HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Path Planning in Dynamic Environments

Students:

Lương Thái Khang- 20224866

Ong Xuân Sơn - 2022

Hanoi, January 2026

ABSTRACT

Table of Content

1 INTRODUCTION

1.1 Motivation

Path planning is a core capability in mobile robotics and autonomous systems: an agent must compute a collision-free route from a start state to a goal state while respecting motion constraints and environmental obstacles. In many realistic applications (e.g., service robots in human environments, warehouse automation, and autonomous driving), the environment is *dynamic*: obstacles move, new obstacles appear, and previously free regions may become temporarily blocked. As a result, a path that is optimal (or even feasible) at planning time can quickly become invalid at execution time.

Classical shortest-path methods on graphs (e.g., Dijkstra and A*) provide strong optimality guarantees for static maps when costs are known [3, 6]. However, dynamic environments introduce an additional time dimension and demand either frequent replanning or policies that can react online. A practical planner must balance *safety* (collision avoidance), *efficiency* (short travel time/length), and *real-time performance* under limited sensing and compute budgets [11, 17].

1.2 Why Dynamic Environments Are Hard

Compared to static planning, dynamic environments create several coupled challenges:

- **Time-varying feasibility:** The free space changes over time, so a solution is more naturally a trajectory in space–time rather than a purely geometric path.
- **Partial observability and uncertainty:** The agent often only observes a local neighborhood and must act before it has perfect knowledge of future obstacle motion [17].
- **Responsiveness:** Replanning from scratch at every step can be too slow; incremental or hierarchical approaches are commonly used to reduce latency [9, 16].
- **Safety with moving obstacles:** Avoidance must consider relative motion. Reactive techniques such as the Dynamic Window Approach and (reciprocal) velocity obstacles have proven effective for local collision avoidance in real time [5, 4, 1].

These difficulties motivate hybrid solutions that combine global reasoning (to avoid dead-ends and reduce detours) with local reactive behaviors (to handle short-term dynamics).

1.3 Scope and Approach of This Report

This report focuses on path planning for a single holonomic agent in a two-dimensional discrete grid with both static and moving obstacles. The agent receives local observations and must reach a fixed goal without collisions.

To address the real-time requirements of dynamic navigation, we adopt a **two-level hierarchical architecture**. A *global planner* computes a waypoint route using the static map, while a *local planner* continuously replans within the agent's observation window to avoid moving obstacles and track the global waypoints. This design follows a widely used separation of concerns in mobile robot navigation:

global planning provides long-horizon structure, while local planning provides short-horizon safety and responsiveness [11, 17].

1.4 Contributions

The main contributions of this work are:

- A unified experimental framework for evaluating multiple grid-based global planners (BFS, DFS, Dijkstra, and A*) under a shared map representation.
- A local replanning module that operates on local observations to avoid dynamic obstacles online while tracking global waypoints.
- A set of path post-processing techniques (sparsification and safety-margin enforcement) to improve waypoint quality and collision robustness.

1.5 Report Organization

The rest of the report is organized as follows. Section 2 reviews related work in motion planning for dynamic environments. Section 3 formalizes the problem setting and assumptions. Section 4 describes the proposed hierarchical planning methodology and the implemented algorithms. Section 5 presents the experimental setup and quantitative results. Finally, Section 6 concludes the report and discusses limitations and future directions.

2 RELATED WORKS

Path planning in dynamic environments lies at the intersection of classical motion planning, real-time replanning, and collision avoidance. This section summarizes the main research directions most relevant to our setting (grid-based navigation with moving obstacles and local sensing).

2.1 Graph Search on Grids and Static Maps

Many robotics systems discretize the workspace into a grid or graph and compute shortest paths using graph search. Dijkstra's algorithm provides optimal shortest paths for nonnegative edge costs [3], while A* accelerates search using admissible heuristics and remains optimal under standard assumptions [6]. These approaches are well suited for static environments but do not directly address time-varying obstacles, since the cost and feasibility of edges can change during execution [11].

2.2 Incremental and Anytime Replanning

To cope with changes in the environment or partial knowledge, incremental search methods reuse previous computation when the map changes. Early work on D* showed how to efficiently update shortest paths in partially known environments [16]. D* Lite reformulates these ideas with a simpler implementation and strong practical performance [9]. Related incremental variants such as Lifelong Planning A* (LPA*) maintain shortest-path information across repeated queries [10].
Another complementary idea is *anytime* planning, where an initial feasible solution is produced quickly and then improved as time allows. Algorithms such as Anytime Repairing A* (ARA*) explicitly trade solution suboptimality for speed and can refine plans online [13]. These approaches are attractive for dynamic environments because they can deliver a usable plan under strict time constraints while still enabling improvement when computation is available.

2.3 Sampling-Based Motion Planning

For high-dimensional or continuous configuration spaces, sampling-based planners are widely used. Probabilistic Roadmaps (PRM) construct a roadmap by sampling collision-free configurations and connecting nearby samples [8]. Rapidly-exploring Random Trees (RRT) grow trees toward unexplored regions and are effective in complex spaces [12]. RRT* extends RRT with asymptotic optimality guarantees, improving solution quality as the number of samples increases [7].
While these methods excel in static environments, dynamic obstacles require either frequent replanning, incorporation of time into the state (space–time planning), or local reactive collision avoidance layered on top of a nominal plan [11].

2.4 Trajectory Optimization and Model-Predictive Planning

Instead of searching discretized graphs, trajectory optimization methods solve for a continuous trajectory by minimizing an objective that penalizes collisions, control effort, and smoothness. CHOMP introduced functional-gradient optimization for collision-free trajectories [14], and later work such as TrajOpt formulated planning as sequential convex optimization with efficient collision checking [15]. In dynamic settings, model-predictive control (MPC)-style replanning can continuously update the trajectory over a receding horizon, providing responsiveness to changing obstacles when computation permits [11].

2.5 Reactive Collision Avoidance for Moving Obstacles

Many deployed navigation stacks separate global path planning from local collision avoidance. The Dynamic Window Approach (DWA) is a classic real-time local planner that selects feasible velocity commands while considering robot dynamics [5]. Velocity-obstacle formulations model imminent collisions in velocity space, enabling fast avoidance in dynamic scenes [4]. Reciprocal Velocity Obstacles (RVO) and its extensions address multi-agent interactions by accounting for mutual avoidance behaviors [1, 2]. Reactive techniques are particularly relevant when the agent only observes a local window, since they can respond immediately to nearby moving obstacles even when long-horizon predictions are uncertain [17].

2.6 Learning-Based Planning (Brief Overview)

Recent work also explores learning-based components (e.g., imitation learning or reinforcement learning) for navigation policies in dynamic environments, typically leveraging large-scale data or simulation. These methods can yield strong empirical performance but may require careful safety handling and often integrate with classical planners to provide guarantees or fallback behaviors [17, 11].

2.7 Positioning of Our Work

Motivated by these strands, our report follows a pragmatic and widely used design: a global planner computes a waypoint route on a static grid using standard graph-search methods (BFS/DFS/Dijkstra/A*), while a local replanning component reacts online within the agent's observation range to avoid moving obstacles. This mirrors the common global–local decomposition found in many robotics navigation systems [17, 11].

3 PROBLEM FORMULATION

We study motion planning for a single holonomic robot operating in a two-dimensional discrete grid. The objective is to synthesize a collision-free trajectory from a known start position to a fixed goal while both static and dynamic obstacles may be present. Unlike classical formulations that assume a static environment, we explicitly allow obstacles to move over time and require the planner to account for this evolution when producing a feasible path.

3.1 General Assumptions

- The environment is a discrete gridworld (discrete space and discrete time).
- The robot is holonomic with deterministic dynamics and is modeled as a point mass.
- The initial robot state and the goal state are known and fixed.
- The number of obstacles and their initial positions are known.
- The robot state is observable at every time step.
- The robot is equipped with a local sensor that yields a 5×5 occupancy grid centered on the robot (local observation of nearby cells).

3.2 Mathematical Model

The planning problem is defined on the following spaces:

$$\begin{aligned}
\text{State space:} \quad & S = \{(x, y) \mid x, y \in \mathbb{N},\ 0 \le x \le X_{\text{dim}},\ 0 \le y \le Y_{\text{dim}}\} \subset \mathbb{N}^2, \\
\text{Action space:} \quad & A = \{(0,1),\ (0,-1),\ (-1,0),\ (1,0),\ (0,0)\} \subset \mathbb{Z}^2, \\
\text{Observation space:} \quad & G \in [0,1]^{5\times 5}, \\
\text{Static obstacles:} \quad & O \subset S.
\end{aligned}$$

The robot dynamics are given by a deterministic transition function

$$f : S \times A \to S,$$

which maps the current state and chosen action to the next state. Dynamic obstacles evolve over time; let $M_i \subset S$ denote the set of moving obstacle states at discrete time index $i \in \{0, \dots, N-1\}$ for a planning horizon of length N.

3.3 Planning Objective

We seek a trajectory $T = \{S_0, S_1, \dots, S_{N-1}\}$ such that S_0 is the known initial state and S_{N-1} coincides with the goal state. The trajectory is feasible if for every $i \in \{0, \dots, N-2\}$ there exists an action $a \in A$ satisfying

$$f(S_i, a) = S_{i+1}, \qquad S_i \notin O, \qquad S_i \notin M_i.$$

The task is to compute at least one such collision-free trajectory that respects both static and time-varying obstacle sets under the stated assumptions.

4 METHODOLOGY

4.1 System Architecture

The implemented system employs a **two-level hierarchical path planning architecture** that combines global strategic planning with local reactive navigation to handle dynamic environments containing moving obstacles. This hierarchical approach decomposes the complex navigation problem into two manageable sub-problems:

1. **Global Planning Layer:** Computes high-level waypoint paths from start to goal using the static map
2. **Local Planning Layer:** Performs real-time reactive navigation within the agent's observation range, adapting to moving obstacles

4.2 Global Path Planning Algorithms

The global planning layer computes strategic paths through the environment using the complete static map. Four different algorithms have been implemented and evaluated.

4.2.1 Grid-Based Breadth-First Search (Grid BFS)

Grid BFS performs a complete breadth-first exploration of the configuration space to find the shortest path in terms of the number of grid cells traversed.

Algorithm:

1. Initialize queue with start position
2. While queue is not empty:
    - Dequeue current node
    - If current node is goal, reconstruct and return path
    - For each 8-connected neighbor:
        - Check if neighbor is free and satisfies safety margin
        - If not visited, add to queue and record parent
3. Apply line-of-sight sparsification to reduce waypoints
4. Push waypoints away from walls to ensure safety margin

Key Features:

- Uses 8-connected grid neighborhood (allowing diagonal movements)
- Maintains safety margin (default: 2 cells) from obstacles using pre-computed clearance map
- Clearance map computed via multi-source BFS from all obstacle cells
- Guarantees shortest path if one exists
- Time complexity: $O(|V| + |E|)$ where V is the number of free cells

4.2.2 Grid-Based Depth-First Search (Grid DFS)

Grid DFS explores paths deeply before backtracking, using a stack-based approach to traverse the configuration space.

Algorithm:

1. Initialize stack with start position
2. Initialize visited set
3. While stack is not empty:
    - Pop current node
    - If current node is goal, reconstruct and return path
    - For each 8-connected neighbor:
        - Check if neighbor is free and satisfies safety margin
        - If not visited, add to stack, visited set, and record parent
4. Apply post-processing (sparsification and wall-pushing)

Key Features:

- Lower memory footprint compared to BFS in large spaces
- Does not guarantee shortest path
- Can be faster when goal is "deep" in search tree
- Same safety margin enforcement as BFS

4.2.3 A* Search

A* is an informed search algorithm that uses a heuristic function to guide exploration toward the goal, combining actual cost from start with estimated cost to goal.

Cost Function:

$$f(n) = g(n) + h(n) \tag{1}$$

where:

- $g(n)$: actual cost from start (1.0 for cardinal moves, 1.414 for diagonal)
- $h(n)$: Manhattan distance heuristic to goal: $h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$

Algorithm:

1. Initialize priority queue with start node and $f(start) = h(start)$
2. While priority queue is not empty:
    - Extract node with minimum f-value
    - If current node is goal, reconstruct and return path
    - For each 8-connected neighbor:
        - Calculate tentative g-score
        - If better than previous g-score:
            - Update g-score and parent
            - Calculate f-score and add to priority queue
3. Apply Bresenham line-of-sight sparsification

Key Features:

- Uses an 8-connected neighborhood with move costs 1.0 (cardinal) and 1.414 (diagonal)
- Heuristic: Manhattan distance $h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$ (simple goal guidance)
- Note: with diagonal moves enabled, Manhattan distance can overestimate the true optimal cost; therefore this implementation is *not guaranteed* to be optimal in all cases
- Uses a min-heap (priority queue) for selecting the next node to expand

4.2.4 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path by exploring nodes in order of increasing cost from the start, without using goal-directed heuristics.

Algorithm:

1. Initialize distances: $d(start) = 0$, all others to ∞
2. Initialize unvisited set with all free cells
3. While unvisited set is not empty:
    - Select node u with minimum distance from unvisited set
    - If u is goal, reconstruct and return path
    - Remove u from unvisited set
    - For each neighbor v of u:
        - Calculate new distance: $d_{new} = d(u) + cost(u, v)$
        - If $d_{new} < d(v)$, update $d(v)$ and parent

Key Features:

- Guarantees optimal path (shortest in terms of actual distance)
- Edge costs: 1.0 for cardinal, 1.414 for diagonal movements
- More thorough but slower than A* for single-goal queries
- Implementation detail: selects the minimum-distance unvisited node via a linear scan over an unvisited set (i.e., no priority queue), giving worst-case time complexity $O(|V|^2)$

4.3 Path Optimization Techniques

Several optimization techniques are applied to improve path quality:

4.3.1 Clearance Map Computation

A clearance map is pre-computed using multi-source BFS to determine the distance from each free cell to the nearest obstacle:

1. Initialize all obstacle cells with distance 0 and add to queue
2. For each cell in queue, propagate distance to 8-connected neighbors
3. Result: each cell contains Chebyshev distance to nearest obstacle

This map enables efficient safety margin enforcement during planning.

4.3.2 Line-of-Sight Sparsification

The Bresenham line algorithm is used to reduce waypoint count:

1. Start with first waypoint as anchor
2. Find furthest waypoint visible from anchor (no obstacles on line)
3. Add this waypoint to sparse path and set as new anchor
4. Repeat until goal is reached

This reduces the dense cell-by-cell paths to sparse waypoint sequences.

4.3.3 Wall-Pushing Strategy

Waypoints are adjusted to maintain safety margin from obstacles:

1. For each waypoint, check if clearance exceeds margin
2. If insufficient clearance, search local neighborhood
3. Select nearby cell with maximum clearance
4. Replace waypoint with safer alternative

4.4 Local Path Planning Algorithms

The local planning layer handles real-time reactive navigation within the agent's observation range. Four reactive strategies have been implemented.

4.4.1 Reactive BFS Planner

Performs BFS within the agent's local observation window to find collision-free paths to the current waypoint.

Agent Graph Construction:

1. Define observation window: $[x_a - r, x_a + r] \times [y_a - r, y_a + r]$
2. For each observed obstacle, inflate by safety margin using Chebyshev distance
3. Create state space of free cells within observation
4. Build adjacency graph using 4-connected moves (UP/DOWN/LEFT/RIGHT). A `NONE` action exists but does not expand the search.

Waypoint Projection: When global waypoint falls outside observation or in inflated obstacles:

1. Find all free cells in observation
2. Compute Euclidean distance from each to waypoint
3. Select cell with minimum distance as projected goal

Path Computation:

1. Apply BFS on agent graph from current position to projected waypoint
2. Extract next step from computed path
3. Convert to action vector (movement direction)

Key Features:

- Fast replanning in constrained local space

- Guaranteed to find path in local observation if one exists
- Adapts to moving obstacles via observation updates
- Time complexity: $O(r^2)$ in the size of the observation window (roughly $(2r+1)^2$ cells)

4.4.2 Reactive DFS Planner

Uses depth-first search strategy for local navigation, exploring paths deeply before backtracking.

Algorithm:

1. Build agent graph with inflated obstacles
2. Apply recursive DFS from current position to projected waypoint
3. Return first found path (not necessarily shortest)
4. Track stuck counter to detect navigation failures

Key Features:

- Lower memory usage than BFS in local space
- Can find alternative paths in environments with multiple routes
- Useful when multiple similar-length paths exist
- May explore longer paths before finding goal

4.4.3 Potential Field Planner

Models navigation as moving under artificial forces: attractive force toward goal and repulsive forces from obstacles.

Force Model:

$$\vec{F}_{total} = \vec{F}_{attractive} + \vec{F}_{repulsive} \tag{2}$$

Attractive Force:

$$\vec{F}_{attractive} = k_a \cdot \frac{\vec{d}_{goal}}{||\vec{d}_{goal}||} \tag{3}$$

where k_a is attraction weight (default: 1.0) and $\vec{d}_{goal}$ is the direction vector to the goal.

Repulsive Force:

$$\vec{F}_{repulsive} = \sum_{i \in obstacles} k_r \cdot \frac{\vec{d}_{away,i}}{||\vec{d}_{away,i}||} \tag{4}$$

where k_r is repulsion weight (default: 2.0), computed for all obstacles within repulsion range (default: 3 cells).

Force-to-Action Conversion:

1. Compute total force vector
2. Determine dominant direction (horizontal vs. vertical)
3. Map to a **4-connected** discrete action: UP, DOWN, LEFT, or RIGHT (no diagonal actions in this planner)
4. Treat cells that violate the clearance margin as obstacles (repulsion is computed against the *inflated* free-space constraint)

Key Features:

- Smooth, reactive behavior without explicit path search
- Naturally avoids obstacles through repulsive forces
- Can suffer from local minima in complex configurations
- Computationally efficient: $O(r^2)$ for force computation
- No memory overhead for graph structures

4.4.4 Greedy Local Planner

Selects actions that maximize progress toward the goal while avoiding obstacles using a simple greedy heuristic.

Algorithm:

1. Calculate current distance to goal: $d_{current} = ||\vec{goal} - \vec{current}||$
2. For each of 8 possible actions (cardinal + diagonal):
   - Compute next position
   - Check validity (within global bounds, within the current observation window, not an obstacle, and satisfies clearance margin)
   - Calculate new distance to goal: d_{new}
   - If $d_{new} < d_{current}$, select this action
3. If no improving action found, increment stuck counter

Key Features:

- Simple and computationally efficient: $O(1)$ per step
- No memory overhead (no graph or queue structures)
- Works well in open spaces with sparse obstacles
- Can get trapped in local minima or dead ends
- Myopic decision-making without lookahead

4.5 Integration and Coordination

The hierarchical system coordinates global and local planning through several mechanisms:

4.5.1 Waypoint Following

1. Local planner maintains current waypoint index in global path
2. Projects current waypoint into local observation space
3. Executes local plan to reach projected waypoint
4. When waypoint reached (within a small tolerance), advances to next waypoint
5. Repeats until final goal is reached

4.5.2 Stuck Detection and Recovery

To handle situations where the agent cannot make progress, the execution loop tracks both *no-motion* and *no-progress* events:

1. Monitor agent position changes across time steps, and whether distance to the *current waypoint* decreases
2. Increment stuck counter when the agent does not move, or when the Manhattan distance to the current waypoint does not decrease
3. When stuck counter exceeds a threshold (typically 15–20 steps depending on the execution script):
    - Trigger global replanning from current position
    - Compute new waypoint sequence to final goal
    - Reset waypoint index and stuck counter
    - Resume local planning with new global path

4.5.3 Observation Updates

After each action execution:

1. Agent moves to new position
2. Observe environment within sensing range
3. Detect obstacle positions (static and moving)
4. Reconstruct agent graph with updated observations
5. Inflate newly detected obstacles by safety margin (Chebyshev radius)
6. Continue local planning with updated graph

4.5.4 Safety Margin Enforcement

Consistent safety enforcement across both layers:

- **Global layer:** Pre-computes clearance map over entire environment using multi-source BFS with Chebyshev distance metric
- **Local layer:** Computes local clearance map within observation window for each update
- **Coordination:** Both layers enforce a configurable margin; in the provided runner/benchmark setup the global margin is typically 2 cells while the local inflation margin is set to 1 cell
- **Obstacle inflation:** Chebyshev ball of radius m around each obstacle cell

4.6 Algorithm Comparison Framework

The implementation provides a flexible framework for comparative evaluation of different algorithm combinations.

4.6.1 Configurable Parameters

- **Global planner:** `grid_bfs`, `grid_dfs`, `astar`, `dijkstra`
- **Local planner:** `reactive_bfs`, `reactive_dfs`, `potential_field`, `greedy`
- **Number of moving obstacles:** Configurable (default: 100 in the demo runner; benchmark evaluates $\{0, 50, 100, 200\}$)

- **Safety margin:** Configurable clearance from obstacles (global typically 2; local inflation typically 1)
- **Agent sensing range:** Observation window radius (default: 2 cells)
- **Map selection:** Multiple test environments (maps 1-7)

4.6.2 Performance Metrics

The system measures the following metrics for evaluation:

- **Global planning time:** Time to compute initial waypoint path (seconds)
- **Total execution steps:** Number of discrete time steps to reach goal
- **Path length:** Total number of cells traversed in complete path
- **Success rate:** Goal reached vs. maximum step limit (1000 steps)
- **Replanning frequency:** Number of stuck detections requiring global replanning

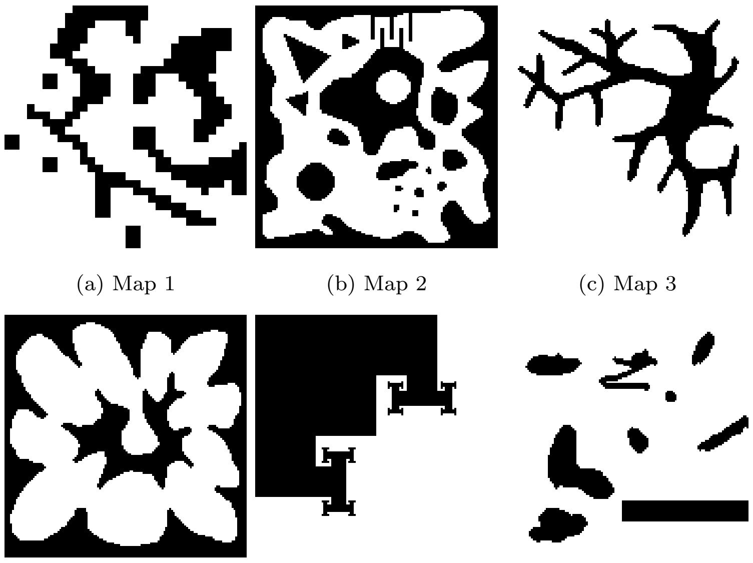
(a) Map 1 (b) Map 2 (c) Map 3
(d) Map 4 (e) Map 5 (f) Map 6

Figure 1: Six benchmark maps used for evaluation.

5 EXPERIMENTS

5.1 Experiment Design

We evaluate the hierarchical planner using the automated benchmark in `Code/benchmark.py`. The benchmark tests all $4 \times 4 = 16$ combinations of global planners (`grid_bfs`, `grid_dfs`, `astar`, `dijkstra`) and local planners (`reactive_bfs`, `reactive_dfs`, `potential_field`, `greedy`) across 7 maps and 4 obstacle counts $\{0, 50, 100, 200\}$, totaling 448 runs.

Each run uses an agent observation radius $r = 2$ (a 5×5 local window) and inflates observed obstacles with margin 1. Runs terminate at 2000 steps or 100 seconds; success is declared when Manhattan distance to the final goal is ≤ 3 (waypoints use tolerance 2). When the agent fails to move or does not reduce distance to its current target for 15 steps, the system triggers global replanning from the current state to the final goal.

We report success rate (primary), executed path length, direction changes (smoothness proxy), accumulated global planning time (including replans), total wall-clock time, and steps. Results are saved to `Code/results/benchmark_results.csv` and summarized using `Code/analyze_results.py`.

Figure 1 shows representative benchmark maps (white: free space, black: obstacles).

5.2 Benchmark Results

The full benchmark contains 448 test configurations (7 maps $\times$ 4 obstacle settings $\times$ 16 algorithm combinations). Overall, 237/448 runs succeeded, corresponding to a 52.9% success rate.

For successful runs, the average path length is 327.10 ± 243.60 cells and the average total runtime is 5.3429 ± 5.1084 seconds.

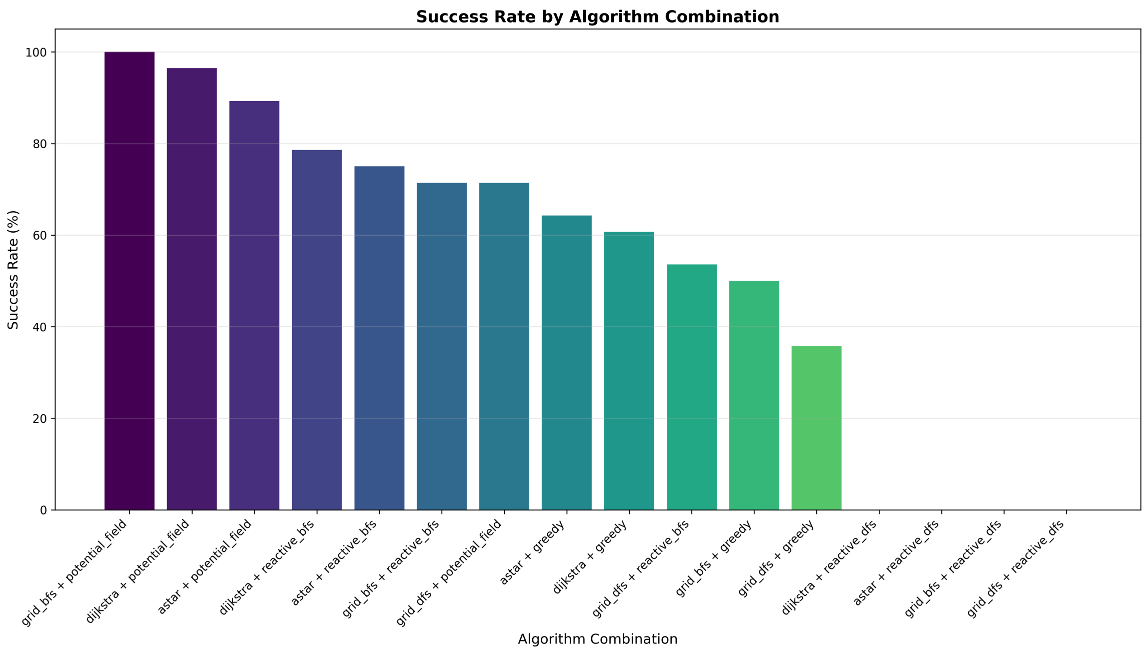

Figure 2: Success rate by global+local algorithm combination (computed from `benchmark_results.csv`).

5.2.1 Success Rate by Algorithm Combination

Figure 2 ranks all 16 planner combinations by success rate (each entry aggregates 7 maps × 4 obstacle settings, i.e., 28 runs per combination). Figure 2 visualizes the same ranking.

Overall, combinations with `potential_field` achieve the best reliability (top three and near-perfect success with `dijkstra` and `astar`, and perfect success with `grid_bfs`). `reactive_bfs` is competitive but consistently below `potential_field`, while `greedy` yields mid-range success rates. In contrast, `reactive_dfs` fails across all global planners (0/28), indicating that its local behavior is not robust to dynamic obstacles and the benchmark replanning trigger.

5.2.2 Path Quality and Runtime

Figure 3 compares the average path length, number of direction changes, and total runtime over successful runs. Overall, `potential_field` produces the shortest and smoothest trajectories (fewest direction changes), especially with `astar` and `grid_bfs`. `reactive_bfs` is a competitive middle ground (moderate path length and smoothness). In contrast, `greedy` tends to generate longer and less smooth paths, with `dijkstra + greedy` showing both high direction-change counts and the largest planning times, which is consistent with more frequent replanning and oscillatory local behavior.

5.2.3 Impact of Obstacle Density

Figure 4 shows how increasing the number of moving obstacles affects the top-performing combinations. Across obstacle counts, `potential_field` remains the most stable: it keeps direction-change counts low and planning time nearly flat, with only mild variation in path length. In contrast, `reactive_bfs` becomes noticeably less smooth as obstacle density increases (higher direction changes) and exhibits larger

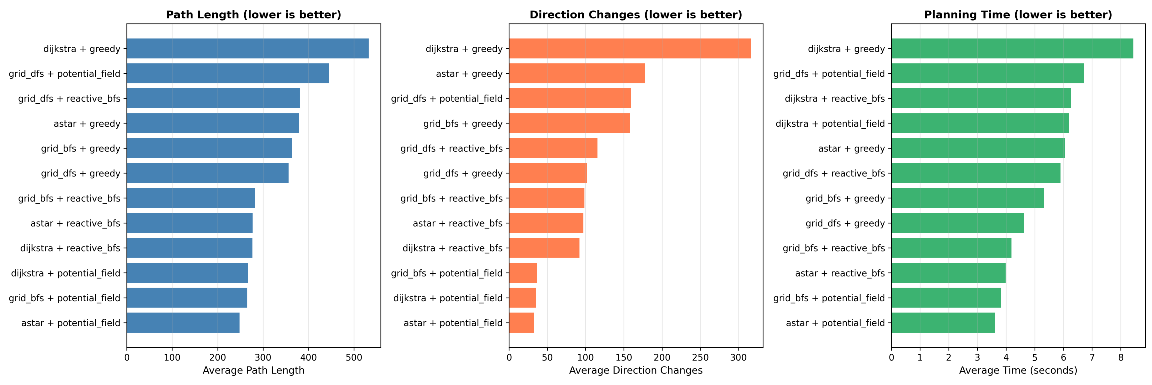

Figure 3: Comparison of path length, direction changes, and runtime across algorithm combinations.

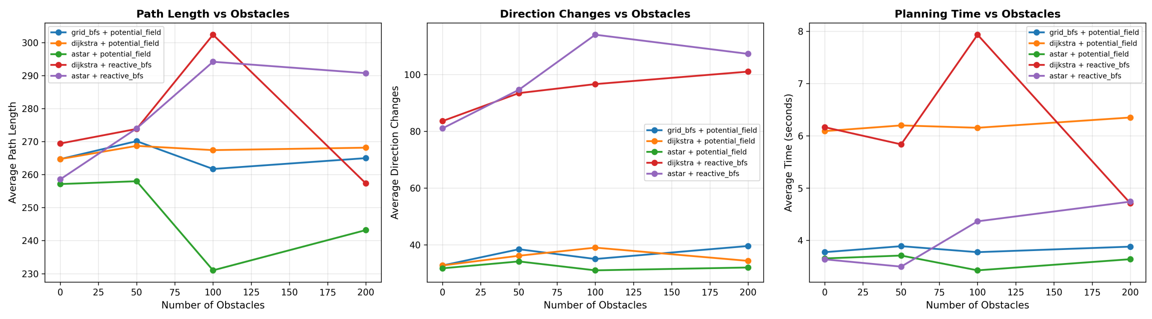

Figure 4: Effect of obstacle count on path length, direction changes, and runtime (top combinations).

variability in both path length and runtime, including a pronounced slowdown around 100 obstacles for `dijkstra + reactive_bfs`. Overall, denser dynamic obstacles primarily increase local maneuvering (more turns) and can trigger additional replanning overhead, especially for reactive local policies.

5.2.4 Heatmap Summary

Figure 5 provides a compact heatmap view of average performance across global and local methods.

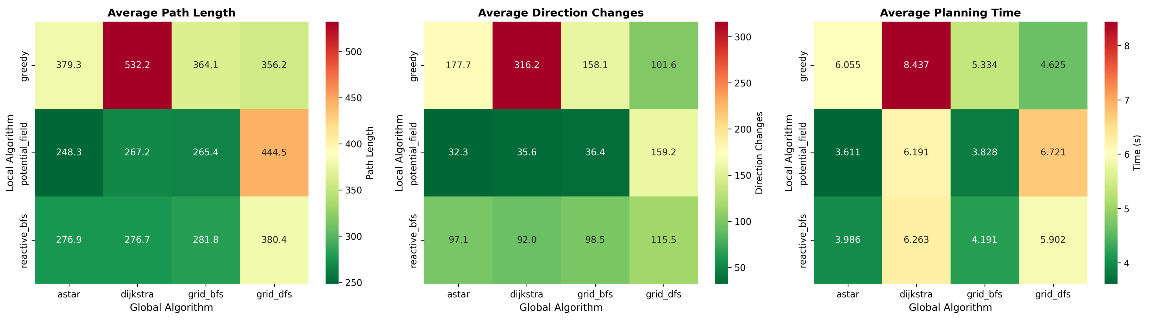

Figure 5: Heatmap summary of average path length, direction changes, and runtime by method.

6 CONCLUSION

References

[1] Jur van den Berg, Ming Lin, and Dinesh Manocha. “Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2008, pp. 1928–1935.

[2] Jur van den Berg et al. “Reciprocal n-Body Collision Avoidance”. In: *Robotics Research: The 14th International Symposium ISRR*. Springer, 2011, pp. 3–19.

[3] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[4] Paolo Fiorini and Zvi Shiller. “Motion Planning in Dynamic Environments Using Velocity Obstacles”. In: *The International Journal of Robotics Research* 17.7 (1998), pp. 760–772.

[5] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. “The Dynamic Window Approach to Collision Avoidance”. In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[7] Sertac Karaman and Emilio Frazzoli. “Sampling-based Algorithms for Optimal Motion Planning”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2011, pp. 3531–3537.

[8] Lydia E. Kavraki et al. “Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 1996, pp. 566–580.

[9] Sven Koenig and Maxim Likhachev. “D* Lite”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2002, pp. 476–483.

[10] Sven Koenig and Maxim Likhachev. “Lifelong Planning A*”. In: *Artificial Intelligence* 155.1–2 (2005), pp. 93–146.

[11] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[12] Steven M. LaValle. “Rapidly-Exploring Random Trees: A New Tool for Path Planning”. In: *Technical Report*. Iowa State University. 1998.

[13] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. “ARA*: Anytime A* with Provable Bounds on Sub-Optimality”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2003.

[14] Nathan Ratliff et al. “CHOMP: Gradient Optimization Techniques for Efficient Motion Planning”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2009, pp. 489–494.

[15] John Schulman et al. “Motion Planning with Sequential Convex Optimization and Convex Collision Checking”. In: *Robotics: Science and Systems (RSS)*. 2014.

[16] Anthony Stentz. “Optimal and Efficient Path Planning for Partially-Known Environments”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 1994, pp. 3310–3317.

[17] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.