Incremental Replanning for Robot Navigation: A D* Lite Implementation and Benchmark, with Multi-Agent Extension

Description: Most path planners assume a robot knows its environment before it starts moving. Real robots rarely get that luxury: a delivery robot's map doesn't show that a street was closed for construction this morning, a warehouse robot doesn't know a pallet was moved until it gets there, a search-and-rescue robot exploring a collapsed structure only learns what's passable as it goes. The naive response, rerunning A* from scratch every time the map turns out to be wrong, works but throws away almost everything the robot already knew about the rest of the map. This project implements D* Lite, the incremental replanning algorithm that instead repairs only the part of a previous solution actually affected by new information, and rigorously benchmarks how much that repair actually saves versus resolving from zero.

What makes it technically substantive, not just a working demo:

- It's a real algorithm with real history, not an invented exercise. D* and its incremental successors were developed for DARPA's Unmanned Ground Vehicle program, ground robots navigating terrain that wasn't fully known in advance, and the same family of algorithms (Field D*) has run on real outdoor field robots since. This is a genuine, applied lineage in mobile robotics, not a textbook toy.
- The core mechanism is a real dynamic-programming idea applied incrementally: maintaining two values per node (`g`, the current best-known cost, and `rhs`, a one-step lookahead), and propagating updates only outward from wherever the map actually changed, rather than recomputing everything from scratch.
- The deliverable is a real empirical result, not just "it runs": a benchmark comparing D* Lite's repair cost against A*-from-scratch across map changes of varying size and locality, with an honest discussion of where the advantage holds and where it shrinks.
- It connects legitimately to your own background: the same "don't resolve what you've already solved, just correct what changed" logic underlies incremental state estimation, which is what you actually worked on modeling IMU sensor noise and bias drift.

Real-world framing: picture a ground robot operating in a human environment where the map is a best guess, not a guarantee, a delivery robot, a warehouse robot, an indoor service robot. It plans a route, discovers midway that the route doesn't match reality, and has to react fast without discarding everything it already knew about the parts of the map that are still accurate. That's the exact problem this project solves and benchmarks.

Stretch extension: once single-robot D* Lite is working, extend to 2-3 robots that must avoid colliding with each other, using each robot's own D* Lite planner plus a lightweight Conflict-Based Search-style constraint mechanism, reusing your existing "something changed, repair the plan" machinery to also mean "a conflict appeared, repair the plan." This mirrors real warehouse fleet coordination, where multiple robots replan around each other constantly, not just around a static map.

Research done:

Tools used:

- **Python** (popular programming language)
- **Numpy**
- **Pandas**
- Json
- CSV
- Self-coded binary priority queues for a* (lazy deletion) and d* lite (with a dictionary to keep track of the index of all keys)

Datasets used:

Math / Algorithms / Logic used:

- https://cdn.aaai.org/AAAI/2002/AAAI02-072.pdf
  A 2002 paper by Sven Koenig (Georgia Institute of Technology) and Maxim Likhachev (Carnegie Mellon University) about D* Lite and its optimized version beginning from A* and Lifelong Planning A* (LPA*).

  A*: The classic, single-agent pathfinding algorithm. It uses an evaluation function f(n) = g(n) + h(n), combining actual distance traveled from the start (g) with an estimated heuristic distance to the goal (h), to explore the grid and find an optimal path efficiently.

  Lifelong Planning A* (LPA*): An incremental version of A* designed for dynamic graphs with a fixed start and goal. It maintains two cost estimates per node: g(n) (current shortest path estimate) and rhs(n) (one-step lookahead cost). When edge costs on the map change (e.g., an obstacle appears), it re-evaluates only the nodes made locally inconsistent g(n) != rhs(n)), reusing previous search work rather than planning from scratch. It is best for scenarios where **the start and goal stay in place**, but terrain costs or obstacles dynamically change over time.

  D* Lite: Similar to LPA*, but it searches **backward from the goal** to the robot's current position. D* Lite updates local inconsistencies (g != rhs) without invalidating the

goal's reference frame. It works for autonomous navigation and robot mapping, where a vehicle traverses an unknown or changing map toward a **fixed goal (no fixed starting point).**

The Why: Because D* Lite roots its g and rhs values at the goal node, the robot's position (s_start) is free to move without invalidating the search tree. In contrast, LPA* roots its values at the start node, so changing the start position forces significant re-evaluation of g and rhs values throughout the graph.

D* Lite optimized: This has the same logic as D* Lite, but it solves two main bottlenecks (in my opinion):

1. **Eliminating Redundant Queue Searches (In Heap Lookups)**
   In the unoptimized version, UpdateVertex(u) is treated as a catch-all "black box." Every time it runs, it checks whether u is in the priority queue.

   In binary heap implementations, checking set/queue membership can be an O(N) operation unless you maintain an auxiliary hash map. By moving the logic inline, the optimized version only interacts with the queue when it *knows* a node needs to be inserted, deleted, or re-prioritized.

2. **Avoiding Unnecessary Neighbor Sweeps**
   In the unoptimized version, calling UpdateVertex() often causes the algorithm to indiscriminately inspect all neighbors of a node just in case their rhs-values changed.

   In the optimized version:
   - **When an edge cost decreases:** The algorithm knows *only* that specific edge could have made u cheaper, so it updates rhs(u) using just that single neighbor rather than re-scanning all 8 directions.
   - **When a node becomes consistent:** It doesn't waste time re-evaluating neighbors that couldn't possibly be affected by the change.

- https://scispace.com/pdf/multi-agent-pathfinding-definitions-variants-and-benchmarks-4fyvj1uy3h.pdf
  A 2019 paper by professors and PhD students from five universities listed below:
  Professors / Senior Faculty:

- **Sven Koenig** (Professor at USC)
- **Ariel Felner** (Professor at Ben-Gurion University)
- **Nathan R. Sturtevant** (Professor at University of Alberta)
- **Roni Stern** (Professor at Ben-Gurion University)
- **Roman Barták** (Professor at Charles University)
- **T. K. Satish Kumar** (Research Associate Professor / Senior Research Scientist at USC)

PhD Students / Postdoctoral Researchers (at the time of publication):

- **Jiaoyang Li** (PhD student at USC under Sven Koenig)
- **Hang Ma** (PhD student / Postdoc at USC)
- **Dor Atzmon** (PhD student at Ben-Gurion University)
- **Eli Boyarski** (PhD student at Ben-Gurion University)
- **Liron Cohen** (Postdoctoral researcher / PhD student at USC)
- **Thayne T. Walker** (PhD student at University of Denver)

The paper provides a unified terminology and taxonomy for Multi-Agent Pathfinding (MAPF) to address inconsistent definitions, implicit assumptions, and non-standardized baselines across existing literature. This paper has the goal to establish a common ground for researchers and practitioners.

Core idea: Each agent at each time step either moves to an adjacent vertex or waits.

We only need to care about:

- A vertex conflict: when two agents are planned to occupy the same vertex at the same time step
- An edge/swapping conflict: when two agents are planned to traverse the same edge at the same time step in the same/opposite direction.
- Note (not in paper): While there are other conflicts, they are refinements that are not necessary for 2-3 robots.

Common functions:

- Makespan: the number of time steps required for all agents to reach their target.
- Sum of costs (also known as flowtime): the sum of time steps required by each agent to reach its target.

Challenge: Implementing the “stop” for a robot to prevent conflicts for mapf.

Since the d* lite algorithm has no concept of time, it was difficult for me to think of a way to implement it. Unlike the implementation of an algorithm, I did not have a guide to follow. So I thought of two possibilities:

1. At the step before the conflict, add the robot’s current position as a possible next step, which can have the same effect as a “stop”.

   Problem: 1. D* lite has no concept of time. 2. How would D* lite know to separate the global terrain changes with robot-dependent blockages due to robot conflicts?

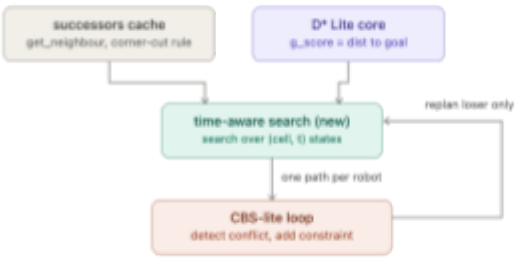
2. Implement the time-step outside of the d* lite function. The time-step aware layer would have (cell, timestep) as a pair.

   Problem: how does this layer know what is the optimal step to take?

Solution: Create a time-aware search that wraps around the d* lite algorithm.

D* lite’s purpose in this version is just to compute distances (rhs and g), not the path. Use all the functions used by D* lite in the new time aware function.

The time-aware search wrapper is finding the actual path, with priority queues, cost tracking, etc. However, since the data collected (from week 5) shown that A* is faster and checks a similar number of nodes than d* lite, especially for single cell changes, therefore, the time-aware search wrapper used a A* like approach rather than d* lite.



Caption: AI generated diagram of the overall structure and relationship between the d* lite algorithm and time-aware wrapper.

Timeline

Week 1: Foundations and baseline

- Read the D* Lite paper (Koenig & Likhachev, 2002) closely enough to understand the g/rhs bookkeeping and the ComputeShortestPath/UpdateVertex procedures, not just skim the abstract.
- Build your environment representation: a 2D grid with traversable/blocked cells, using NumPy for the grid and Matplotlib for visualization.
- Implement plain A* first, as your known-correct reference. You'll need this both as a sanity check and as the "resolve from scratch" baseline for your benchmark later.
- Deliverable: A* running correctly on a static grid, with a plotted path.

Implementation:

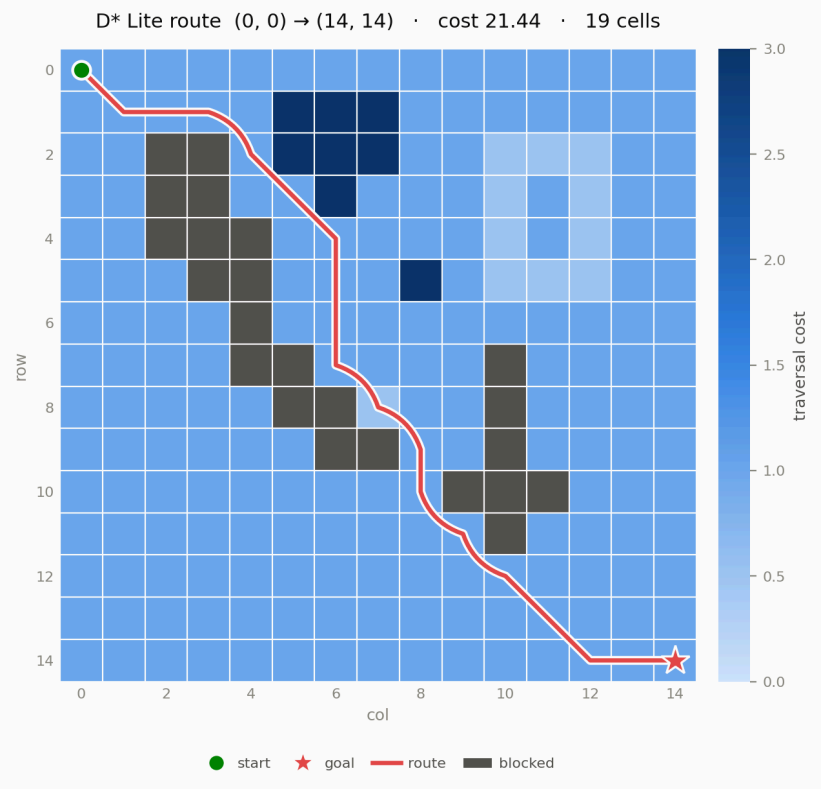
- Used **Numpy**
- Used priority queue for A* (**traditional heapq with lazy deletion** to consider for the decreasing key)
- Have terrains with different difficulties (0.5, 1.0, 3.0)
    - Problem: While this doesn’t change the difficulty of calculating g(n), it does change the difficulty of h(n) for f(n) = g(n) + h(n)
    - Solution: h(n) must never overestimate the true remaining cost, or else A* isn't guaranteed to find the optimal path. Therefore, to guarantee h(n) <= true cost, all cells are assumed to be the min cost of 0.5.
- Claude code to make visualizations

Week 2: D Lite core, static case*

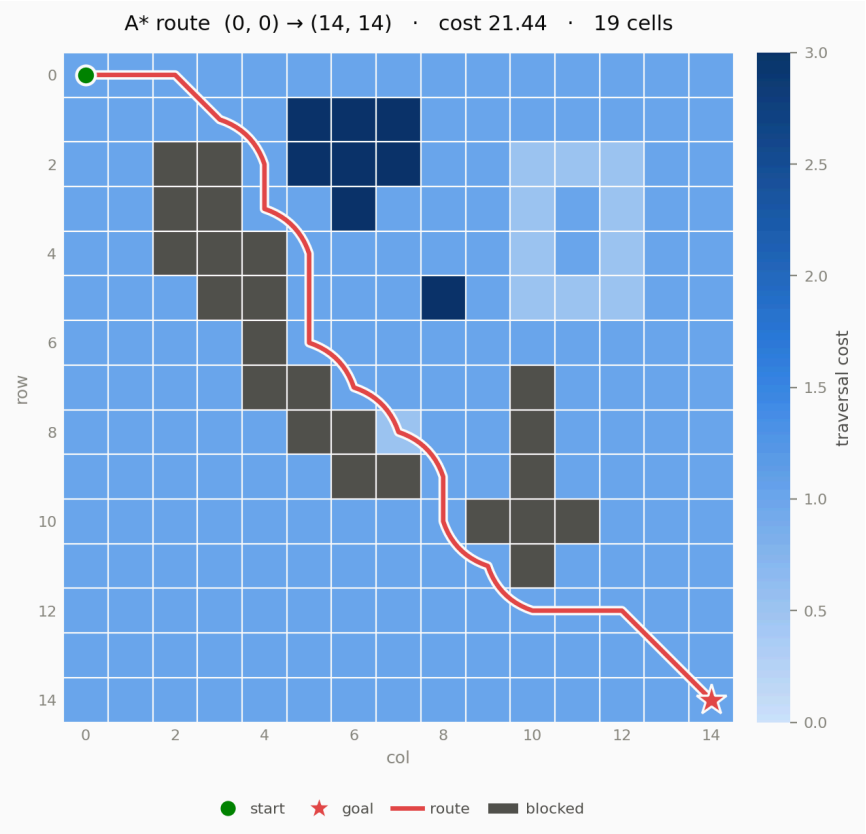
- Implement the D* Lite data structures: the priority queue keyed by the algorithm's key function, and the g/rhs values per node.
- Implement ComputeShortestPath and get it working on a fully known, unchanging map.
- Sanity check: on a static map with no changes, D* Lite must produce the same path as A*. If it doesn't, something in your implementation is wrong, don't move on until this matches.
- Deliverable: D* Lite matching A*'s output on several static test maps.

Implementation:

- Similar file structure and code from week 1 (just an altered priority queue and path-finding algorithm)
- Upon final comparison of the two paths generated (as a comparison), they were sightly different, yet both correct



D* Lite Computated Path



A* Computed Path

Good news: Both routes report cost 21.44 and 19 cells (which is the most important part of the check)
This means there are multiple paths of the sane cost, and A* and D* lite algorithms picked different, but tied options. It is just a consequence of *how* each algorithm happens to break ties.

Week 3: Incremental updates, the actual point of the project

- Implement UpdateVertex and the edge-cost-change handling that lets the algorithm react to a map change without a full recompute.
- Script a sequence of map changes (an obstacle appears, an obstacle disappears, a corridor turns out to be blocked) and confirm the robot replans correctly and only reprocesses nodes near the change.
- Add a visualization showing which nodes get reprocessed after each change, this becomes a genuinely useful debugging tool and a good figure for your write-up.
- Deliverable: a working demo of the robot navigating a map that changes mid-traversal, with visible incremental repair.

Implementation:

- Update the terrain with terrain_updates.json, which is loaded into the pre-existing d_star_lite.py file from week 2
- Updated the animation to include grid changes.

Week 4: The benchmark

- Build a harness that runs the same sequence of map changes through both D* Lite (incremental) and A* (from scratch each time), recording nodes expanded and wall-clock time per replan.
- Run this across map changes of varying size and locality: a single blocked cell, a small blocked region, a large blocked region. You want to show not just "D* Lite wins" but where and why the margin changes.
- Deliverable: raw benchmark data and a first-pass plot of nodes-expanded/time vs. change size, for both methods.

Implementation:

- Added to existing documents, specifically:
    - updating_a_star() function
    - Time_elapsed, node_visited, node_expanded to compare the updating_a_star() and d_star_lite() algorithms

Week 5: Analysis and core write-up

- Turn Week 4's data into your headline figure: a clean comparison plot with a clear explanation of why D* Lite's advantage shrinks as changes get larger and more global (this nuance is what separates a real result from a one-sided claim).
- Write the core sections of your report: problem statement, method, results, so far. This is your safety-net milestone, if nothing else gets built, this alone is a complete, legitimate project.
- Deliverable: a solid draft write-up and the core repo in a presentable state.

Implementation:

- Data collection:
    - **A harness script** that runs both algorithms over the *same* sequence of scripted map changes and logs the results — not something re-run by hand each time.
    - **Multiple change scenarios of varying size and locality** — single blocked cell, small region, large region to show *where the advantage holds and where it shrinks*, not just one favorable case.
    - **The grid-size variation experiment**
    - **Multiple trials per scenario**, averaged — wall-clock time is noisy run to run
    - **Raw data saved somewhere reusable**: CSV

- Asked Claude to generate all benchmark scenarios, including:
    - 8 different terrains (15 x 15, 30 x 30, 50 x 50, 80 x 80, 150 x 150, 300 x 300, 500 x 500, 1000 x 1000)
    - 4 type of cases for each terrain
        - Changes in a large region
        - Changes in a small region
        - Changing a single cell
        - Reopening a shortcut

- Data Tables

Performance Summary by Grid Size

Terrain size	A* Time (s)	D* Time (s)	Speed Ratio	A* Eval.	D* Eval.	Eval. Reduction
15	0.000562064550103925	0.0026271853499565	4.67x	254	188	25.94%
30	0.002762388359697	0.010285561720083899	3.72x	1178	687	41.68%
50	0.009102256720216199	0.030683258359931596	3.37x	3406	1982	41.80%
80	0.029939788319956222	0.08004401995989605	3.82x	7772	4912	36.80%
150	0.07060087168007154	0.280658346680284	3.98x	26573	17112	35.60%
300	0.310607308360195	1.1628495366399876	3.74x	106994	67042	37.34%
500	0.9403066833199409	3.34555470331994	3.56x	298128	189308	36.50%
1000	3.9104838216801	13.516429623279981	3.46x	1197215	752573	37.14%

==========

Performance Summary by Change Type

Change type	A* Time (s)	D* Time (s)	Speed Ratio	A* Eval.	D* Eval.	Eval. Reduction
large_region	0.8039076625499547	2.8056370802249146	3.49x	252446	157110	37.76%
reopen_shortcut	0.8076803833251688	2.79706800619997	3.46x	251592	157085	37.56%
single_cell	0.46375788002672674	1.6634511772133798	3.59x	144044	93235	35.27%
small_region	0.8096489103749263	2.7962032052000723	3.45x	251798	157092	37.61%

Definitions:
Start is the moment the algorithm is introduced to the terrain. **Finish** is completing the path from start to end node considering changes in terrain.
Evaluations are the nodes that are pushed and evaluated by the algorithms from start to finish.
Runtime is the amount of time each algorithm took from start to finish.

- Data Analysis

Search Efficiency Paradox (Evaluations vs. Real Runtime)

- **D* Lite is ~37% more efficient in search space traversal:** Across all map sizes, D* Lite consistently requires fewer node evaluations than A*. However, the gap between D* lite and A* reduces as grid size increases.
- **A* is ~3.5×-4.5× faster in wall-clock time:** Despite evaluating more nodes, A* completes runs significantly faster than D* Lite. However, the gap between D* lite and A* increases as grid size increase.
- This suggested that D* lite may be increasingly more beneficial than A* as grid size increases.

Constant-Factor Overhead Bottleneck

- The runtime discrepancy is driven by **D* Lite's heavy key and queue operations**. Maintaining rhs-values, calculating complex 2D keys (k(s) = [k_1, k_2]), and handling a higher volume of node discoveries/heap pushes (~1.5×-2.5× more than A*) creates a substantial per-node processing overhead in Python.

Invariance to Change Types

- Within any given grid size, performance metrics vary by **less than 2%** across terrain change scenarios (`single_cell`, `small_region`, `large_region`, `reopen_shortcut`). In this benchmark harness, the initial full-grid search dominates the compute budget relative to incremental repair.

Additional break (to optimize and simulate more realistic scenarios after testing):

- Observation: D* lite is always slower than A*, even though they have less nodes visited.
  Solution: Implemented the optimized version of the d* lite from figure 4 of the research paper.

- Observation: The robot would go diagonally through two blocked sections, which would be unrealistic for robots with any width.
  Solution: Added new corner cutting rules for all three algorithms such that the robot cannot squeeze diagonally through two blocked sections.

More comparisons:

Execution Time by Terrain Size

Terrain size	A* Time (s)	D* Lite Time (s)	D* Lite Optimized Time (s)	Speed Ratio (A* vs D* Lite Opt)	Speed Ratio (D* Lite vs D* Lite Opt)
15	0.0007038062511128	0.00320535005012055	0.002570274950267	3.65x	1.25x
30	0.00339548663992896	0.0161167849990306	0.012106535001075761	3.57x	1.33x
50	0.009689906678977 38	0.036873975039925 36	0.030423213320900644	3.14x	1.21x
80	0.0228825916000641 38	0.09449864499911195	0.07565238835930353	3.31x	1.25x
150	0.081408758360194	0.3402607032406376	0.27534244999988	3.38x	1.24x
300	0.345946083399467 14	1.3715384100400843	1.0989890199608636	3.18x	1.25x
500	1.158645894960500 3	3.9297402949997924	3.164062245039968	2.73x	1.24x
1000	5.045795553319622	15.79058879663935	13.524837584960625	2.68x	1.17x

==========

Execution Time by Change Type

Change type	A* Time (s)	D* Lite Time (s)	D* Lite Optimized Time (s)	Speed Ratio (A* vs D* Lite Opt)	Speed Ratio (D* Lite vs D* Lite Opt)
large_region	0.9219253468745592	3.381460143749427	2.7480134041747077	2.98x	1.23x
reopen_shortcut	0.9528681052004685	3.3587743052245065	2.736004741650686	2.87x	1.23x
single_cell	0.7370249116267467	1.8211831044399878	1.6637357688803849	2.26x	1.09x
small_region	0.9109894135996001	3.333910911449857	2.761145822900653 4	3.03x	1.21x

Data analysis:

- While D* lite optimized is still slower than A*, D* lite optimized is around 1.25 times faster than D* lite consistently across all terrain sizes.
- The results also suggests that D* lite optimized shows a greater advantage against D* lite for large region changes in comparison to small region changes.

Week 6: MAPF stretch extension

- Extend to 2-3 robots, each with its own D* Lite planner.
- Implement basic conflict detection (two robots planned to occupy the same cell at the same time step) and a Conflict-Based-Search-style fix: add a constraint forbidding that robot from that cell at that time, then have only that robot replan.
- Keep this honestly scoped: a small, clearly-demonstrated proof of concept is the goal, not a scaled system. If this starts eating into Week 7, stop and fall back to documenting it as "here's a working small-scale version, here's what would need to change to scale it," which is a legitimate and honest way to close this section.
- Deliverable: a demo of 2-3 robots replanning around both map changes and each other.

Implementation:

- The algorithmic idea (Conflict-Based Search): plan every robot independently first, ignoring each other. Simulate the paths together. Find the *first* timestep where two robots collide. Pick one of the two, add a constraint and replan *only that robot* under the new constraint. Repeat.

  Note: a constraint is agent- and time-scoped — NOT the same as a terrain change. Terrain updates (real obstacles) block a cell for every robot at every future timestep; constraints block one robot at one timestep only.

- The robots only physically move when the final, conflict-free path is found, where the robots can stop/wait or move at each step.

- conflict_search() returns the first conflict found (or None), not all conflicts at once. On a conflict, add one entry to the constraints list for the *chosen robot*, then replan that robot and re-run conflict_search() from the robot's original start. The *chosen robot* is the first robot (the earlier robot initialized from json) by default. If that choice makes destination unreachable, the *chosen robot* is set as the second robot. If both does not work, then the destination is marked as unreachable.