderate (20%)	20.7%	79.3%	78	0.000314	710	0.002126
High (50%)	47.3%	52.7%	111	0.000155	208	0.000288

Table 1: Data Comparison Chart and Table

On the 8×8 grid:

1. No obstacles
    a. Nodes speedup: S_nodes = 63/14 = 9/2 = 4.5
    b. Nodes percent reduction: R_nodes = 1 − 14/63 = 0.777777 = 77.7778%
    c. Time speedup: S_time = 0.000109/0.000021 = 109/21 = 5.19047
    d. Time percent reduction: R_time = 1 − 21/109 = 0.807339449 = 80.7339%
2. Moderate (20%)
    a. Nodes speedup: S_nodes = 53/15 = 3.5333
    b. Nodes percent reduction: R_nodes = 1 − 15/53 = 0.716981132 = 71.6981%
    c. Time speedup: S_time = 0.000174/0.000078 = 174/78 = 2.230769
    d. Time percent reduction: R_time = 1 − 78/174 = 0.551724138 = 55.1724%
3. High (50%)
    a. Nodes speedup: S_nodes = 34/25 = 1.36
    b. Nodes percent reduction: R_nodes = 1 − 25/34 = 0.264705882 = 26.4706%
    c. Time speedup: S_time = 0.000075/0.000068 = 75/68 = 1.102941176
    d. Time percent reduction: R_time = 1 − 68/75 = 7/75 = 0.093333333 = 9.3333%
4. 8×8 means (simple average across the three scenarios)
    a. Mean nodes speedup = (4.5 + 3.53333… + 1.36) / 3 = 3.1311
    b. Mean time speedup = (109/21 + 29/13 + 75/68) / 3 = 2.841395533
    c. Mean nodes percent reduction = average of (7/9, 38/53, 9/34) = 58.6488%
    d. Mean time percent reduction = average of (88/109, 16/29, 7/75) = 48.4132%

Thus, averaging across scenarios gives mean speedups of 3.13 in nodes and 2.84 in time, corresponding to mean percent reductions of nodes = 58.65% and time = 48.41%. That is, A* is more efficient.

On the 30x30 grid:

1. No obstacles
    a. Nodes speedup: S_nodes = 899/59 = 15.237288135
    b. Nodes percent reduction: R_nodes = 1 − 59/899 = 0.93437152 = 93.4372%
    c. Time speedup: S_time = 0.001187/0.000108 = 10.990740741
    d. Time percent reduction: R_time = 1 − 108/1187 = 0.909014 = 90.9014%
2. Moderate (20%)
    a. Nodes speedup: S_nodes = 710/78 = 355/39 = 9.102564103
    b. Nodes percent reduction: R_nodes = 1 − 78/710 = 0.89014084 = 89.0141%
    c. Time speedup: S_time = 0.002126/0.000314 = 6.770700637
    d. Time percent reduction: R_time = 1 − 314/2126 = 0.852304798 = 85.2305%
3. High (50%)
    a. Nodes speedup: S_nodes = 208/111 = 1.873873873
    b. Nodes percent reduction: R_nodes = 1 − 111/208 = 0.466346154 = 46.6346%
    c. Time speedup: S_time = 0.000288/0.000155 = 1.858064516
    d. Time percent reduction: R_time = 1 − 155/288 = 0.461805556 = 46.1806%
4. 30×30 means (simple average across the three scenarios)
    a. Mean nodes speedup = (899/59 + 355/39 + 208/111)/3 = 8.737906
    b. Mean time speedup = (1187/108 + 1063/157 + 288/155)/3 = 6.539835
    c. Mean nodes percent reduction = average of (840/899, 316/355, 97/208)
    = 0.7636195 = 76.3619%
    d. Mean time percent reduction = average of (1079/1187, 906/1063, 133/288)
    = 0.7410417 = 74.1042%

Thus, averaging across scenarios for the 30×30 grid gives mean speedups of 8.74 in nodes and 6.54 in time with mean percent reductions of nodes = 76.36% and time = 74.10%. That is, A* is more efficient.

To quantify how search cost grows when the grid increases from 8×8 to 30×30, the per-scenario scale factors (not balanced obstacle density) and a scenario-balanced scale-up factor (ratio of per-grid means) that obstacle density does not skew the result (equal weights across obstacle densities) are considered.

1. Per-scenario scale factors: (30×30 grid) / (8×8 grid)
    a. A* nodes
        i. No obstacles: 59/14 = 4.2142857
        ii. Moderate: 78/15 = 5.2
        iii. High: 111/25 = 4.44
    b. A* time
        i. No obstacles: 0.000108/0.000021 = 108/21 = 5.1428571
        ii. Moderate: 0.000314/0.000078 = 314/78 = 4.0256410
        iii. High: 0.000155/0.000068 = 155/68 = 2.2794118
    c. Dijkstra nodes
        i. No obstacles: 899/63 = 14.2698413
        ii. Moderate: 710/53 = 13.3962264
        iii. High: 208/34 = 6.1176471
    d. Dijkstra time
        i. No obstacles: 0.001187/0.000109 = 10.8899083
        ii. Moderate: 0.002126/0.000174 = 12.2183908
        iii. High: 0.000288/0.000075 = 3.84

    Thus, enlarging the grid to 30×30 increases Dijkstra's cost by 11.26 in nodes and 8.98 in time, while increasing A*'s cost by 4.62 in nodes and 3.82 in times. That is, A* scales substantially better with grid size.
2. Scenario-balanced scale-up factor (ratio of per-grid means): (30×30 mean) / (8×8 mean)
    a. A* nodes scale factor
        i. 8×8 mean nodes = (14 + 15 + 25) / 3 = 18
        ii. 30×30 mean nodes = (59 + 78 + 111) / 3 = 248 / 3
        iii. Scale = (30×30 mean) / (8×8 mean) = (248/3) / 18 = 4.592592
    b. A* time scale factor

i. 8×8 mean time = (0.000021 + 0.000078 + 0.000068) / 3
ii. 30×30 sum time = (0.000108 + 0.000314 + 0.000155) / 3
iii. Scale = (0.000577/3) / (0.000167/3) = 3.45508982

c. Dijkstra nodes scale factor
i. 8×8 mean nodes = (63 + 53 + 34) / 3 = 150 / 3 = 50
ii. 30×30 mean nodes = (899 + 710 + 208) / 3 = 1817 / 3
iii. Scale = (1817/3) / 50 = 1817 / 150 = 12.113333

d. Dijkstra time scale factor
i. 8×8 mean time = (0.000109 + 0.000174 + 0.000075) / 3
ii. 30×30 mean time = (0.001187 + 0.002126 + 0.000288) / 3
iii. Scale = (0.003601/3) / (0.000358/3) = 10.0586592

That is, Dijkstra's cost grows much faster with grid size, 12.11 times in nodes expanded and 10.06 times in runtime than A*, which scales by 4.59 times in nodes and 3.46 times in runtime. That is, A* is more efficient.

Next, we aggregate pooled comparison across all scenarios and both grid sizes and then compare the totals for A* and Dijkstra.

1. Total nodes
   a. A* = 14 + 15 + 25 + 59 + 78 + 111 = 302
   b. Dijkstra = 63 + 53 + 34 + 899 + 710 + 208 = 1,967
   c. Pooled nodes speedup = 1,967 / 302 = 6.513245
   d. Pooled nodes percent reduction = 1 − 302/1,967 = 84.6755%
2. Total time
   a. A* = 0.000021 + 0.000078 + 0.000068 + 0.000108 + 0.000314 + 0.000155
      = 0.000744
   b. Dijkstra = 0.000109 + 0.000174 + 0.000075 + 0.001187 + 0.002126 + 0.000288
      = 0.003959
   c. Pooled time speedup = 0.003959 / 0.000744 = 5.32123656
   d. Pooled time percent reduction = 1 − 0.000744/0.003959 = 81.277%

A* required 84.6755% fewer total node expansions than Dijkstra (302 vs 1,967; pooled speedup = 6.513) and 81.277% less total runtime (0.000744 sec vs 0.003959 sec; pooled speedup =

5.321) across all six conditions (two grid sizes × three obstacle levels). Thus, A* offers a strong overall efficiency advantage.

CONCLUSION

On both grids and across all obstacle densities, A* achieves the same optimal paths while using less search and less time than Dijkstra. On 8×8, A* reduces node expansions by 26.47–77.78% and time by 9.33–80.73% (scenario-dependent). On 30×30, the reductions are even larger: 46.63–93.44% (nodes) and 46.18–90.90% (time). The scale-up analysis shows Dijkstra's cost grows much faster with grid size (nodes 12.11, time 10.06) than A* (nodes 4.59, time 3.46). Pooled (workload-weighted) across all six conditions (two grid sizes × three obstacle levels), A* required 302 total node expansions vs 1,967 for Dijkstra, which is ≈ 84.68% fewer (speedup = 6.513), and 0.000744 second total runtime vs 0.003959 second, which is ≈ 81.28% less (speedup = 5.321).

Therefore, A* is faster and more efficient than Dijkstra across obstacle densities from clear roads to heavily blocked cases, and its advantage widens as the map grows. These factors support that A* is a strong default for emergency vehicle routing where rapid and optimal rerouting is critical.

LIMITATION AND FUTURE IMPROVEMENT

There are a lot of limitations. Firstly, it is heuristic dependency. Because the efficiency of the algorithm heavily depends on the heuristic function, therefore inaccurate heuristic estimations can significantly reduce the algorithm performance. Secondly, due to time constraints, we simplified and used a simple n × n grid to simulate the real-world environment. However, real-world environments may present dynamic changes that the static grid simulations in this evaluation might not fully represent. Additionally, while computationally efficient in smaller scenarios, the performance advantage of A* could diminish when applied to more complex environments. Lastly, our simulations do not account for one-way streets and cycles. These directional constraints could significantly affect algorithmic efficiency and accuracy. Effective implementation of A* in such real-world scenarios requires explicit logic to handle directionality and cycle detection mechanisms.

For future improvement, we should incorporate real-time traffic data and then dynamically adjusting heuristic parameters could further enhance the A* algorithm's responsiveness and accuracy. In addition, future research might explore advanced heuristics or hybrid methods integrating elements of other pathfinding algorithms, potentially yielding further improvements in efficiency and adaptability in diverse and dynamically changing environments.

Future Research and Development Directions

Building upon the comparative analysis of A* and Dijkstra's algorithms, several promising directions for future research emerged, especially in adapting pathfinding techniques to meet the dynamic and complex needs of real-world urban navigation systems.

1. Integration with Real-Time Traffic Data

   Future implementations could incorporate real-time data feeds from GPS, IoT sensors, traffic APIs to dynamically adjust the cost function or heuristic in response to current road conditions, congestion levels, and temporary closures. Such integration would significantly enhance responsiveness and routing accuracy in time-sensitive emergency scenarios.

2. Development of Adaptive Heuristic Functions

   The performance of A* is heavily influenced by the quality and behavior of its heuristic function. Investigating adaptive or learned heuristics through machine learning or reinforcement learning could enable the algorithm to yield better estimate costs in similar environments or dynamically learn from prior routing outcomes.

3. Implementation of D* and D* Lite Algorithms

   Dynamic real-world environments are frequently changing during traversing with sudden blockages, accidents and shifting traffic flow. To handle these cases, future work should investigate the D* (Dynamic A*) and D* Lite algorithms, which were originally developed for robotic navigation. They support the efficient incremental replanning by updating only the affected areas or parts of the path rather than recomputing entire paths. They are suited for complex urban settings by effectively handling not only the sudden changes in traffic conditions but also directional constraints, cycles, and variable edge costs.

4. Scalability to Large Scale Maps

To support larger geographic areas or hierarchical map structures, research could investigate hierarchical pathfinding, graph contraction, and region abstraction techniques in combination with A*. These methods can reduce the search space and computation time in large scale urban or inter-city navigation.

5. Hybrid Algorithmic Approaches

Combining the strengths of multiple algorithms may yield better performance in complex environments. For instance:

- A* + Bellman-Ford: for environments with variable weights or temporary negative edge costs such as temporary speed boosts or emergency lanes.
- A* + Floyd-Warshall (preprocessing): to improve heuristic accuracy by referencing shortest paths between key nodes.
- Reinforcement Learning + A*: to allow learning-based pathfinding where the heuristic refines through experience in the environment.

REFERENCES

Patel, A. (2025, May 28). Introduction to A*. Amit's Thoughts on Pathfinding. Retrieved August 5, 2025, from https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968, January 1). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107. Retrieved August 5, 2025, from https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf

Biswas, D. (2021, June 27). *D* Lite, LPA*, and D*. Dibyendu Biswas on Medium*. Retrieved August 5, 2025, from https://dibyendu-biswas.medium.com/d-d-lite-lpa-e7483779a7ca

Dolgov, D., Thrun, S., Montemerlo, M., & Diebel, J. (2008). Practical Search Techniques in Path Planning for Autonomous Driving. *Ann Arbor*, 1001(48105), 18-80. Retrieved from https://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf

FRC Robotics. (2014, October 17). *D Star Algorithm* [Video]. YouTube. Retrieved August 5, 2025, from https://www.youtube.com/watch?v=e_7bSKXHvOI

GeeksforGeeks. (n.d.). *Tarjan's algorithm to find strongly connected components.* Retrieved August 5, 2025, from https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/

GeeksforGeeks. (n.d.). *Kosaraju's algorithm for strongly connected components.* Retrieved August 5, 2025, from https://www.geeksforgeeks.org/strongly-connected-components/

Hoshino, R. (2025). *Lecture 2: Informed Search Algorithms [PDF Document]*. CS 5100: Foundations of Artificial Intelligence. Northeastern University. Canvas. Retrieved January 14, 2025

Hoshino, R. (2025). *Lecture 2: Informed Search Methods [PDF Document]*. CS 5100: Foundations of Artificial Intelligence. Northeastern University. Canvas. Retrieved January 14, 2025

Hoshino, R. (2025). *Lecture 2: Problem Solving and Search [PDF Document]*. CS 5100: Foundations of Artificial Intelligence. Northeastern University. Canvas. Retrieved January 21, 2025

Hoshino, R. (2025). *Lecture 2: Solving Problems by Searching [PDF Document]*. CS 5100: Foundations of Artificial Intelligence. Northeastern University. Canvas. Retrieved January 21, 2025

Koenig, S., & Likhachev, M. (2002). *D* Lite*. Proceedings of the AAAI Conference on Artificial Intelligence, 476–483. Retrieved from https://www.aaai.org/Papers/AAAI/2002/AAAI02-072.pdf

Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed., pp. 92–112). Pearson.

Stentz, A. (1995). The focussed D* algorithm for real-time replanning. *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, 1652–1659. Retrieved from https://www.aaai.org/Papers/Workshops/1994/WS-94-03/WS94-03-007.pdf