A* Pathfinding Algorithm: Problem Set with Solutions

Part 1: Conceptual Understanding

Problem 1.1: Algorithm Comparison

Question: Fill in the table comparing pathfinding algorithms:

Algorithm	Uses g(n)?	Uses h(n)?	Guarantees Optimal Path?	Typical Performance
BFS	?	?	?	?
Dijkstra	?	?	?	?
Greedy Best-First	?	?	?	?
A*	?	?	?	?

Answer:

Algorithm	Uses g(n)?	Uses h(n)?	Guarantees Optimal Path?	Typical Performance
BFS	Implicit (depth)	No	Yes (unweighted)	O(V + E)
Dijkstra	Yes	No	Yes	O((V + E) log V)
Greedy Best-First	No	Yes	No	O(b^d) worst, often fast
A*	Yes	Yes	Yes (if h admissible)	O(E log V), better in practice

Explanation:

- BFS treats all edges as equal cost, essentially using depth as g(n)
- Dijkstra is A* with h(n) = 0
- Greedy only considers estimated distance, can get trapped
- A* combines both for optimal efficiency with optimality guarantee

---

Problem 1.2: Heuristic Admissibility

Question: For a grid with 4-directional movement (up, down, left, right), which of these heuristics are admissible? Explain why or why not.

a) h(n) = |n.x - goal.x| + |n.y - goal.y| (Manhattan) b) h(n) = $\sqrt{}$[(n.x - goal.x)^2 + (n.y - goal.y)^2] (Euclidean) c) h(n) = 2 × (|n.x - goal.x| + |n.y - goal.y|) d) h(n) = max(|n.x - goal.x|, |n.y - goal.y|) (Chebyshev) e) h(n) = 0

Answer:

a) **Admissible** ✓ - Manhattan distance equals the actual minimum steps needed for 4-directional movement. It never overestimates.

b) **Admissible** ✓ - Euclidean distance is the straight-line distance, which is always ≤ the actual path length (we can't walk through walls diagonally). It underestimates for grid movement.

c) **Not Admissible** <#2717> - This doubles the Manhattan distance, which will overestimate. If the true distance is 5 steps, this returns 10.

d) **Admissible** ✓ - Chebyshev distance is appropriate for 8- directional movement but underestimates for 4-directional, making it admissible (though not the best choice).

e) **Admissible** ✓ - Zero always underestimates (unless you're already at the goal). This reduces A* to Dijkstra's algorithm.

Key Principle: A heuristic is admissible if h(n) ≤ actual_cost(n, goal) for all n.

Problem 1.3: Trace the Algorithm

Question: Given this 5×5 grid where S=start, G=goal, #=wall:

```
S . . . .
. # # # .
. . . # .
. # . . .
. . . . G
```

Using Manhattan distance heuristic, show the first 5 nodes explored by A* (in order). For each node, calculate f(n) = g(n) + h(n).

Answer:

Starting position: S at (0,0), Goal: G at (4,4)

Step 1: Start at S(0,0)

- g(0,0) = 0
- h(0,0) = |0-4| + |0-4| = 8
- f(0,0) = 0 + 8 = 8
- Neighbors: (1,0) and (0,1)

Step 2: Explore (1,0) [right of start]

- g(1,0) = 1
- h(1,0) = |1-4| + |0-4| = 7
- f(1,0) = 1 + 7 = 8

Add to open set: (0,1) with f=8

Step 3: Explore (0,1) [down from start]

- g(0,1) = 1
- h(0,1) = |0-4| + |1-4| = 7
- f(0,1) = 1 + 7 = 8

Both have f=8, choose by insertion order: (1,0)'s neighbors next

Step 4: From (1,0), explore (2,0)

- g(2,0) = 2

- h(2,0) = |2-4| + |0-4| = 6
- f(2,0) = 2 + 6 = 8

Step 5: From (0,1), explore (0,2)

- g(0,2) = 2
- h(0,2) = |0-4| + |2-4| = 6
- f(0,2) = 2 + 6 = 8

Exploration order: S(0,0) → (1,0) → (0,1) → (2,0) → (0,2)

Note: All early nodes have f=8 because we're moving away from goal in one dimension while approaching in another. The algorithm explores uniformly until the heuristic can guide it.

Part 2: Mathematical Analysis

Problem 2.1: Heuristic Design

Question: You're implementing pathfinding for a strategy game where units can move:

- Cardinal directions (N, S, E, W): cost = 1
- Diagonal directions (NE, NW, SE, SW): cost = 1.4 (approximately $\sqrt{2}$)

Design an admissible heuristic for this movement model. Prove it never overestimates.

Answer:

Diagonal Distance (Octile Distance) Heuristic:

```
dx = |n.x - goal.x|
dy = |n.y - goal.y|
h(n) = max(dx, dy) + (√2 - 1) × min(dx, dy)
```

Alternatively: `h(n) = min(dx, dy) × √2 + |dx - dy|`

Proof of Admissibility:

The optimal path will use diagonal moves for min(dx, dy) steps, then straight moves for |dx - dy| steps.

Actual cost:

- Diagonal steps: min(dx, dy) × 1.4
- Straight steps: |dx - dy| × 1

Total: 1.4 × min(dx, dy) + |dx - dy|

Our heuristic:

- max(dx, dy) + ($\sqrt{2}$ - 1) × min(dx, dy)

- = max(dx, dy) + 0.414... × min(dx, dy)
- ≈ max(dx, dy) + 0.4 × min(dx, dy)

Since we use $\sqrt{2} \approx 1.414$ instead of the actual movement cost (1.4), and the formula structure matches optimal movement, this heuristic is admissible.

For game implementation using cost 1.4: `h(n) = max(dx, dy) + 0.4 × min(dx, dy)`

Problem 2.2: Complexity Analysis

Question:

a) What is the time complexity of A* in the worst case for a grid of size N×N? b) If we use a linear search to find the minimum f-score (as in our implementation), what's the complexity per iteration? c) How would using a binary min-heap improve this?

Answer:

a) Worst Case Time Complexity:

In the worst case, A* explores every node in the grid:

- N^2 nodes total
- Each node can be added/removed from open set: $O(N^2)$ operations
- Each node has up to 4 neighbors (grid): checking/updating neighbors: $O(4) = O(1)$

With efficient priority queue: $\mathbf{O(N^2 \log N^2) = O(N^2 \log N)}$

b) Linear Search per Iteration:

With linear search for minimum f-score:

- Open set can contain up to $O(N^2)$ nodes
- Finding minimum: $O(N^2)$ per iteration
- Total: $O(N^2)$ iterations × $O(N^2)$ per search = $\mathbf{O(N^4)}$

This is why our educational implementation works for small grids (20×15 = 300 cells) but wouldn't scale.

c) Binary Min-Heap Improvement:

With binary heap:

- Extract minimum: O(log n) where n is open set size
- Insert/update node: O(log n)
- Total: $O(N^2)$ operations × $O(\log N^2)$ = $\mathbf{O(N^2 \log N)}$

This is an $O(N^2)$ improvement factor! For a 100×100 grid:

- Linear: $10{,}000^4 = 10^{16}$ operations (infeasible)
- Heap: $10{,}000^2 \times \log(10{,}000) \approx 10^8$ operations (manageable)

Problem 2.3: Heuristic Strength

Question: Two heuristics are admissible:

- $h_1(n)$ = Manhattan distance
- $h_2(n)$ = Euclidean distance

For 4-directional grid movement, which is "stronger" (explores fewer nodes)? Why?

Answer:

h_1 (Manhattan) is stronger for 4-directional movement.

Reasoning:

A heuristic h_1 is stronger (dominates) h_2 if $h_1(n) \geq h_2(n)$ for all n, and both are admissible.

For any point (x, y) going to goal (gx, gy):

- Manhattan: $h_1 = |x - gx| + |y - gy|$
- Euclidean: $h_2 = \sqrt{[(x - gx)^2 + (y - gy)^2]}$

Example: From (0,0) to (3,4):

- $h_1 = |0-3| + |0-4| = 3 + 4 = 7$
- $h_2 = \sqrt{(3^2 + 4^2)} = \sqrt{25} = 5$

We see $h_1 > h_2$, so Manhattan is stronger.

Why this matters:

A stronger (higher) admissible heuristic provides better guidance:

- Higher $h(n) \rightarrow$ higher $f(n)$ for non-optimal paths
- Better pruning of unpromising branches
- Fewer nodes explored

For 4-directional movement, the actual shortest path is exactly the Manhattan distance (in absence of obstacles), so h_1 is not just stronger but **perfect** (equals true cost in obstacle-free space).

Counterintuitive Result: Euclidean distance, despite being geometrically "more accurate," is weaker for grid pathfinding because it underestimates more severely.

Part 3: Implementation & Debugging

Problem 3.1: Bug Hunt

Question: A student implements A* but gets suboptimal paths. Here's their code (in pseudocode):

```
function A_Star(start, goal):
    openSet = [start]
    closedSet = []
    start.g = 0
    start.h = heuristic(start, goal)
    start.f = start.h

    while openSet not empty:
        current = node in openSet with minimum f

        if current == goal:
            return reconstructPath(current)

        remove current from openSet
        add current to closedSet

        for each neighbor of current:
            if neighbor in closedSet:
                continue

            tentative_g = current.g + 1

            if neighbor not in openSet:
                add neighbor to openSet

            neighbor.g = tentative_g
            neighbor.h = heuristic(neighbor, goal)
            neighbor.f = neighbor.g + neighbor.h
            neighbor.parent = current

    return null  // No path found
```

Identify the bug and explain why it causes suboptimal paths.

Answer:

Bug Location:

```
if neighbor not in openSet:
    add neighbor to openSet

neighbor.g = tentative_g
neighbor.h = heuristic(neighbor, goal)
neighbor.f = neighbor.g + neighbor.h
neighbor.parent = current
```

Problem: The code always updates the neighbor's g-score, h-score, f-score, and parent, even if the neighbor was already in the open set with a better path.

Correct Implementation:

```
if neighbor not in openSet:
    add neighbor to openSet
elif tentative_g >= neighbor.g:
    continue  // This path is not better

neighbor.g = tentative_g
neighbor.h = heuristic(neighbor, goal)
neighbor.f = neighbor.g + neighbor.h
neighbor.parent = current
```

Or more commonly:

```
if neighbor not in openSet:
    add neighbor to openSet
elseif tentative_g >= neighbor.g:
    continue

// Only update if we found a better path
neighbor.g = tentative_g
neighbor.f = neighbor.g + neighbor.h
neighbor.parent = current
```

Why This Causes Suboptimal Paths:

1. Suppose we reach cell C via two paths:
    - Path A: start → X → C (g = 5)
    - Path B: start → Y → C (g = 7)
2. If Path A reaches C first, C.g = 5, C.parent = X
3. Later, Path B reaches C. The buggy code unconditionally sets:
    - C.g = 7 (worse!)
    - C.parent = Y (wrong!)
4. When reconstructing the path, we follow the suboptimal parent pointer.

Correct Behavior: Only update neighbor if tentative_g < neighbor.g (we found a better path).

Problem 3.2: Feature Implementation

Question: Modify the pathfinding algorithm to support **weighted terrain**. Grass costs 1, Water costs 3, Mud costs 2. Show the changes needed to:

a) The cell data structure b) The pathfinding algorithm c) The heuristic function

Answer:

a) Cell Data Structure:

```
;; Original
{:x x :y y :walkable true :g 0 :h 0 :f 0 :parent nil}

;; Modified
{:x x :y y :walkable true
 :terrain :grass  ;; :grass, :water, :mud, :wall
 :g 0 :h 0 :f 0 :parent nil}
```

```
;; Cost lookup
(fn getTerrainCost [terrain]
  (match terrain
    :grass 1
    :mud 2
    :water 3
    :wall math.huge  ;; Unwalkable
    _ 1))  ;; Default
```

b) Algorithm Changes:

Original:

```
(local tentativeG (+ (. current :g) 1))
```

Modified:

```
(local moveCost (getTerrainCost (. neighbor :terrain)))
(local tentativeG (+ (. current :g) moveCost))
```

Full context:

```
(each [_ neighbor (ipairs neighbors)]
  (var inClosed false)
  (each [_ c (ipairs self.closedSet)]
    (when (= c neighbor)
      (set inClosed true)))

  (when (not inClosed)
    ;; Changed this line:
    (local tentativeG (+ (. current :g) (getTerrainCost (. neighbor :terrain))))

    (var inOpen false)
    (each [_ o (ipairs self.openSet)]
      (when (= o neighbor)
        (set inOpen true)))

    (when (or (not inOpen) (< tentativeG (. neighbor :g)))
      (tset neighbor :parent current)
      (tset neighbor :g tentativeG)
      (tset neighbor :h (self:heuristic neighbor self.goal))
      (tset neighbor :f (+ (. neighbor :g) (. neighbor :h)))

      (when (not inOpen)
        (table.insert self.openSet neighbor)))))
```

c) Heuristic Adjustment:

The heuristic should be scaled by the minimum terrain cost to remain admissible:

```
(fn Grid.heuristic [self a b]
  (local minCost 1)  ;; Minimum terrain cost (grass)
  (local manhattanDist (+ (math.abs (- (. a :x) (. b :x)))
                          (math.abs (- (. a :y) (. b :y)))))
  (* minCost manhattanDist))
```

Why multiply by minCost?

- If all terrain costs are ≥1, Manhattan distance underestimates
- If minCost = 1, this is unchanged (admissible)
- If minCost = 3, multiply by 3 (still admissible, stronger heuristic)

Advanced: For even better heuristics, if you know the terrain distribution:

```
;; If path will likely traverse mostly water (cost 3):
(* 3 manhattanDist)  ;; More accurate estimate

;; If uncertain, stay conservative:
(* 1 manhattanDist)  ;; Always admissible
```

Problem 3.3: Optimization Challenge

Question: The current implementation uses linear search to find the minimum f-score node. Implement a simple binary min-heap to improve this. Provide the heap operations needed:

a) `heap_insert(heap, node)` b) `heap_extract_min(heap)` c) How does this change the algorithm's performance?

Answer:

a) Heap Insert (Bubble Up):

```
(fn heap_insert [heap node]
  ;; Add to end of array
  (table.insert heap node)
  (local idx (length heap))

  ;; Bubble up
  (var i idx)
  (while (> i 1)
    (local parent (math.floor (/ i 2)))
    (when (<= (. (. heap i) :f) (. (. heap parent) :f))
      ;; Swap with parent
      (local temp (. heap parent))
      (tset heap parent (. heap i))
      (tset heap i temp)
      (set i parent))
    (else
      (lua "break"))))
```

b) Extract Minimum (Bubble Down):

```
(fn heap_extract_min [heap]
  (when (= (length heap) 0)
    (lua "return nil"))

  (local min (. heap 1))
  (local last (table.remove heap))

  (when (> (length heap) 0)
    (tset heap 1 last)

    ;; Bubble down
    (var i 1)
    (var done false)
    (while (not done)
      (local left (* i 2))
      (local right (+ (* i 2) 1))
      (var smallest i)

      (when (and (<= left (length heap))
                 (< (. (. heap left) :f) (. (. heap smallest) :f)))
        (set smallest left))

      (when (and (<= right (length heap))
                 (< (. (. heap right) :f) (. (. heap smallest) :f)))
        (set smallest right))

      (if (not= smallest i)
          (do
            ;; Swap
            (local temp (. heap i))
            (tset heap i (. heap smallest))
            (tset heap smallest temp)
            (set i smallest))
          (set done true))))

  min)
```

c) Performance Change:

Before (Linear Search):

- Find minimum: O(n) where n = open set size
- Per iteration: O(n)
- Total: O(V × V) = O(V^2) where V = number of vertices

After (Min-Heap):

- Extract minimum: O(log n)
- Insert: O(log n)
- Per iteration: O(log n)
- Total: O(V log V) for all extractions

Concrete Example: For 20×15 grid (300 cells):

- Linear: ~300 comparisons per iteration × 300 iterations = 90,000 operations
- Heap: ~$\log_2(150) \approx$ 7-8 comparisons per iteration × 300 iterations = 2,400 operations

~37× speedup!

For 100×100 grid (10,000 cells):

- Linear: $10{,}000^2 = 100{,}000{,}000$ operations
- Heap: $10{,}000 \times \log_2(5000) \approx 120{,}000$ operations

~833× speedup!

Practical Note: For very small grids (<100 cells), linear search might actually be faster due to lower constant factors and better cache locality. Always profile!

Part 4: Advanced Applications

Problem 4.1: Moving Target

Question: In a real-time strategy game, units need to pathfind to enemy units that are moving. How would you modify A* to handle a moving goal? Consider:

a) The naive approach and its problems b) A better solution using replanning c) When to trigger replanning

Answer:

a) Naive Approach:

Recompute entire path from scratch every frame:

```
(fn love.update [dt]
  (when unit.hasTarget
    (local path (A_star unit.position target.position))
    (unit:followPath path)))
```

Problems:

1. **Wasteful:** Most of the path hasn't changed
2. **Expensive:** Full A* search every frame (60 FPS = 60 searches/second)
3. **Janky movement:** Path changes cause units to stutter
4. **Scalability:** 100 units × 60 FPS = 6,000 pathfinding operations/second

b) Better Solution: D *Lite (Incremental Replanning):**

D* Lite efficiently updates paths when the goal moves:

```
;; Pseudocode for D* Lite concept
(fn initialize_dstar [unit target]
  {:unit unit
   :target target
   :path (A_star unit.pos target.pos)
   :last_target_pos target.pos})

(fn update_with_moving_target [state]
  (local target_moved_dist
    (distance state.last_target_pos state.target.pos))
  (when (> target_moved_dist replan_threshold)
    ;; Instead of full replan, update affected nodes
    (local affected_nodes (get_nodes_affected_by_goal_move state))
    (each [_ node (ipairs affected_nodes)]
      (update_node_cost node state.target.pos))

    (set state.last_target_pos state.target.pos)))
```

Key Insight: Only nodes near the old and new goal positions need cost updates. Most of the path remains valid.

c) Replanning Triggers:

```
(fn should_replan [unit]
  (or
    ;; Target moved significantly
    (> (distance unit.target.pos unit.last_target_pos) 5)

    ;; Path is blocked (new obstacle)
    (not (is_walkable (next_cell_in_path unit.path)))

    ;; Periodic refresh (every N seconds)
    (> unit.time_since_replan 2.0)

    ;; Target velocity changed (predictive)
    (> (vector_diff unit.target.velocity unit.target.last_velocity)
       threshold)))
```

Advanced Techniques:

1. **Prediction:** Path to where target will be, not where it is

```
(local predicted_pos
  (+ target.pos (* target.velocity look_ahead_time)))
```

1. **Hybrid Approach:** Global path to approximate region, local steering for final approach

```
(local region (get_region target.pos))
(when (not= region unit.target_region)
  (replan_global unit region))
;; Always do local steering
(unit:steer_toward target.pos)
```

1. **Influence Maps:** Pre-compute "flow field" pointing toward moving target's likely positions

```
;; Update flow field when target moves
(when target_moved
  (update_flow_field_region target.pos radius))
;; Units just follow the field
(unit:follow_flow_field)
```

Problem 4.2: Multi-Agent Pathfinding

Question: In a tower defense game, 50 enemies need to pathfind to the same goal (the player's base). What optimization techniques can avoid computing 50 separate A* paths?

Answer:

Solution: Flow Fields (Vector Fields)

Instead of individual paths, compute a single direction field that all units follow.

Algorithm:

```
(fn compute_flow_field [grid goal]
  ;; Step 1: Dijkstra from goal (backward search)
  (local cost_field (dijkstra_from_goal grid goal))

  ;; Step 2: Compute flow field (gradient descent)
  (local flow_field [])
  (for [y 1 grid.rows]
    (for [x 1 grid.cols]
      (local cell (grid:getCell x y))
      (local neighbors (grid:getNeighbors cell))

      ;; Find neighbor with lowest cost
      (var best_neighbor nil)
      (var lowest_cost math.huge)
      (each [_ n (ipairs neighbors)]
        (when (< (. cost_field (cell_index n)) lowest_cost)
          (set lowest_cost (. cost_field (cell_index n)))
          (set best_neighbor n)))

      ;; Direction = vector toward best neighbor
      (when best_neighbor
        (local direction
          {:x (- (. best_neighbor :x) (. cell :x))
           :y (- (. best_neighbor :y) (. cell :y))})
        (tset flow_field (cell_index cell) direction))))

  flow_field)
```

Usage:

```
;; Compute once
(local flow_field (compute_flow_field grid goal))

;; All units use it
(fn Enemy.update [self dt]
  (local cell (grid:getCellAt self.x self.y))
  (local direction (. flow_field (cell_index cell)))

  (set self.vx (* direction.x self.speed))
  (set self.vy (* direction.y self.speed))

  (set self.x (+ self.x (* self.vx dt)))
  (set self.y (+ self.y (* self.vy dt))))
```

Benefits:

1. **One-time cost:** Single Dijkstra instead of N × A*
2. **Constant-time movement:** Each unit just looks up its cell
3. **Memory efficient:** One vector per cell vs. one path per unit
4. **Smooth motion:** Natural flow around obstacles

Performance Comparison:

For 50 units on 50×50 grid:

Individual A*:*

- 50 paths × 2,500 cells = 125,000 cell evaluations
- Per frame if paths invalidate: 125,000 × 60 FPS = 7.5M ops/sec

Flow Field:

- 1 Dijkstra × 2,500 cells = 2,500 cell evaluations (one-time)
- 50 lookups/frame = 50 × 60 FPS = 3,000 ops/sec
- **2,500× more efficient!**

When to Recompute:

```
(var flow_field_dirty false)

;; Mark dirty when obstacles change
(fn add_obstacle [grid x y]
  (local cell (grid:getCell x y))
  (set cell.walkable false)
  (set flow_field_dirty true))

;; Recompute only when needed
(fn love.update [dt]
  (when flow_field_dirty
    (set flow_field (compute_flow_field grid goal))
    (set flow_field_dirty false))

  ;; All units update using cached field
  (each [_ enemy (ipairs enemies)]
    (enemy:update dt)))
```

Limitations:

- All units share same goal (good for tower defense, bad for diverse objectives)
- Static goal (need replanning if goal moves frequently)
- Requires Dijkstra preprocessing

Hybrid Solution for Multiple Goals:

```
(local flow_fields {})  ;; Cache multiple fields

(fn Enemy.update [self dt]
  (when (not (. flow_fields self.goal_id))
    ;; Compute on-demand
    (tset flow_fields self.goal_id
      (compute_flow_field grid self.goal)))

  (local field (. flow_fields self.goal_id))
  ;; Follow field...
  )
```

Problem 4.3: Hierarchical Pathfinding

Question: Design a hierarchical pathfinding system for a large 200×200 grid. Describe:

a) How to partition the map into regions b) The two-level pathfinding algorithm c) When this is beneficial vs. standard A*

Answer:

a) Map Partitioning:

Divide the 200×200 grid into 10×10 regions (each region is 20×20 cells):

```
(fn create_region_graph [grid]
  (local region_size 20)
  (local regions [])

  ;; Create regions
  (for [ry 0 9]  ;; 10 regions vertically
    (for [rx 0 9]  ;; 10 regions horizontally
      (local region
        {:id (+ (* ry 10) rx)
         :x rx :y ry
         :min_x (* rx region_size)
         :min_y (* ry region_size)
         :max_x (+ (* rx region_size) region_size -1)
         :max_y (+ (* ry region_size) region_size -1)
         :entrances []  ;; Border cells to other regions
         :walkable true})  ;; Has any walkable cells?

      ;; Check if region is blocked
      (var has_walkable false)
      (for [y region.min_y region.max_y]
        (for [x region.min_x region.max_x]
          (when (. (grid:getCell (+ x 1) (+ y 1)) :walkable)
            (set has_walkable true))))
      (set region.walkable has_walkable)

      (table.insert regions region)))

  ;; Find entrances between regions
  (each [_ region (ipairs regions)]
    (when region.walkable
      ;; Check borders with neighbors
      (find_entrances grid region regions)))

  regions)

(fn find_entrances [grid region all_regions]
  ;; Check right border
  (when (< region.x 9)
    (local neighbor (get_region all_regions (+ region.x 1) region.y))
    (when neighbor.walkable
      (for [y region.min_y region.max_y]
        (local cell (grid:getCell (+ region.max_x 1) (+ y 1)))
        (local next (grid:getCell (+ region.max_x 2) (+ y 1)))
        (when (and (. cell :walkable) (. next :walkable))
          (table.insert region.entrances
            {:cell cell :to_region neighbor.id})))))

  ;; Similar for bottom, left, top borders...
  )
```

b) Two-Level Algorithm:

```
(fn hierarchical_pathfind [grid regions start goal]
  ;; LEVEL 1: High-level region path
  (local start_region (find_region regions start))
  (local goal_region (find_region regions goal))

  (local region_path
    (A_star_regions start_region goal_region regions))

  ;; LEVEL 2: Low-level detailed path through each region
  (local detailed_path [])

  (for [i 1 (- (length region_path) 1)]
    (local current_region (. region_path i))
    (local next_region (. region_path (+ i 1)))

    ;; Find entrance to next region
    (local entrance (find_entrance current_region next_region))

    ;; Path within current region
    (local local_start (if (= i 1) start
                           (. detailed_path (length detailed_path))))
    (local local_goal entrance.cell)

    (local segment (A_star grid local_start local_goal))
    (append_path detailed_path segment))

  ;; Final segment to goal
  (local last_segment
    (A_star grid
      (. detailed_path (length detailed_path))
      goal))
  (append_path detailed_path last_segment)

  detailed_path)
```

c) When Hierarchical is Beneficial:

Beneficial when:

1. **Large maps:** 200×200 grid = 40,000 cells
   - Standard A*: explores thousands of cells
   - Hierarchical: 100 regions → region path might be 5-10 regions → only search within those regions
2. **Long-distance paths:** Crossing entire map
   - Standard A*: O(40,000) cells in worst case
   - Hierarchical: O(100) regions + O(400) cells per region × 10 regions = ~4,100 cells
3. **Multiple queries:** If pathfinding frequently
   - Preprocessing cost amortized across many queries
   - Region connectivity rarely changes

Not beneficial when:

1. **Short distances:** Within single region
   - Overhead of hierarchical system wasted
   - Better to use standard A* directly
2. **Dynamic environments:** Obstacles change frequently
   - Region graph needs frequent updates
   - Invalidates cached connectivity
3. **Small maps:** <50×50 grids
   - Overhead exceeds benefits
   - Standard A* is fast enough

Performance Analysis:

For 200×200 grid, start to goal across entire map:

Standard A*:*

```
Worst case: 40,000 cells explored
Average case: ~15,000 cells (with good heuristic)
Time: 15,000 × 100ns = 1.5ms
```

Hierarchical (10×10 regions of 20×20):

```
Level 1 (region): 100 regions, explore ~30 = 30 × 50ns = 1.5µs
Level 2 (detailed): 10 regions × 400 cells × 30% = 1,200 cells
```

```
Time: 1.5μs + 1,200 × 100ns = 120μs = 0.12ms

Speedup: 1.5ms / 0.12ms = 12.5× faster!
```

Implementation Tips:

```
;; Cache region connectivity
(local region_graph
  {:nodes regions
   :edges (compute_region_adjacency regions)})

;; Quick rejection
(fn quick_path_check [start goal]
  (when (same_region? start goal)
    ;; Skip hierarchical, use direct A*
    (return (A_star grid start goal)))

  ;; Use hierarchical for cross-region
  (hierarchical_pathfind grid regions start goal))
```

Part 5: Game Design & Optimization

Problem 5.1: Performance Budget

Question: You're working on a mobile RTS game that needs to maintain 60 FPS. You have a performance budget of 2ms per frame for all pathfinding. Given:

- 30 units that path per frame
- Grid size: 100×100
- Each A* search averages 800 cell evaluations
- Each cell evaluation: 200 CPU cycles
- Mobile CPU: 1.5 GHz

Will you meet the budget? If not, propose three optimizations with estimated improvements.

Answer:

Current Performance:

```
Calculations per frame:
- 30 units × 800 cells × 200 cycles = 4,800,000 cycles

Time per frame:
- 4,800,000 cycles ÷ 1,500,000,000 cycles/sec = 3.2ms

Verdict: OVER BUDGET by 1.2ms (160%)
```

Three Optimization Strategies:

1. Time-Slicing (Budget: 0.5ms)

Spread pathfinding across multiple frames:

```
(local pathfinding_queue [])
(local max_searches_per_frame 10)  ;; Instead of 30

(fn request_path [unit start goal callback]
  (table.insert pathfinding_queue
    {:unit unit :start start :goal goal :callback callback}))

(fn love.update [dt]
  (var searches_this_frame 0)
  (while (and (> (length pathfinding_queue) 0)
              (< searches_this_frame max_searches_per_frame))
    (local req (table.remove pathfinding_queue 1))
    (local path (A_star grid req.start req.goal))
    (req.callback path)
    (set searches_this_frame (+ searches_this_frame 1))))
```

Improvement:

- 10 units/frame instead of 30
- 3.2ms → 1.07ms (67% reduction)
- **But:** Units wait 1-3 frames for paths

2. Flow Fields for Groups (Budget: 0.2ms)

When 5+ units share destination, use flow field:

```
(local active_flow_fields {})  ;; goal_pos → flow_field

(fn request_path_optimized [unit start goal]
  (local goal_key (.. goal.x ":" goal.y))

  ;; Count units heading to same goal
  (local units_to_goal
    (count_units_with_goal goal))

  (if (>= units_to_goal 5)
      ;; Use/create flow field
      (do
        (when (not (. active_flow_fields goal_key))
          (tset active_flow_fields goal_key
            (compute_flow_field grid goal)))
        (unit:use_flow_field goal_key))

      ;; Individual A*
      (request_path unit start goal
        (fn [path] (unit:follow_path path)))))
```

Improvement:

- 20 units use flow field (5 groups of 4)
- 10 units use A*

- Cost: 5 flow fields (one-time) + 10 A* = 1.5ms + 1.07ms = 2.57ms
- After first frame (fields cached): 1.07ms
- **Improvement: 67% reduction after warmup**

3. Hierarchical Pathfinding (Budget: 0.8ms)

For 100×100 grid, use 5×5 regions (20×20 each):

```
(local region_system (create_regions grid 20))

(fn optimized_pathfind [start goal]
  ;; Quick same-region check
  (when (same_region? start goal)
    (return (A_star grid start goal)))

  ;; Hierarchical for long paths
  (hierarchical_pathfind grid region_system start goal))
```

Improvement:

- Average cells evaluated: 800 → 250 (hierarchical)
- Time: 3.2ms → 1.0ms (69% reduction)

Combined Strategy:

```
;; Hybrid approach
(fn smart_pathfind [unit start goal]
  ;; 1. Check for flow field opportunity
  (when (can_use_flow_field? goal)
    (return (use_flow_field unit goal)))

  ;; 2. Use hierarchical for long paths
  (when (> (manhattan_distance start goal) 50)
    (queue_hierarchical_path unit start goal))

  ;; 3. Direct A* for short paths
  (queue_astar unit start goal))

;; Time slice all queued requests
(fn process_pathfinding_budget [max_time]
  (var time_used 0)
  (while (and (< time_used max_time)
              (has_queued_requests?))
    (local start_time (love.timer.getTime))
    (process_next_request)
    (set time_used (+ time_used (- (love.timer.getTime) start_time)))))
```

Final Performance:

```
Best case (flow fields): 0.2ms (90% improvement)
Average case (mixed): 1.2ms (63% improvement)
Worst case (all hierarchical): 1.5ms (53% improvement)

MEETS BUDGET: ✓
```

Problem 5.2: Dynamic Obstacles

Question: Players can build walls during gameplay, invalidating existing paths. Design a system that efficiently updates affected paths without full recomputation. Consider 50 units with active paths.

Answer:

Solution: Incremental Path Repair with Lazy Revalidation

```
(local PathManager
  {:active_paths {}    ;; unit_id → {path, current_step, valid}
   :dirty_regions {}}) ;; regions needing revalidation

(fn PathManager.assign_path [self unit path]
  (tset self.active_paths unit.id
    {:path path
     :current_step 1
     :valid true
     :path_cells (build_cell_set path)}))  ;; For fast lookup

(fn PathManager.on_obstacle_added [self cell]
  ;; Mark paths that cross this cell as invalid
  (each [unit_id path_info (pairs self.active_paths)]
    (when (. path_info.path_cells (cell_key cell))
      (set path_info.valid false)

      ;; Add to repair queue
      (when (not (. self.repair_queue unit_id))
        (table.insert self.repair_queue unit_id)))))

(fn PathManager.update [self dt]
  ;; Process repair queue (time-sliced)
  (local max_repairs_per_frame 5)
  (var repairs_done 0)

  (while (and (< repairs_done max_repairs_per_frame)
              (> (length self.repair_queue) 0))
    (local unit_id (table.remove self.repair_queue 1))
    (self:repair_path unit_id)
    (set repairs_done (+ repairs_done 1)))

  ;; Update all units
  (each [unit_id path_info (pairs self.active_paths)]
    (local unit (get_unit unit_id))
    (when path_info.valid
      (unit:follow_path path_info.path path_info.current_step))))

(fn PathManager.repair_path [self unit_id]
  (local path_info (. self.active_paths unit_id))
  (local unit (get_unit unit_id))

  ;; Strategy 1: Try local repair first
  (local repaired (self:try_local_repair unit path_info))
  (if repaired
      (set path_info.valid true)

      ;; Strategy 2: Full replan
      (do
        (local new_path (A_star grid unit.pos unit.goal))
        (self:assign_path unit new_path)
        (set path_info.valid true))))

(fn PathManager.try_local_repair [self unit path_info]
  ;; Find first blocked cell in path
  (var blocked_idx nil)
  (for [i path_info.current_step (length path_info.path)]
    (local cell (. path_info.path i))
    (when (not (. cell :walkable))
      (set blocked_idx i)
      (lua "break")))

  (when (not blocked_idx)
    (return true))  ;; Path is actually fine

  ;; Try to repair around obstacle
  (local before (. path_info.path (- blocked_idx 1)))
  (local after (. path_info.path (+ blocked_idx 1)))

  ;; Can we path around the obstacle locally?
  (when (< (manhattan_distance before after) 10)
    (local detour (A_star_limited grid before after 20))  ;; Max 20 cells
    (when detour
      ;; Splice detour into path
      (splice_path path_info.path blocked_idx detour)
      (return true)))

  false)  ;; Need full replan
```

Key Optimizations:

1. Lazy Revalidation:

```
;; Don't check all paths immediately
;; Mark as "potentially invalid" and check on-demand
(fn Unit.update [self dt]
  (when (not self.path_info.valid)
    ;; Only replan when unit needs the path
    (path_manager:repair_path self.id)))
```

2. Spatial Hashing:

```
;; Only check paths that pass through affected region
(local path_spatial_hash {})  ;; region → [unit_ids]

(fn register_path [unit_id path]
  (each [_ cell (ipairs path)]
    (local region (cell_to_region cell))
    (when (not (. path_spatial_hash region))
      (tset path_spatial_hash region []))
    (table.insert (. path_spatial_hash region) unit_id)))

(fn on_obstacle_added [cell]
  (local region (cell_to_region cell))
  (local affected_units (. path_spatial_hash region))

  ;; Only these units might be affected
  (each [_ unit_id (ipairs affected_units)]
    (invalidate_path unit_id)))
```

3. Path Caching:

```
;; Cache recent repairs
(local repair_cache {})  ;; "{from}→{to}" → {path, obstacles_hash}

(fn cached_repair [from to current_obstacles]
  (local key (.. from.x "," from.y "→" to.x "," to.y))
  (local cached (. repair_cache key))

  (when (and cached
           (= cached.obstacles_hash (hash_obstacles current_obstacles)))
    (return cached.path))  ;; Cache hit!

  ;; Cache miss, compute and store
  (local new_path (A_star grid from to))
  (tset repair_cache key
    {:path new_path
     :obstacles_hash (hash_obstacles current_obstacles)})

  new_path)
```

Performance Analysis:

Without optimization (naive):

```
Obstacle added → 50 units × full A* replan
= 50 × 800 cells × 200 cycles = 8,000,000 cycles = 5.3ms
```

With lazy + spatial hashing:

```
Obstacle added → mark ~5 affected units invalid
Next frame → repair 5 units × local repair (~100 cells)
= 5 × 100 cells × 200 cycles = 100,000 cycles = 0.067ms

95% reduction!
```

Edge Cases:

```
;; Handle complete blockage
(fn try_local_repair [self unit path_info]
  (local detour (A_star_limited grid before after 20))

  (when (not detour)
    ;; Local repair failed, mark for full replan
    (set path_info.needs_full_replan true)
    false))

;; Handle units stuck behind new wall
(fn Unit.update [self dt]
  (when (and (not self.path_info.valid)
             self.path_info.needs_full_replan)
    ;; Might discover goal is unreachable
    (local new_path (A_star grid self.pos self.goal))

    (if new_path
        (self:follow_path new_path)
        ;; Goal unreachable, pick new goal or idle
        (self:handle_unreachable_goal))))
```

This system gracefully handles dynamic obstacles while maintaining performance, only doing expensive replanning when absolutely necessary.