**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.

1. ***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) <= h\*(n) where h\*(n) is the true cost of the shortest path => 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 == 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 containing nodes [n0, …, nk]. Let’s also consider there is a node nj in path P where j is between node 0 and node k that the algorithm does not expand to when it picks node t, which means the node 0 … j -1 must be visited and in the closed\_set but node j is still in the open\_set. By admissibility of the Manhattan heuristic we know f(nj) = g(nj) + h(nj) <= j + h\*(nj) = j + k - j = g\* meanwhile, at the goal node t f(nt) = g(nt) + h(nt) > g\* => f(nj) < f(t) => since A\* always chooses the open‐set node of smallest f, it would have expanded nj​ before nt. 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.

1. ***Time Complexity:***

Since we run the algorithm in a n x n grid. Grid size: V = n × n = 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.log) = 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(.log()) = O(.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: O(V.logV + E.logV) = O(5V.logV) = O(V.logV) = O(.log()) = O(logn)

1. ***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) but because we run in an n x m grid, where V = n^2 and E 2n(n-1).

Therefore, the final space complexity is:

1. **Comparison of A\* and Dijkstra (GPS)**

We conducted a comprehensive 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. The grid maps were generated randomly, representing three distinct conditions based on obstacle density:

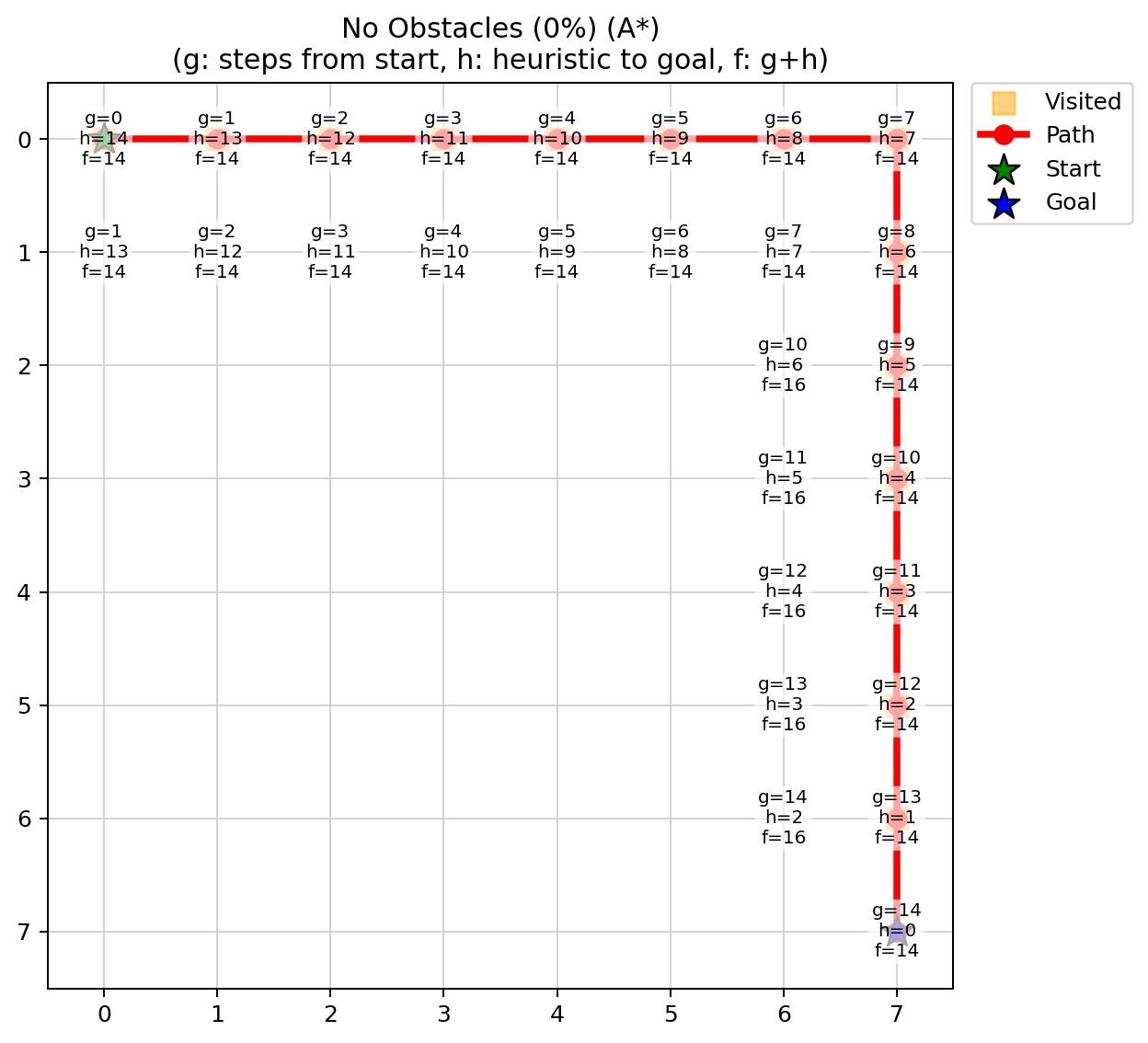
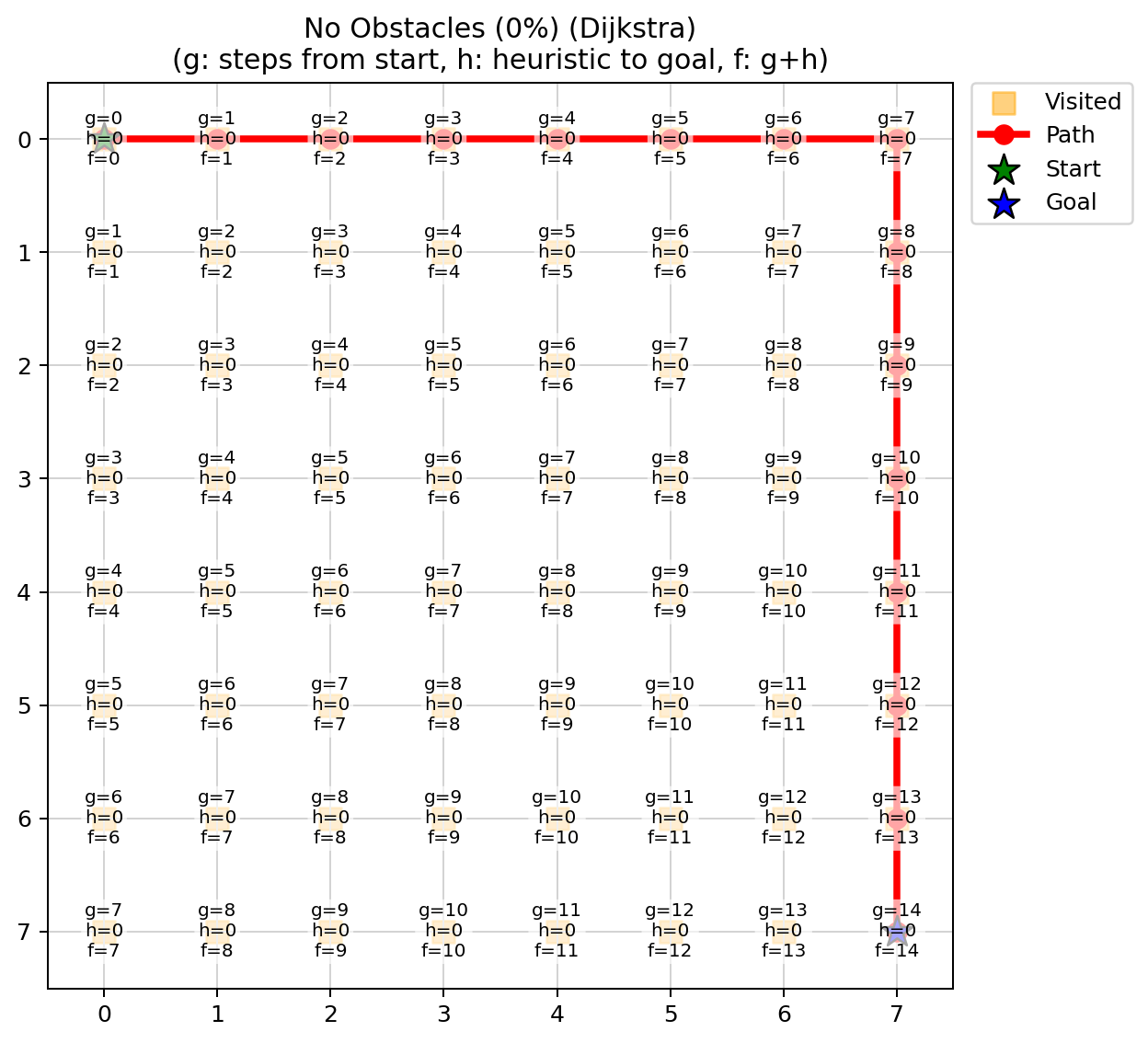
* No Obstacles (0%): An optimal and ideal scenario with no obstacles.
* Moderate Obstacles (20%): Representing moderate conditions such as minor traffic disruptions or temporary blockages.
* High Obstacles (50%): Simulating severe conditions akin to widespread disruptions or emergencies, such as natural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

For each scenario, the following metrics were precisely measured and recorded:

* Computation Time: The duration taken by each algorithm to compute the optimal path.
* Expanded Nodes: The number of nodes the algorithm explored during pathfinding, indicating computational efficiency.
* Grid Composition: Detailed statistics regarding obstacle percentage and free cell availability.

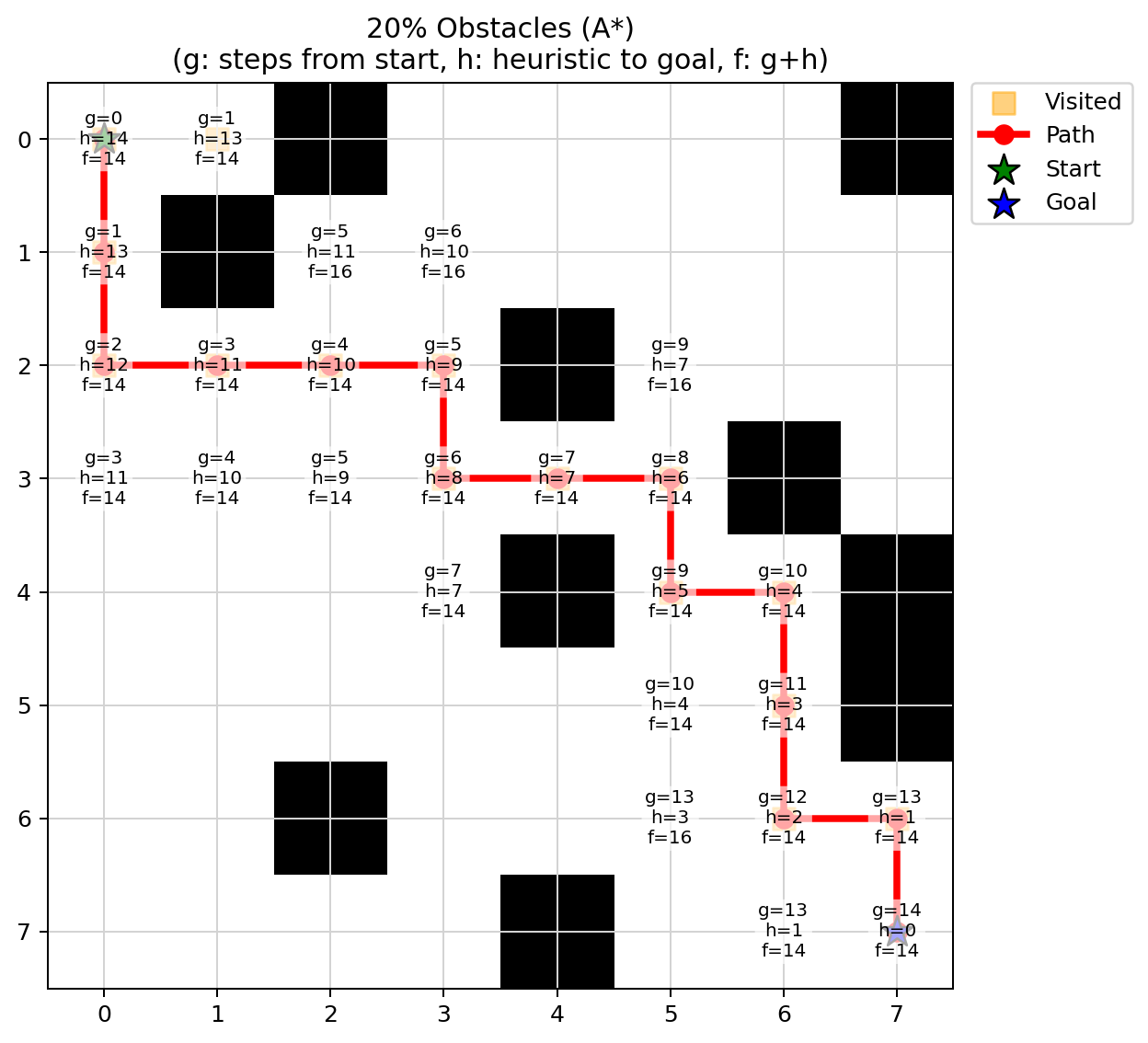
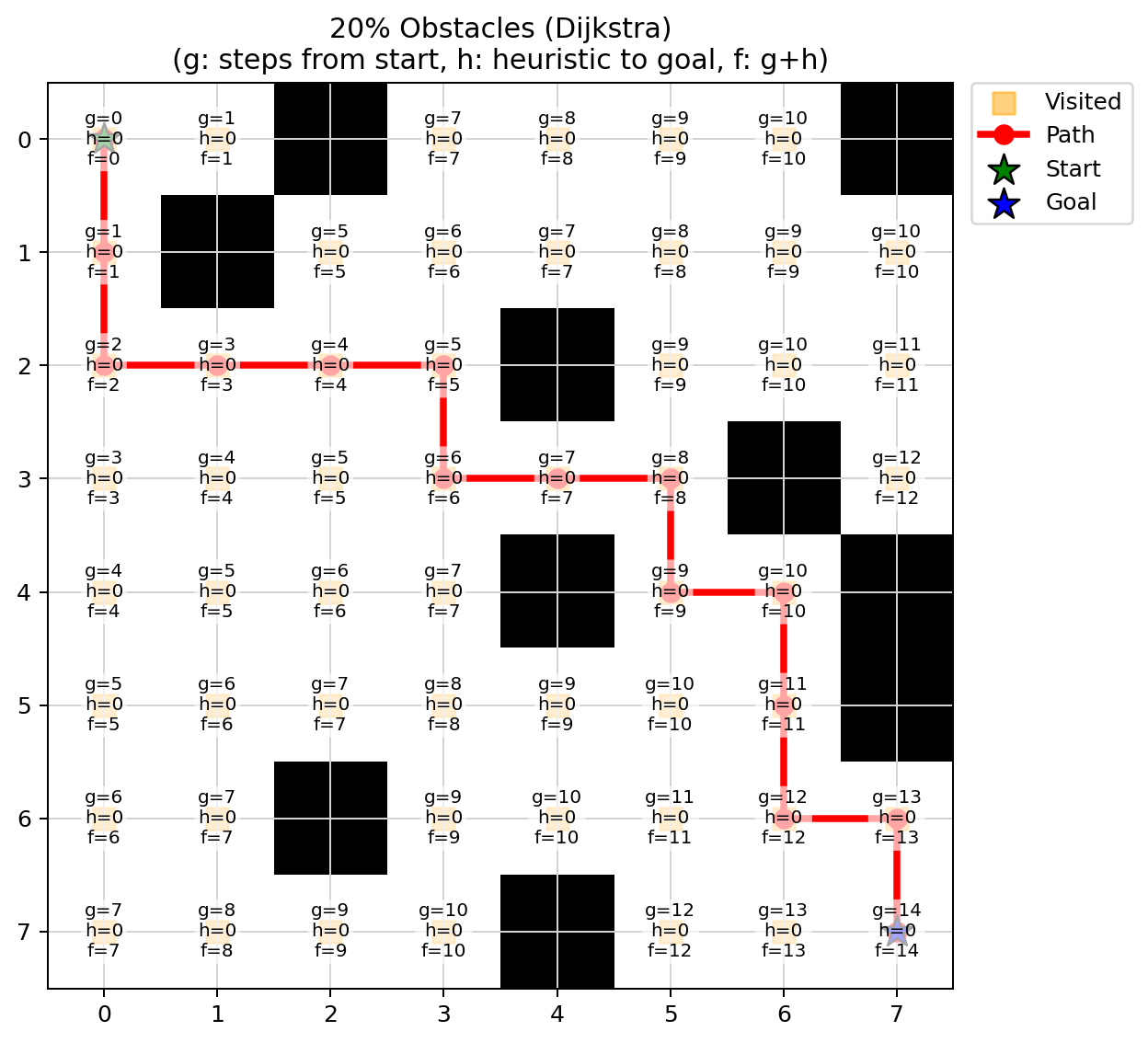
**Scenario Evaluations:**

1. No Obstacles (0%) Scenarios:
   1. Grid Composition: 0% obstacles, 100% free cells.
   2. A\* Algorithm Performance: Expanded 14 nodes with a computation time of 0.000021 seconds, indicating near-instantaneous computation and exceptional efficiency.
   3. 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).

Detailed data: <https://sxando.github.io/Emergency-Vehicle-Routing-Data/>

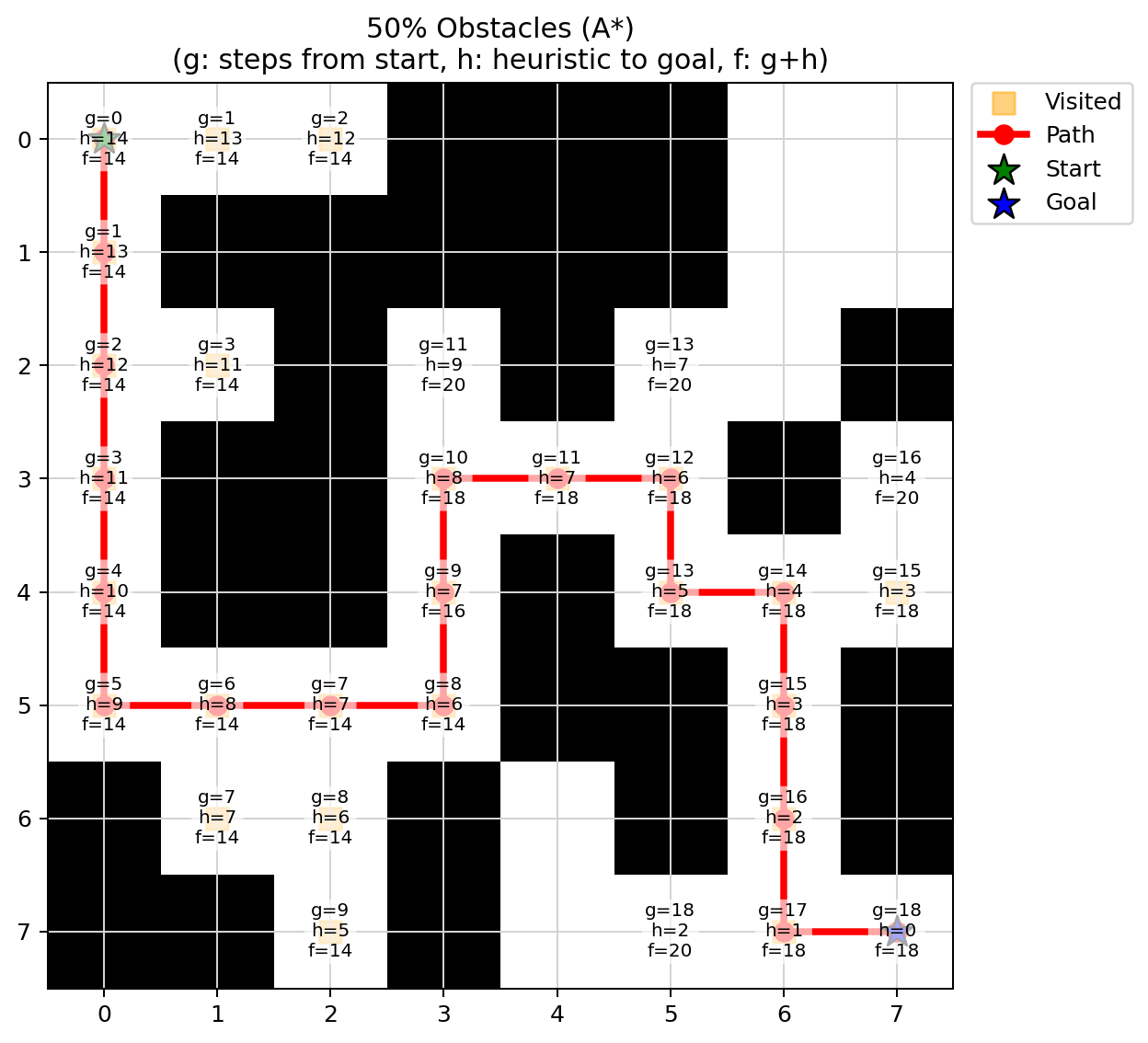
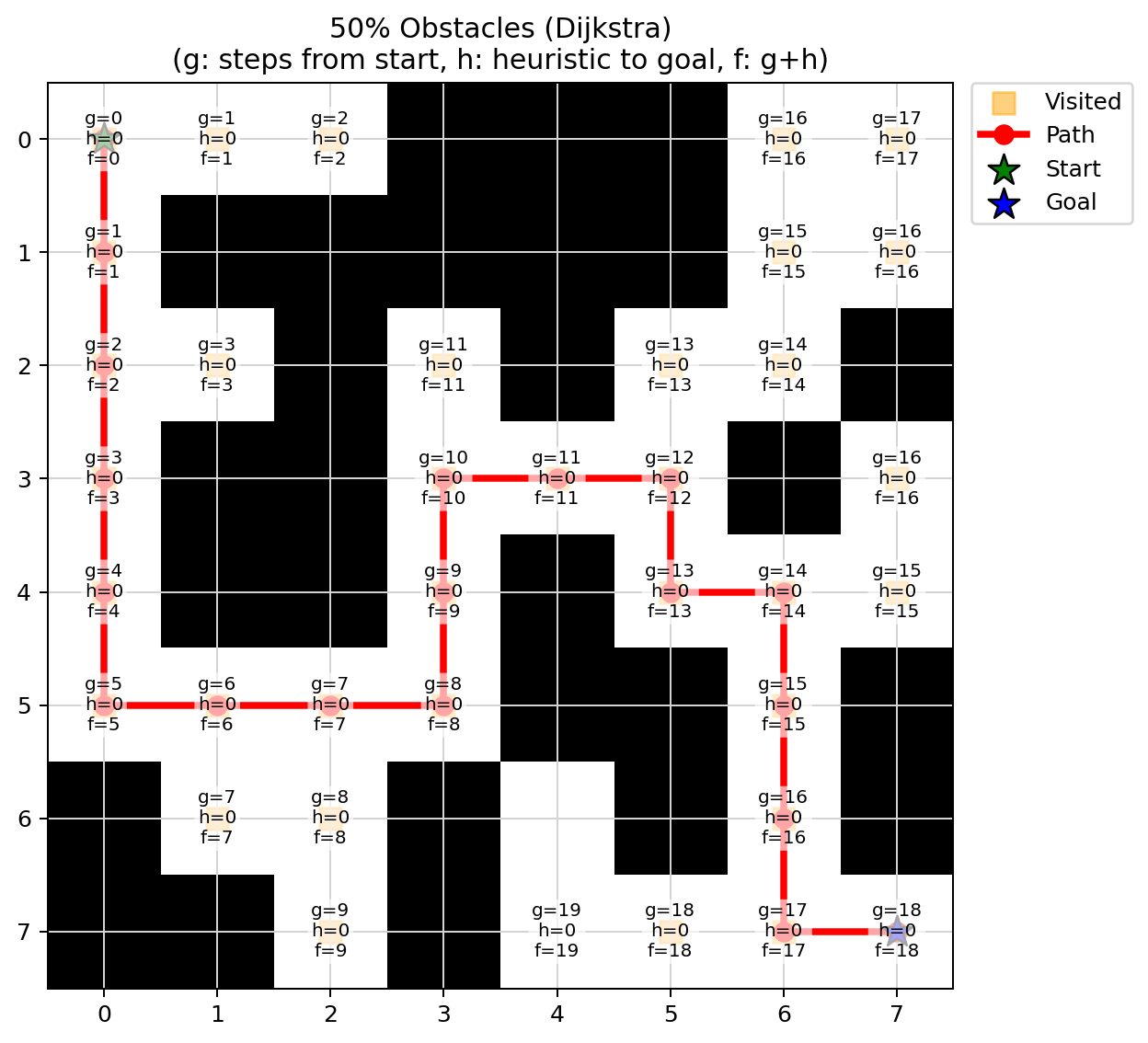
1. Moderate Obstacles (20%) Scenario:
   1. Grid Composition: Approximately 15.6% obstacles, 84.4% free cells, effectively representing moderate traffic disruption scenarios.
   2. A\* Algorithm Performance: Expanded 15 nodes with a computation time of 0.000078 seconds. Despite encountering minor obstacles, A\* effectively recalculated paths, demonstrating robust adaptability and maintaining high computational efficiency.
   3. 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.

Detailed data: <https://sxando.github.io/Emergency-Vehicle-Routing-Data/>

3. High Obstacles (50%) Scenario:

1. Grid Composition: 42.2% obstacles, 57.8% free cells, simulating severe traffic disruption scenarios.
2. 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.
3. 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.

Detailed data: <https://sxando.github.io/Emergency-Vehicle-Routing-Data/>

**Data Comparison Table:**

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

**Conclusion**

The results confirm that A\* with an admissible Manhattan heuristic delivers optimal routes while guiding the search far more efficiently than Dijkstra’s algorithm. Across our three test scenarios, A\* consistently expanded fewer nodes and outperformed Dijkstra in computation time: 14 vs 63 nodes (0.000021 sec vs 0.000109 sec) with no obstacles, 15 vs 53 nodes (0.000078 sec vs 0.000174 sec) under moderate obstacles, and 25 vs 34 nodes (0.000068 sec vs 0.000075 sec) in heavily disrupted environments. While the performance gap narrows as blockage density increases, A\* never underperformed Dijkstra and always returned the optimal path.

For emergency vehicle routing, where conditions change rapidly and every second matters, these findings indicate that A\* is a strong default choice. Its heuristic guidance reduces unnecessary exploration, enabling rapid rerouting when roads close or congestion emerges, and its optimality guarantees that the selected route is the shortest possible given current information.

**☆ Detailed Data**: Includes the large environment setting simulation (30×30 grid)

<https://sxando.github.io/Emergency-Vehicle-Routing-Data/>

**Simulation Setup: Map and Obstacles**

1. **Map and Obstacle Generation**

The map creation procedure is designed to produce randomized yet guaranteed-solvable grid instances for evaluating the performance of the A\* and Dijkstra’s algorithms under varying obstacle densities (0%, 20%, and 50%). The method ensures that each generated map reflects stochastic variability in obstacle placement while maintaining global connectivity between the start and goal nodes.

1. Random Sampling of Obstacles
   1. Obstacles are assigned independently to cells. Blocked (1) with probability p = prob\_block and Open (0) with probability 1 - p.

|  |
| --- |
| Python code:  grid = np.random.choice([0, 1], size=(N, N), p=[1 - prob\_block, prob\_block]) |

1. Endpoint Accessibility

The start and goal cells are set to open. To prevent complete isolation, each endpoint is guaranteed to have at least one open orthogonal neighbor (up, down, left, or right). If all four neighboring cells are blocked, the algorithm randomly selects one neighbor and sets it to open. This adjustment ensures endpoint accessibility without directional bias.

1. Solvability Check

The generated grid is passed to the A\* pathfinding algorithm (using the Manhattan distance heuristic) to verify that a valid path exists between start and goal. This step makes sure there is a path through the entire map, not just around the start and goal.

1. Iteration Until Solvable (Max 1000 attempts – increase the value for larger grid)

If A\* fails to find a path, the generator discards the current grid and repeats the process. This loop continues until a solvable instance is obtained or a maximum of 1000 attempts is reached (for 8×8 grid).

**2. Obstacle Handling**

When expanding the current node, the algorithm calculates the neighboring cell for each possible direction (up, down, left, and right). It then checks if the neighbor is within the grid boundaries and not an obstacle. If the neighbor is either out of bounds or blocked (has a value of 1), the algorithm skips it and does not consider it as part of the path. In other words, the algorithm never tries to move through obstacles or outside the grid. If the algorithm cannot find a path from start to goal because obstacles block all possible routes, it stops and returns no solution.

|  |
| --- |
| Python code: DIRS = [(-1, 0), (1, 0), (0, -1), (0, 1)] # up, down, left, right  for dx, dy in DIRS:  neighbor = (current\_node[0] + dx, current\_node[1] + dy)  # Check if neighbor is within grid bounds  if not (0 <= neighbor[0] < N and 0 <= neighbor[1] < N):  continue # Skip if out of bounds  # Check if neighbor is an obstacle  if grid[neighbor] == 1:  continue # Skip if blocked |

**Limitation and Potential 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. Simulation of Realistic Urban Conditions

While grid-based models provide clarity and simplicity, future studies should simulate more realistic urban networks including variable street widths, traffic lights, turn penalties, and time-dependent travel costs. These simulations would yield more practical insights and bridge the gap between theoretical efficiency and real-world performance.

1. Deployment in Edge Devices or Embedded Systems

With the growing use of onboard systems in emergency vehicles, future development should focus on lightweight, optimized implementations of these algorithms that can run on embedded or resource-constrained systems that deliver fast and reliable performance without depending on cloud infrastructure.

**Contribution**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Standard A\* | Modified A\* | Correctness | Complexity | GPS comparison | Report |
| Aayushi |  |  |  |  |  |  |
| Divit | 50% | 50% |  |  |  |  |
| Sean | 50% | 50% |  |  |  |  |
| Ngoc |  |  |  |  |  |  |
| Zuzu |  |  |  |  |  |  |

**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

**Implementation (A\* algorithm) - Concise Version**

The city is modeled as an N × N grid (N = 8) with 4-way movement and unit traversal cost. A\* searches for a shortest path from start to goal while minimizing explored nodes (states).

1. Heuristic

Manhattan distance (h(n) = ∣ x1−x2 ∣ + ∣ y1 − y2 ∣) is used. It is admissible and consistent on the grid. So, the first time the goal is selected the path is optimal (if one exists).

1. State and structures

open set (priority queue) ordered by f(n)=g(n) + h(n)

closed set (expanded nodes)

g\_score map (best known cost from start)

parent map (come\_from) for path reconstruction.

1. A\* Search Procedure

Initialize g(s)=0 and push s with f(s) = h(s). Repeatedly pop the node with the smallest f.

If it is the goal, reconstruct the path via parents. Otherwise, consider each traversable neighbor, relax its cost (update g, parent, and f if improved), and insert it into the open set. If the open set empties before reaching t, report “no path”.

1. Output

A path from start to goal (sequence of cells) or failure if no path exists.

1. Scope across obstacle levels

The search logic is identical for 0%, 20%, and 50% obstacle densities. Only the input maps are different.

**Implementation (A\* algorithm)**

This project implements the A\* pathfinding algorithm on an N × N grid, and evaluates its performance under varying obstacle densities (0%, 20% and 50%). The implementation visualizes the optimal path on the map, along with g (cumulative cost from the start node to current node), h(heuristic), and f (total estimate cost) values for both the actual path and all considered nodes. Nodes that are visited (expanded) during the search are highlighted in pale orange. Nodes that are not considered by the algorithm (nodes never added to the open\_set during the search) remain blank with no associated g, h, or f values. The algorithm doesn’t evaluate them as potential steps on any path from the start to the goal.

The implementation measures the actual runtime of searching the optimal path in each scenario. All textual outputs are printed to the console, and the map images are opened for presenting the result. After each run, the textual results (as .txt files) and the maps (as .png images) are automatically saved to the same directory where the Python script is located.

1. Grid representation
   1. The environment is modeled as an N×N grid.
   2. Each cell may be open (0) or blocked (1).
2. Node expansion and movement
   1. Each node corresponds to a cell (row, col).
   2. Four-way movement (up, down, left, right) is allowed.
   3. Only unblocked, in-bounds neighbors are considered.
3. Heuristic function
   1. Manhattan distance is used as the heuristic (h(n)), which is both admissible and consistent for grid worlds.
4. Priority queue (open set)
   1. The open set is a min-heap storing nodes to be expanded.
   2. Each entry contains:
      1. The total estimated cost f(n) = g(n) + h(n)
      2. The heuristic h(n) (used tie-breaking method)
      3. The node’s coordinates
5. Cost tracking
   1. g\_score (dictionary) records the minimum cost from the start to each node.
   2. The algorithm only updates a node’s cost and parent if a lower-cost path is found.
6. Parent pointer map (came\_from)
   1. For each node, store the parent node from which it was reached.
   2. This method efficiently reconstructs the shortest path from goal to start after the successful searching.
7. Closed set
   1. A set tracks all nodes already expanded to prevent reprocessing.
8. Path reconstruction
   1. Once the goal is reached, the path is reconstructed by parent pointers from the goal to start (backtracking), then reversing the sequence.
9. Failure handling
   1. if the open\_set is exhausted before the goal is reached (no path exists due to obstacles), the algorithm reports failure and returns no path.

* The search logic is the same for various obstacle densities (0%, 20%, and 50%). Only the input maps are different.