Navigation and Pathfinding Systems

1. What is Pathfinding and Why Do We Need It?

1.1 The Pathfinding Problem

Imagine you're playing a strategy game and you click on a unit to move it across the map. How does the computer figure out the best route? This is **pathfinding** - finding the optimal path from point A to point B while avoiding obstacles.
Real Examples in Games:
NPCs walking around obstacles in The Sims
Units navigating terrain in *Clash Of Clans*
Enemies chasing the player in Pac-Man
GPS-style navigation in open-world games like Grand Theft Auto

1.2 Why This Matters

Good pathfinding makes games feel intelligent and responsive. Bad pathfinding breaks immersion. Nobody wants to see enemies walk into walls or units take ridiculous detours.

2. Representing Game Worlds as Graphs

2.1 From Game Maps to Graphs

Before we can find paths, we need to represent our game world in a way the computer can understand. We use **graphs** made of:
Nodes (Vertices): Individual positions/tiles where a character can stand
Edges: Connections between adjacent positions (where you can move)
Weights: The "cost" of moving along an edge (distance, terrain difficulty, etc.)
Example: Subway map - stations are nodes, rail lines are edges, and travel time is the weight.

2.2 Grid-Based Graphs (Our Focus)

The simplest and most common approach in games is a **grid graph**:
Square Grids: Like a chessboard
Hexagonal Grids: Six-sided tiles (think Civilization, Settlers of Catan)
Each tile is a node, and you can move to adjacent tiles. Some tiles might be walls (no node), water (high cost), or roads (low cost).

3. The Simple Approach: Breadth-First Search (BFS)

3.1 How BFS Works

Breadth-First Search explores the map like ripples in a pond:

Start at your starting position
Check all immediate neighbors (1 step away)
Then check all positions 2 steps away
Continue until you reach the goal
Visual Analogy: Imagine pouring water on the start tile - it spreads outward evenly in all directions.

3.2 The Problem with BFS

BFS **always finds the shortest path**, but it's inefficient.
Example: If your goal is directly to the east, BFS will still search north, south, and west, wasting time exploring in the wrong directions.
In a 50×50 grid, BFS might check 1,000+ tiles just to find a path across!

4. The Key Insight: Using a Heuristic

4.1 What's a Heuristic?

A **heuristic** is an educated guess about which direction to search. Instead of exploring blindly in all directions, we use knowledge about where the goal is to guide our search.
Example: If you're trying to get from New York to Los Angeles, you don't explore routes going east into the Atlantic Ocean. You know the general direction (west) and prioritize those routes

4.2 Distance as a Heuristic

For pathfinding, we use **distance to the goal** as our heuristic:
Manhattan Distance (for 4-directional grids):

```
h = |goal_x - current_x| + |goal_y - current_y|
```

This counts the number of horizontal + vertical steps to the goal (like walking city blocks in Manhattan).
Euclidean Distance (for any movement):

```
h = √[(goal_x - current_x)² + (goal_y - current_y)²]
```

This is the straight-line "as the crow flies" distance.

5. A* Algorithm: The Best of Both Worlds

5.1 The A* Formula

A* combines two pieces of information:
f(n) = g(n) + h(n)
Where:
g(n) = Actual cost from START to current position (what we know for sure)
h(n) = Estimated cost from current position to GOAL (our educated guess)
f(n) = Estimated total cost of path through this position
Think of it like planning a road trip:

g(n) = miles you've already driven
h(n) = estimated miles remaining to destination
f(n) = estimated total trip length

5.2 How A* Works (Step by Step)

```
1. Put the START node in an "open list" (nodes to explore)
2. While there are nodes to explore:
   a. Pick the node with the LOWEST f(n) value
   b. If it's the GOAL - success! Trace back the path
   c. Otherwise, mark it as "closed" (already explored)
   d. Look at all its neighbors:
      - Calculate their g(n) = current g + movement cost
      - Calculate their h(n) = distance to goal
      - Calculate their f(n) = g(n) + h(n)
      - Add them to the open list (or update if better path found)
3. If open list becomes empty - no path exists!
```

5.3 Why A* is Optimal

A* has a beautiful mathematical guarantee: *If your heuristic never overestimates the real distance, A will always find the shortest path.**
This property is called **admissibility**. Since Manhattan and Euclidean distances never overestimate (you can't get there faster than a straight line), they're admissible heuristics.

6. A* vs Other Algorithms: A Comparison

Algorithm	Finds Shortest Path?	Explores Efficiently?	Use Case
BFS	✓ (unweighted)	✗ (searches everywhere)	Simple maze solving
Dijkstra's	✓ (weighted)	✗ (searches everywhere)	When you need paths to ALL nodes
Greedy Best-First	✗ (not guaranteed)	✓ (very fast)	When speed > optimality
A*	✓ (with good heuristic)	✓ (directed search)	**Game pathfinding**

Performance Example: In a 50×50 grid finding a path across the diagonal:
BFS: ~1,200 nodes explored
A* with Manhattan: ~50-100 nodes explored (10-20× faster!)

A* Algorithm - Step-by-Step Implementation

```
def find_path(start_node, goal_node):
    """
    Find the shortest path from start_node to goal_node using A*.

    Args:
        start_node: Starting node (must have neighbors, G, H, F properties)
        goal_node: Goal node

    Returns:
        List of nodes representing the path, or None if no path exists
    """
    # STEP 1: Initialize the open and closed lists
    open_list = [start_node]  # Nodes to be evaluated
    closed_list = []          # Nodes already evaluated

    # STEP 2: Main algorithm loop
    while len(open_list) > 0:

        # STEP 3: Get the node with the lowest F score from open list
        current = open_list[0]
        for node in open_list:
            if node.F < current.F or (node.F == current.F and node.H < current.H):
```

```
        current = node

# STEP 4: Move current node from open to closed list
closed_list.append(current)
open_list.remove(current)

# STEP 5: Check if we've reached the goal
if current == goal_node:
    return reconstruct_path(current)

# STEP 6: Process each neighbor of the current node
for neighbor in current.neighbors:

    # Skip if neighbor is not walkable or already evaluated
    if not neighbor.walkable or neighbor in closed_list:
        continue

    # STEP 7: Calculate the G cost to this neighbor
    cost_to_neighbor = current.G + current.get_distance(neighbor)

    # STEP 8: Check if this path to neighbor is better
    is_in_open = neighbor in open_list

    if not is_in_open or cost_to_neighbor < neighbor.G:
        # This is the best path to this neighbor so far!
```

```
            # Update the neighbor's costs and parent
            neighbor.G = cost_to_neighbor
            neighbor.parent = current

            # If not in open list, calculate H and add it
            if not is_in_open:
                neighbor.H = neighbor.get_distance(goal_node)
                neighbor.F = neighbor.G + neighbor.H
                open_list.append(neighbor)

    # STEP 9: No path found
    return None


def reconstruct_path(goal_node):
    """
    Trace back from goal to start using parent pointers.

    Args:
        goal_node: The final node in the path

    Returns:
        List of nodes from start to goal
    """
    path = []
    current = goal_node
```

```
    # Follow the chain of parents back to start
    while current is not None:
        path.append(current)
        current = current.parent

    path.reverse()  # Path was built backwards, so reverse it
    return path
```

What Each Step Does

Step 1: Initialize Lists

```
open_list = [start_node]   # Frontier - nodes to explore
closed_list = []           # Already explored
```

Purpose: The open list contains nodes we might explore. The closed list tracks nodes we've already fully processed.

Step 2: Main Loop

```
while len(open_list) > 0:
```

Purpose: Continue until we've either found the goal or exhausted all possibilities.

Step 3: Find Best Node

```
current = open_list[0]
for node in open_list:
```

```
    if node.F < current.F or (node.F == current.F and node.H < current.H):
        current = node
```

Purpose:

- Select the node with the **lowest F score** (best estimated total cost)
- If F scores tie, prefer the node **closer to the goal** (lowest H)
- This is the "greedy" part that makes A* efficient

Why this matters: By always choosing the most promising node, A* focuses its search toward the goal rather than exploring randomly.

Step 4: Move to Closed List

```
closed_list.append(current)
open_list.remove(current)
```

Purpose: Mark this node as fully processed so we don't examine it again.

Step 5: Goal Check

```
if current == goal_node:
    return reconstruct_path(current)
```

Purpose: If we've reached the goal, we're done! Trace back the path and return it.

Why here: We check after selecting the node, ensuring we've found the optimal path (not just *a* path).

Step 6: Process Neighbors

```
for neighbor in current.neighbors:
    if not neighbor.walkable or neighbor in closed_list:
        continue
```

Purpose:

- Examine all adjacent positions
- Skip obstacles (not walkable)
- Skip already-processed nodes (in closed list)

Step 7: Calculate Cost

```
cost_to_neighbor = current.G + current.get_distance(neighbor)
```

Purpose: Calculate how much it would cost to reach this neighbor through the current node.

Components:

- `current.G` = cost to reach current node from start
- `get_distance(neighbor)` = cost to move from current to neighbor (usually 1)
- Sum = total cost to reach neighbor via this path

Step 8: Update or Add Neighbor

```
is_in_open = neighbor in open_list

if not is_in_open or cost_to_neighbor < neighbor.G:
    neighbor.G = cost_to_neighbor
    neighbor.parent = current

    if not is_in_open:
        neighbor.H = neighbor.get_distance(goal_node)
        neighbor.F = neighbor.G + neighbor.H
        open_list.append(neighbor)
```

Purpose: This is the core of A*'s path-finding logic:

If neighbor is NEW (not in open list):

- Set its G cost (distance from start)
- Set its H cost (estimated distance to goal)
- Calculate F = G + H (total estimated cost)
- Add to open list for future exploration
- Record current as its parent (for path reconstruction)

If neighbor is ALREADY in open list but we found a **better path**:

- Update its G cost to the lower value
- Update its parent to current
- Keep it in open list with new priority

Key Insight: A* is willing to reconsider nodes if it finds a cheaper route to them!

Step 9: Path Reconstruction

```
def reconstruct_path(goal_node):

    path = []

    current = goal_node



    while current is not None:

        path.append(current)

        current = current.parent



    path.reverse()

    return path
```

Purpose:

- Start at the goal
- Follow parent pointers back to start
- Reverse the list (since we built it backwards)
- Return the complete path

The Required Node Properties

For this algorithm to work, each node must have:

```
class Node:
    def __init__(self):
        self.G = 0              # Cost from start to this node
        self.H = 0              # Heuristic: estimated cost to goal
        self.F = 0              # Total: G + H
        self.parent = None      # Previous node in path
        self.neighbors = []     # Adjacent nodes
        self.walkable = True    # Can we traverse this node?

    def get_distance(self, other):
        """Calculate distance to another node (heuristic function)"""
        # For hex grids, use hex_distance()
        # For square grids, use Manhattan or Euclidean distance
        pass
```

Practice Exercises:

Exercise 1:

Part A:

Match each cost with its correct definition by writing the letter:

Costs:

1. G-cost
2. H-cost
3. F-cost

Definitions: A. The total estimated cost (G + H) B. Distance already traveled from start C. Estimated distance remaining to goal

Your answers:

- G-cost = _____
- H-cost = _____
- F-cost = _____

Part B:

You are playing a grid-based game. Your character started at position (0,0) and needs to reach a goal at position (3,3). You are currently at position (2,1). Each step costs 1 unit, and you can only move up, down, left, or right (no diagonals).

The path you took so far:

- Started at (0,0)
- Moved RIGHT to (1,0) — 1 step
- Moved RIGHT to (2,0) — 2 steps total
- Moved DOWN to (2,1) — 3 steps total ← **YOU ARE HERE**

Calculate the costs for your current position (2,1):

1. **G-cost (How many steps have you taken from start to current position?):**

   G-cost = _____ steps

2. **H-cost (How many steps remain to reach the goal at (3,3)?)**

   Hint: You need to move RIGHT once (to reach x=3) and DOWN twice (to reach y=3)

   H-cost = _____ steps

3. **F-cost (What is the total estimated cost?):**

   F-cost = G + H = _____ + _____ = _____ steps

Part C:

From your current position (2,1), A* considers three possible next moves:

- **Move RIGHT to (3,1):**
  G = 4, H = 2, F = 6
- **Move DOWN to (2,2):**
  G = 4, H = 2, F = 6
- **Move LEFT to (1,1):**
  G = 4, H = 4, F = 8

Answer these questions:

1. **Which node(s) have the lowest F-cost?**

   ____

2. *Which node will A explore next?**

   ☐ RIGHT to (3,1)
   ☐ DOWN to (2,2)
   ☐ Either RIGHT or DOWN (they're tied)
   ☐ LEFT to (1,1)

3. *Why doesn't A choose to move LEFT even though it's a valid move?**

   ____

   ____

Part D:

Scenario 1: Counting neighbors

You are at position (2,2) in the middle of a 5×5 grid with no obstacles. You can move in 4 directions (up, down, left, right).

1. **How many neighbors can you move to from position (2,2)?**

Number of neighbors = _____

2. **List all four neighbor positions:**
    - UP: (2, _____)
    - DOWN: (2, _____)
    - LEFT: (_____, 2)
    - RIGHT: (_____, 2)

Scenario 2: Blocked by a wall

You are still at position (2,2), but now there is a wall at position (2,3) directly below you.

3. **How many VALID neighbors can you move to now?**

    Number of valid neighbors = _____

4. **Explain why the wall at (2,3) matters:**

    ____

    ____

Scenario 3: Can a path exist?

Consider this situation:

- Start is at (0,0)
- Goal is at (2,0)
- There are walls at positions: (1,0) and (1,1)
- Grid is 3×2 (3 columns, 2 rows)

5. *Can A find a path from Start to Goal?**

    ☐ YES ☐ NO

6. **Explain your reasoning:**

    ____

    ____

    ____

Exercise 2:

Part A: True or False

For each statement, mark TRUE or FALSE:

*1. A always finds the shortest path between start and goal.**

☐ TRUE ☐ FALSE

2. The G-cost increases as you move farther away from the start position.

☐ TRUE ☐ FALSE

3. The H-cost is the exact, actual distance remaining to reach the goal.

☐ TRUE ☐ FALSE

*4. A explores the node with the lowest F-cost next.**

☐ TRUE ☐ FALSE

5. The formula for F-cost is: F = G × H

☐ TRUE ☐ FALSE

If FALSE, write the correct formula: ____________________________________

---

6. Manhattan distance measures straight-line distance "as the crow flies."

☐ TRUE ☐ FALSE

*7. A can determine when no path exists between start and goal.**

☐ TRUE ☐ FALSE

8. Walls have infinite movement cost because you cannot pass through them.

☐ TRUE ☐ FALSE

*9. A will always explore every single node on the grid before finding the goal.**

☐ TRUE ☐ FALSE

10. The H-cost is called a "heuristic" because it's an estimate, not an exact value.

☐ TRUE ☐ FALSE

Part B:

Scenario 1: Tower Defense Game

You're developing a tower defense game where enemies follow paths to reach the player's base. Players can build towers that act as walls to block enemy paths.

Questions:

1. **An enemy is halfway to the base when the player builds a tower blocking its current path. What should happen?**

   ☐ The enemy stops moving (stuck)
   ☐ A* recalculates a new path around the tower
   ☐ The enemy walks through the tower
   ☐ The game crashes

2. **What happens if the player completely surrounds the enemy spawn point with towers so no path to the base exists?**

   ____

   ____

Scenario 2: Strategy Game Performance

You're making a real-time strategy game like StarCraft. You have 100 units that all need to find paths across a large map at the same time. The game starts lagging badly.

Questions:

3. **What is most likely causing the performance problem?**

   ☐ Too many graphics on screen
   ☐ Running 100 A* pathfinding calculations simultaneously
   ☐ The map is too colorful
   ☐ Players are clicking too fast

4. **Suggest ONE way to improve performance:**

   ____

   ____

5. **Would it make sense to calculate paths for all 100 units every single frame (60 times per second)?**

   ☐ YES ☐ NO

   Why or why not? ____

Scenario 3: Choosing the Right Heuristic

You need to implement A* for two different games:

Game A: Pac-Man

- Grid-based maze

- Movement: Only up, down, left, right (NO diagonals)
- Must follow grid lines

Game B: Top-Down Adventure Game

- Open terrain
- Movement: Can move in 8 directions including diagonals
- Smooth diagonal movement allowed

Questions:

6. **For Pac-Man, which heuristic is better?**

   ☐ Manhattan distance ($|x1-x2| + |y1-y2|$)
   ☐ Euclidean distance ($\sqrt{[(x1-x2)^2 + (y1-y2)^2]}$)

   Why? ____

7. **For the Adventure Game, which heuristic is better?**

   ☐ Manhattan distance ($|x1-x2| + |y1-y2|$)
   ☐ Euclidean distance ($\sqrt{[(x1-x2)^2 + (y1-y2)^2]}$)

   Why? ____

Answer Key

Exercise 1:

Part A:

- G-cost = B (Distance already traveled from start)
- H-cost = C (Estimated distance remaining to goal)
- F-cost = A (The total estimated cost, G + H)

Part B:

1. G-cost = 3 steps
2. H-cost = 3 steps (1 right + 2 down)
3. F-cost = 6 steps

Part C:

1. RIGHT to (3,1) and DOWN to (2,2) both have F=6
2. Either RIGHT or DOWN (they're tied)

3. Moving LEFT has a higher F-cost (8 vs 6), so it's less promising

Part D:

1. 4 neighbors
2. UP: (2,1), DOWN: (2,3), LEFT: (1,2), RIGHT: (3,2)
3. 3 valid neighbors
4. The wall blocks movement in that direction
5. NO - the wall at (1,0) blocks the direct path, and the wall at (1,1) blocks going around
6. The goal is completely blocked by walls

Exercise 2:

Part A:

1. TRUE
2. TRUE
3. FALSE (H is an estimate, not exact)
4. TRUE
5. FALSE (F = G + H, not G × H)
6. FALSE (that's Euclidean; Manhattan is grid-based)
7. TRUE
8. TRUE
9. FALSE (A* explores efficiently, not exhaustively)
10. TRUE

Part B:

1. A* recalculates a new path around the tower
2. A* will determine no path exists; enemy cannot reach the base
3. Running 100 A* pathfinding calculations simultaneously
4. Possible answers: stagger pathfinding over multiple frames, cache paths, use hierarchical pathfinding, group units with similar paths
5. NO - recalculating 100 paths 60 times per second is wasteful; only recalculate when obstacles change or destination changes
6. Manhattan distance - matches 4-directional grid movement
7. Euclidean distance - better matches diagonal movement capabilities

Core References

For Beginners:
Patel, A. (2023). "Introduction to A*." *Red Blob Games*.
https://www.redblobgames.com/pathfinding/a-star/introduction.html
Lester, P. (2005). "A* Pathfinding for Beginners." *GameDev.net*.

Textbook References:
Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Chapter 3.
Millington, I., & Funge, J. (2009). *Artificial Intelligence for Games* (2nd ed.). Chapter 4.

Original Paper:
Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). "A Formal Basis for Heuristic Search." *IEEE Transactions*.

Algorithm Source:
Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.

Implementation Style:
Lester, P. (2005). "A* Pathfinding for Beginners." *GameDev.net*.
https://www.gamedev.net/tutorials/programming/artificial-intelligence/a-pathfinding-for-beginners-r2003/