Grid-Based Pathfinding Using Dijkstra and A* Algorithms

January 10, 2026

Abstract

Path planning is a fundamental problem in robotics and artificial intelligence, where an agent must find an optimal path from a start location to a goal while avoiding obstacles. In this project, we model a two-dimensional grid as a graph and study shortest-path algorithms under realistic movement constraints. We implement Dijkstra's algorithm and the A* search algorithm on a large grid with obstacles, allowing movement in eight directions with non-uniform costs. We analyze path optimality, number of expanded nodes, and computational efficiency, and compare grid-based solutions with the true geometric shortest path.

1 Introduction

Pathfinding problems appear in a wide range of applications including robot navigation, autonomous vehicles, video games, and network routing. A common abstraction is to discretize the environment into a grid and treat each cell as a node in a graph. Classical graph search algorithms can then be applied to compute shortest paths.

Breadth First Search (BFS) is optimal only when all edges have equal cost, which is unrealistic for diagonal or continuous movement. Dijkstra's algorithm generalizes BFS to weighted graphs, while A* improves efficiency by using heuristics to guide the search toward the goal.

This project focuses on implementing and comparing Dijkstra's algorithm and A* search for grid-based path planning with obstacles.

2 Graph Theory Background

2.1 Graphs: Basic Definitions

A graph is a mathematical structure used to model pairwise relationships between objects. It consists of:

- A set of vertices (or nodes)
- A set of edges connecting pairs of vertices

Graphs are widely used to model real-world systems such as road networks, social networks, file systems, and dependency graphs.

2.2 Types of Graphs

2.2.1 Directed and Undirected Graphs

- **Undirected Graph**: Edges have no direction. Connections are bidirectional (e.g., two-way roads).
- **Directed Graph**: Edges have a direction. Connections are one-way (e.g., follower relationships on social media).

2.2.2 Weighted and Unweighted Graphs

- **Unweighted Graph**: All edges are assumed to have equal cost.
- **Weighted Graph**: Each edge has an associated cost (distance, time, energy, etc.).

Grid-based path planning naturally forms a **weighted graph** when diagonal and straight movements have different costs.

2.3 Trees

A tree is a special type of graph with the following properties:

- No cycles
- Exactly one unique path between any two nodes
- If a tree has N nodes, it has exactly $N - 1$ edges

Trees are used in file systems, organizational hierarchies, and search structures such as binary search trees.

2.4 Why Graph Traversal is Needed

Given a graph, we often want to:

- Visit all nodes
- Check connectivity
- Find a path between two nodes

Two fundamental graph traversal algorithms are:

- Depth First Search (DFS)
- Breadth First Search (BFS)

2.5 Depth First Search (DFS)

2.5.1 Intuition

DFS explores a graph by going as deep as possible along one path before backtracking. It behaves like exploring a maze by always taking a new corridor until no further progress is possible.

2.5.2 Algorithm

1. Start from a source node
2. Mark the node as visited
3. Recursively visit each unvisited neighbor
4. Backtrack when no unvisited neighbors remain

DFS can be implemented using recursion or an explicit stack.

2.5.3 Applications of DFS

- Exploring all possible paths
- Cycle detection
- Topological sorting
- Puzzle and maze solving

2.6 Breadth First Search (BFS)

2.6.1 Intuition

BFS explores a graph level by level, visiting all nodes at distance 1 from the source, then distance 2, and so on. It uses a queue data structure.

2.6.2 Algorithm

1. Start at the source node and mark it visited
2. Push the source into a queue
3. While the queue is not empty:
    - Pop a node from the queue
    - Visit all unvisited neighbors and add them to the queue

2.6.3 Applications of BFS

- Finding shortest paths in unweighted graphs
- Finding connected components
- Level-order traversal of trees

2.7 BFS vs DFS

Feature	**BFS**	**DFS**
Data Structure	Queue	Stack / Recursion
Exploration Style	Level-by-level	Go deep first
Shortest Path	Yes (unweighted)	No
Typical Uses	Shortest paths	Full exploration, cycles

Table 1: Comparison of BFS and DFS

2.8 Limitations of BFS and DFS

DFS may find a path to the target, but it does not guarantee the shortest path. BFS guarantees shortest paths only when all edges have equal cost.

In real-world scenarios such as road networks and robotic navigation, edge costs are often non-uniform. Therefore, BFS is insufficient.

2.9 Motivation for Dijkstra's Algorithm

To compute shortest paths in graphs with non-negative and non-uniform edge weights, Dijkstra's algorithm is used. It can be seen as a generalization of BFS that replaces the queue with a priority queue.

This motivates the use of Dijkstra's algorithm and heuristic-based extensions such as A* for efficient path planning in weighted graphs.

3 Problem Formulation

3.1 Grid Representation

The environment is modeled as a 200×200 grid. Each grid cell represents a vertex in a graph. Traversable cells are connected to their neighbors via weighted edges, while obstacle cells are removed from the graph.

3.2 Movement Model

Movement is allowed in eight directions:

$$(1,0),(-1,0),(0,1),(0,-1),(\pm1,\pm1)$$

with the following costs:

- Horizontal / vertical move: cost 1
- Diagonal move: cost $\sqrt{2}$

This cost model approximates Euclidean motion on a grid.

3.3 Start, Goal, and Obstacles

- Start cell: $(0, 0)$
- Goal cell: $(199, 199)$
- Obstacles are rectangular regions that cannot be traversed

4 Algorithms

4.1 Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights. It repeatedly expands the node with the smallest known distance using a priority queue.

4.2 A* Search Algorithm

A* is a best-first search algorithm that uses a heuristic function $h(n)$ to estimate the remaining cost from node n to the goal. The evaluation function is:

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

where:

- $g(n)$ is the cost from the start to node n
- $h(n)$ is a heuristic estimate of the remaining distance

4.3 Heuristic Design

For an 8-direction grid with diagonal movement, we use the diagonal distance heuristic:

$$h = D(dx + dy) + (D_2 - 2D)\min(dx, dy)$$

where:

$$dx = |x_n - x_g|, \quad dy = |y_n - y_g|$$
$$D = 1, \quad D_2 = \sqrt{2}$$

This heuristic does not overestimate the true shortest path and is therefore admissible.

5 Implementation Details

5.1 Graph Construction

The grid is stored as a 2D array, with obstacles marked as invalid cells. A priority queue (min-heap) is used to efficiently select the next node to expand.

5.2 Path Reconstruction

A parent array stores the predecessor of each visited cell. Once the goal is reached, the path is reconstructed by backtracking from the goal to the start.

5.3 Node Expansion Metric

The number of expanded nodes is counted as the number of times a node is popped from the priority queue and marked visited. This metric reflects algorithm efficiency.

6 Experimental Results

6.1 Dijkstra's Algorithm

- Shortest path cost: ≈ 300.8
- Expanded nodes: High
- Guarantees optimality but explores a large portion of the grid

6.2 A* Search

- Shortest path cost: ≈ 300.8
- Expanded nodes: Significantly fewer than Dijkstra
- More efficient due to heuristic guidance

6.3 Comparison

Metric	**Dijkstra**	**A***
Path Optimality	Optimal	Optimal
Expanded Nodes	High	Low
Heuristic Used	No	Yes
Efficiency	Lower	Higher

Table 2: Comparison of Dijkstra and A*

7 Geometric Shortest Path Analysis

Although grid-based Dijkstra with diagonal moves is optimal under the grid constraints, it does not produce the true shortest path in continuous 2D space. The globally shortest path around polygonal obstacles bends only at obstacle corners.

Using straight-line segments between obstacle vertices, the true geometric shortest path was found to be approximately:

$$287.66$$

which is shorter than the grid-based result.

8 Conclusion

This project demonstrated the application of Dijkstra and A* algorithms to grid-based path planning with obstacles. While both algorithms produce optimal paths under the grid model, A* significantly reduces node expansions through heuristic guidance. The study highlights the limitations of grid discretization and motivates more advanced geometric planning techniques for real-world navigation.