Pathfinding Algorithms: From Basics to Game Optimizations

Introduction to Pathfinding

Pathfinding is the computational problem of finding a route between two points while navigating around obstacles. In games, this determines how NPCs, enemies, and units move through the game world.

Fundamental Algorithms

Breadth-First Search (BFS)

The simplest approach explores all neighbors at the current distance before moving further. It guarantees the shortest path in terms of steps but doesn't consider actual distances or costs.

Dijkstra's Algorithm

An improvement over BFS that handles weighted graphs where edges have different costs (like terrain difficulty). It explores nodes in order of their distance from the start, guaranteeing the optimal path.

A *(A-Star)**

The workhorse of game pathfinding. A* combines Dijkstra's actual cost tracking with a heuristic estimate of remaining distance to the goal. The key formula is:

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

Where g(n) is the actual cost from start to node n, and h(n) is the estimated cost from n to goal. Common heuristics include Manhattan distance (for grid-based movement) and Euclidean distance (for free movement).

Game-Specific Optimizations

Games need to find paths for dozens or hundreds of units every frame, so raw algorithms aren't enough. Here are the main techniques studios use:

Hierarchical Pathfinding

Divide the map into large regions or clusters. First find a high-level path between regions, then compute detailed paths within each region. This is like planning a road trip: you first decide which cities to pass through, then worry about street-level navigation in each city.

Navigation Meshes (NavMesh)

Instead of grids, represent walkable space as convex polygons. This drastically reduces the node count compared to fine-grained grids. Pathfinding happens between polygon centers, and units move freely within each polygon.

Jump Point Search (JPS)

For uniform-cost grids, JPS identifies "jump points" where the path must turn, skipping over straight-line segments. This can be 10-20x faster than vanilla A* on open maps.

Goal Bounding

Quickly reject impossible paths by checking if the goal is reachable at all. Store precomputed connectivity data or use simple geometric tests before running expensive pathfinding.

Cached and Incremental Pathfinding

Store recently computed paths and reuse them when units make similar requests. For moving targets, use algorithms like D* Lite that efficiently update paths when the goal shifts rather than recalculating from scratch.

Theta *and Lazy Theta***

Variants of A* that produce smoother, more natural- looking paths by allowing line- of- sight shortcuts that aren't constrained to grid edges. The "lazy" version defers expensive visibility checks for better performance.

Flow Fields

When many units share the same destination (like an RTS army converging on a base), compute a single flow field showing the best direction to move from every grid cell. Each unit just follows the field, avoiding redundant pathfinding.

Time-Slicing and Asynchronous Pathfinding

Don't compute all paths in one frame. Spread pathfinding across multiple frames or run it on background threads. Units can continue with their current path while awaiting updates.

Local Steering and Obstacle Avoidance

Pathfinding gives the strategic route; local steering handles tactical movement. Algorithms like RVO (Reciprocal Velocity Obstacles) prevent units from colliding with each other and dynamic obstacles without re-pathing.

Precomputed Pathfinding Data

For static maps, precompute all-pairs shortest paths or distance tables between key locations. This trades memory for runtime speed, useful for strategic decision- making about which objective to pursue.

Path Simplification

After finding a path, remove unnecessary waypoints. If you can draw a straight line from waypoint A to waypoint C that avoids obstacles, delete waypoint B. This produces cleaner movement and reduces memory.

Different Algorithms for Different Scenarios

Use simple algorithms (even just "move toward goal") for small distances or when precision doesn't matter. Reserve A* for important, long-distance paths. Use specialized algorithms for specific unit types (flying units might use simpler 3D pathfinding since they have fewer obstacles).

Practical Considerations

Real game pathfinding systems combine multiple techniques. A typical setup might use a NavMesh with hierarchical pathfinding, asynchronous A* computation, and local steering for collision avoidance. The specific mix depends on game genre, unit count, map complexity, and performance budget.

The art is knowing which corners to cut—most players won't notice if an enemy takes a slightly suboptimal path, but they'll definitely notice if the game stutters because pathfinding is eating the CPU.