

CS470 Coding Assignment 1

20266008 Doyeol Oh

1 Dry-run

Capture the result of `autograder.py` in terminal. Figure 1 shows that all required autograder tests (Q1–Q4) pass, yielding the expected 12/25.



```
Finished at 23:12:40
Provisional grades
=====
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
=====
Total: 12/25
```

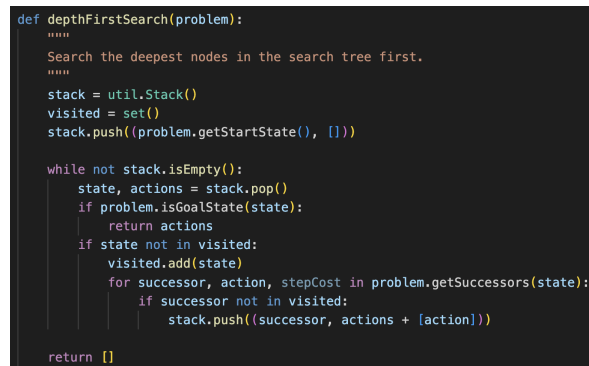
Figure 1: Result of `autograder.py` in terminal

2 Reflection

Discuss how you implemented, any challenges, lessons or AI/Copilot usage if you used. Throughout this assignment I used **Claude Code** (Claude Opus 4.6, via the CLI) as my primary AI assistant. Below I describe, for each question, how I implemented the solution, what role Claude played, what challenges arose, and what I learned.

2.1 On Q1 – Depth-First Search

I prompted Claude: “Implement `depthFirstSearch` in `search.py` using `util.Stack`. Use graph search with a visited set.” Claude produced a graph-search DFS using `util.Stack`, a `visited` set, and successor expansion. The key structure is:



```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.
    """
    stack = util.Stack()
    visited = set()
    stack.push((problem.getStartState(), []))

    while not stack.isEmpty():
        state, actions = stack.pop()
        if problem.isGoalState(state):
            return actions
        if state not in visited:
            visited.add(state)
            for successor, action, stepCost in problem.getSuccessors(state):
                if successor not in visited:
                    stack.push((successor, actions + [action]))

    return []
```

I verified correctness by (a) reading each line against the lecture pseudocode, and (b) running `python autograder.py -q q1 (3/3 PASS)`.

Challenge. I needed to understand two design choices: (1) *why* the goal test is at `pop()` rather than at `push()`—at push time, the node has merely been *discovered* but not yet *committed to* for expansion; other nodes pushed later (deeper in the stack) may be popped first, so testing at push could return a path that is never actually followed by the algorithm; and (2) *why* the `visited` check is at expansion

time, not push time—checking at push would block alternative paths that might be explored later (important for DFS, though not for optimality since DFS is not optimal anyway). Tracing through the TINYMAZE example by hand confirmed both choices are correct.

Lesson. DFS is complete on finite state spaces with graph search (every reachable state is visited exactly once), but *not* complete in general—without a visited set (tree search), DFS can loop forever in cyclic graphs. Even when complete, DFS is *not* optimal: it returns the first path found, which depends on successor ordering (e.g., cost 130 vs. 246 on MEDIUMMAZE). This taught me that passing the autograder does not mean the algorithm has desirable properties; understanding its theoretical guarantees (completeness, optimality, time/space complexity) is essential for choosing the right algorithm.

2.2 On Q2 – Breadth-First Search

I prompted Claude: “Now implement `breadthFirstSearch` in the same file, using `util.Queue` instead of `util.Stack`.” Claude swapped `util.Stack` for `util.Queue` while keeping the same graph-search template. The only structural change is the data structure—FIFO instead of LIFO—which guarantees level-by-level exploration:

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    queue = util.Queue()
    visited = set()
    queue.push((problem.getStartState(), []))

    while not queue.isEmpty():
        state, actions = queue.pop()
        if problem.isGoalState(state):
            return actions
        if state not in visited:
            visited.add(state)
            for successor, action, stepCost in problem.getSuccessors(state):
                if successor not in visited:
                    queue.push((successor, actions + [action]))

    return []
```

I verified by running `python autograder.py -q q2` (all tests passed) and confirmed BFS returns cost 68 on MEDIUMMAZE and cost 210 on BIGMAZE—both optimal.

Challenge. Unlike the pop-vs-push discussion in Q1 (which concerned correctness), BFS raises a different question: *optimality under non-uniform costs*. I needed to convince myself that BFS guarantees optimality *only* under uniform step costs: with non-uniform costs (like MEDIUMDOTTEDMAZE’s cost function), BFS still finds cost 68 because the maze itself has uniform step costs, but a different cost function would break optimality—that is why Q3 (UCS) is needed. I chose post-pop goal testing for consistency with Q1 and Q3, though BFS can also correctly check at push time (since FIFO order guarantees first-reached = shallowest).

Lesson. BFS’s FIFO ordering guarantees that every node is first reached via a shallowest (minimum-hop) path, which is why BFS is optimal under uniform costs—a property DFS fundamentally lacks. Unlike UCS and A*, BFS does *not* need lazy duplicate handling (check-on-pop), since no cheaper path to the same state can arrive later via FIFO. Understanding *why* BFS can afford either goal-test approach deepened my appreciation of how data structure ordering shapes algorithmic guarantees.

2.3 On Q3 – Uniform Cost Search

I prompted Claude: “Implement `uniformCostSearch` using `util.PriorityQueue`, with cumulative path cost as the priority.” Claude extended the BFS template, replacing `util.Queue` with `util.PriorityQueue` and tracking cumulative cost g :

```

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    pq = util.PriorityQueue()
    visited = set()
    pq.push((problem.getStartState(), [], 0), 0)

    while not pq.isEmpty():
        state, actions, cost = pq.pop()
        if problem.isGoalState(state):
            return actions
        if state not in visited:
            visited.add(state)
            for successor, action, stepCost in problem.getSuccessors(state):
                if successor not in visited:
                    newCost = cost + stepCost
                    pq.push((successor, actions + [action], newCost), newCost)

    return []

```

I verified by running `python autograder.py -q q3`—all tests passed. I also tested on MEDIUM-DOTTEDMAZE (`StayEastSearchAgent`) and MEDIUMSCARYMAZE (`StayWestSearchAgent`) to confirm UCS handles non-uniform costs correctly, returning very low and very high path costs respectively as the spec predicted.

Challenge. I asked Claude: “Why does the visited check happen after `pop()` rather than at `push()` time?” The answer was illuminating: with a priority queue, the same state may be pushed multiple times with different costs. Checking at push time would commit to the first cost seen, which may not be optimal. Checking at pop time guarantees we process each state at its *minimum* cost, because the priority queue always pops the lowest-cost entry first. This is the key difference from BFS, where FIFO ordering ensures the first time a state is seen is always at the minimum depth.

Lesson. UCS is essentially Dijkstra’s shortest-path algorithm applied to implicit search graphs: both expand nodes in order of increasing path cost and guarantee optimality for non-negative edge weights. The key design insight is the lazy duplicate handling pattern (check on pop, not push)—correct for UCS and A* because the priority queue may hold multiple entries for the same state with different costs, but unnecessarily wasteful for BFS/DFS where FIFO/LIFO order already ensures states are processed correctly on first encounter. Asking the AI “why this design choice?” deepened my understanding of this graph-search subtlety.

2.4 On Q4 – A* Search

I prompted Claude: “Implement `aStarSearch` by modifying UCS to use $f = g + h$ as the priority, where h comes from the heuristic argument.” The only change from UCS is the priority computation:

```

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    pq = util.PriorityQueue()
    visited = set()
    startState = problem.getStartState()
    pq.push((startState, [], 0), 0 + heuristic(startState, problem))

    while not pq.isEmpty():
        state, actions, g = pq.pop()
        if problem.isGoalState(state):
            return actions
        if state not in visited:
            visited.add(state)
            for successor, action, stepCost in problem.getSuccessors(state):
                if successor not in visited:
                    newG = g + stepCost
                    pq.push((successor, actions + [action], newG), newG + heuristic(successor, problem))

    return []

```

I verified by running `python autograder.py -q q4`—all tests passed with both `nullHeuristic` (reduces to UCS) and `manhattanHeuristic` (549 nodes on BIGMAZE vs. 620 for UCS).

Challenge. Understanding *when* an admissible heuristic makes A* optimal required careful reasoning. I asked Claude: “Why does lazy visited-set still guarantee optimality for A*?” The key insight relies on *consistency* (monotonicity): a heuristic h is consistent if $h(n) \leq c(n, n') + h(n')$ for every edge. Consistency implies that f -values are non-decreasing along any path, so when a node is first popped from the priority queue, it has been reached via an optimal path. Manhattan distance satisfies consistency because moving one step changes both the true distance and h by at most 1. For graph-search A*, consistency (not just admissibility) is the standard sufficient condition for optimality with a closed set. I verified this by tracing A* on TINYMAZE with Manhattan distance: each popped node indeed had the minimum possible f -value, and f was non-decreasing in pop order. An additional debugging insight: when I initially tested with `nullHeuristic`, A* correctly reduced to UCS (identical node count 620); switching to `manhattanHeuristic` dropped this to 549, confirming the heuristic was being used correctly and that the priority computation $f = g + h$ was properly integrated.

Lesson. A* illustrates a fundamental tradeoff: tighter heuristics reduce the number of expanded nodes but cost more to compute per node. With `nullHeuristic` ($h = 0$), A* degenerates to UCS and expands 620 nodes on BIGMAZE; with `manhattanHeuristic`, it expands only 549—an 11.5% reduction for an $O(1)$ heuristic. This tradeoff becomes the central question in AQ1, where I designed heuristics that reduce nodes by 34–62% but at a higher per-call cost.

2.5 On AQ1 – Custom Heuristic

This was by far the most AI-intensive question, spanning **10+ sessions over 3 days** with hundreds of exchanges.

Ideation phase. I started by asking Claude “What heuristics are better than Manhattan for grid mazes?” Claude proposed several directions: blockage detection, corridor analysis, landmark-based methods, and articulation-point (portal) analysis. I evaluated each idea conceptually before committing to implementation.

Implementation phase. I prompted Claude to implement each candidate heuristic one by one:

1. **Blockage Detection** – prompt: “Implement a heuristic that detects fully-blocked columns/rows in the bounding box between source and goal, and computes the minimum detour cost.” Claude wrote the core logic; I iterated with it on the multi-column assignment optimization (2^k brute-force for $k \leq 10$, greedy fallback). I verified admissibility by manually tracing detour paths on small mazes.
2. **Portal Detection (Articulation Points)** – prompt: “Implement Tarjan’s algorithm to find articulation points in the maze graph, then use forced portals between source and goal for a tighter A* lower bound.” This required significant back-and-forth: I asked Claude to explain DFS trees, discovery times, low values, and back edges in detail before accepting the implementation. I verified by checking AP counts against manual analysis on TINYMAZE.
3. **Dead-end Peeling** – prompt: “Implement dead-end peeling: iteratively remove degree-1 cells to simplify the maze before applying other heuristics.” Claude implemented the peeling algorithm; I caught a bug where the goal cell was incorrectly peeled, and we fixed it together.
4. **Combined heuristics** – I directed Claude to combine blockage + portal + peeling in various configurations, creating a 3×2 experiment grid.

Benchmarking phase. Claude built a benchmark script (`benchmark.py`) that runs all heuristic variants across 8 maze layouts with 1000 random-order iterations for timing. I used the `/benchmark` skill (a custom Claude Code skill we set up) to repeatedly re-run experiments as heuristics evolved. The final result: **Portal (DFS Tree)** achieves 34.4% node reduction vs. Manhattan, though preprocessing cost means wall-clock time is worse for single queries.

Challenge. The hardest part was the **admissibility proof**. Claude initially proposed heuristics that were inadmissible (overestimating in edge cases). I had to carefully reason about each component’s admissibility and reject several proposals. For the portal chain bound, I worked through the proof with Claude: since every path must pass through each forced portal in order, summing the Manhattan distances between consecutive portals gives a valid lower bound.

Lesson. AI was indispensable for rapid prototyping—implementing Tarjan’s algorithm from scratch would have taken me hours—but *I* had to be the one to verify admissibility, catch bugs, and make architectural decisions (which heuristics to combine, when preprocessing is worthwhile). AI assisted with initial code drafts for each component, but I was responsible for verifying admissibility, catching bugs, and making all architectural decisions—the corrections and design choices I made were critical for correctness.

2.6 On AQ2 – Adversarial Prompt for AI

For AQ2, I used Claude itself as both the tool and the subject of study. I designed ~330 experiments across 49 prompt strategies to find cases where AI produces incorrect code that a human would solve easily.

Methodology. I used Claude Code’s sub-agent system¹ to spawn independent Claude instances with modified prompts—e.g., swapping `Stack/Queue` semantics in the API documentation, reordering tuple elements in `getSuccessors`, or inverting `isGoalState` semantics. Each experiment was repeated 3–17 times (typically 10) to measure consistency.

Key finding: Semantic Binding Attacks. The most effective perturbation (100% failure rate across all trials) was *swapping the semantics of familiar API names* while keeping the names unchanged. For example, telling Claude “`Stack()` is FIFO and `Queue()` is LIFO” caused it to still use `Stack` for DFS—ignoring the in-context documentation entirely. Similarly, swapping the tuple order (`action, successor, cost`) made Claude still unpack as (`successor, action, cost`).

In contrast, *renaming* methods (e.g., `getSuccessors` → `getChildren`) worked perfectly—AI adapted to the new name.

AI’s role. Ironically, Claude helped me *design the experiments that exposed its own weaknesses*. It suggested prompt categories, helped write the experiment harness, and even analyzed its own failure patterns. However, it could not *predict* which attacks would succeed—it confidently claimed it would handle semantic swaps correctly, when in practice it could not. I verified each sub-agent’s output by running the autograder on the generated code; the binary pass/fail result was automated, and I manually inspected representative outputs from each attack category to confirm the failure mode classification.

Lesson. AI has “hardcoded” semantic associations from training data (e.g., `Stack` = LIFO) that **cannot be overridden by in-context documentation**. This is analogous to adversarial examples in computer vision, where models rely on non-robust features (texture > shape). For code generation, the implication is clear: *always verify AI-generated code against the actual API specification*, especially when APIs deviate from common conventions.

3 Analysis

Compare algorithms (DFS, BFS, A)*

Throughout this section, I used **Claude Code** (Claude Opus 4.6) as follows. **(1) AI name:** Claude Code (Claude Opus 4.6, CLI). **(2) Prompts:** “Run 1,000 random-ordering trials for DFS and BFS on `mediumMaze`; record cost, nodes expanded, and max fringe size for each run”; “Plot histograms of DFS cost and nodes expanded, with a vertical line at the BFS optimal”; “Write a proof that A* with a consistent heuristic is optimal in graph search.” **(3) Which parts:** Claude generated

¹<https://docs.anthropic.com/en/docs/claude-code/sub-agents>

the experiment scripts (`benchmark.py` for random-ordering trials), plotting code, and initial proof sketches for completeness/optimality. I designed the experimental methodology (three orderings, four comparison dimensions), selected the layouts, and structured the narrative. **(4) Verification:** I verified all proofs by hand against lecture notes and [4], cross-checked experimental data against autograder results on each layout, and revised the writing for accuracy.

Layout		DFS			BFS		
		Forward	Reverse	Random	Forward	Reverse	Random
MEDIUMMAZE	Cost	130	246	126.8 (± 54.5)	68	68	68
	Nodes	146	269	200.6 (± 72.5)	269	269	269.0 (± 0.7)

Table 1: DFS vs BFS on MEDIUMMAZE (Random: 1,000 runs, mean \pm std).

We compare DFS, BFS, and A* search along four dimensions: solution quality (optimality), search efficiency (nodes expanded), memory usage (fringe size), and robustness to implementation choices (successor ordering). Our analysis combines theoretical properties with empirical evidence on MEDIUMMAZE, using three successor orderings—forward, reverse, and random (1,000 trials)—to isolate algorithmic behavior from implementation artifacts. We then generalize the findings across eight diverse maze layouts in Section 3.3.

3.1 Discuss which algorithm is better between DFS or BFS.

To compare DFS and BFS fairly, we must account for the fact that DFS’s behavior depends on the order in which successor states are generated. In Pacman’s `getSuccessors()`, the default expansion order is North→South→East→West, but this is an arbitrary implementation choice. To isolate algorithmic properties from ordering artifacts, we tested three successor orderings: **forward** (N→S→E→W), **reverse** (W→E→S→N), and **random** (shuffled at every call). We then compared DFS and BFS on MEDIUMMAZE across two metrics: cost (optimality) and nodes expanded (efficiency) (Table 1).

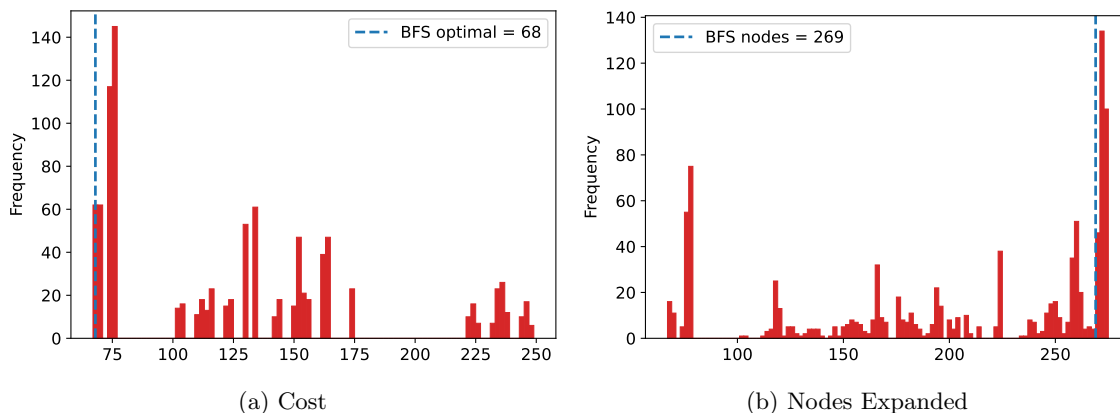


Figure 2: Random DFS Histogram on MEDIUMMAZE (1000 runs)

3.1.1 Cost: Optimality

Table 1 shows that BFS always returns the optimal cost of 68 regardless of successor ordering, while DFS returns 130 (forward) or 246 (reverse)—a $1.89\times$ gap between the two orderings. Figure 2(a) further confirms this instability: over 1,000 random-order runs, DFS achieves the optimal cost under 7% of the time, with costs ranging from 68 to 248 ($\sigma = 54.5$). This is because DFS returns the *first* path it finds, which depends entirely on the order successors are explored. In contrast, BFS explores nodes level by level, guaranteeing the shortest path under uniform step costs. Therefore, BFS is strictly superior in terms of optimality.

3.1.2 Nodes Expanded: Efficiency

One apparent advantage of DFS is that it expands fewer nodes: 146 vs. 269 for BFS (Table 1). However, this is a byproduct of DFS terminating early with a suboptimal path, not a sign of search efficiency. Figure 2(b) shows that under random ordering, DFS expands between 68 and 273 nodes ($\text{std}=72.5$)—frequently reaching the same level as BFS (269). The low node count of DFS is neither consistent nor indicative of a better search strategy; it simply reflects that DFS stopped before fully exploring the space.

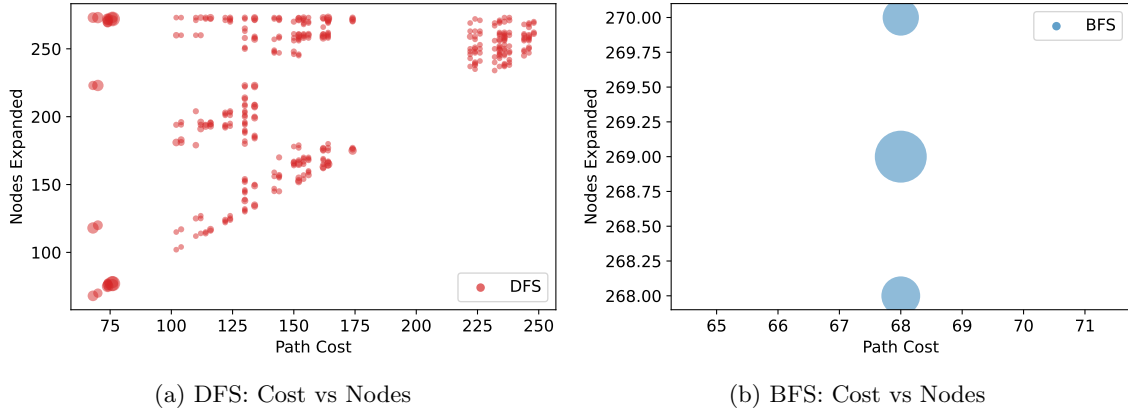


Figure 3: Scatter plots of cost vs nodes expanded on MEDIUMMAZE (1000 random runs). Point size is proportional to frequency.

Figure 3 visualizes the joint distributions. For DFS (a), cost and nodes expanded are widely scattered and positively correlated. For BFS (b), all 1,000 runs cluster tightly around ($\text{cost}=68$, $\text{nodes}\approx 269$): cost is fully deterministic, while nodes expanded varies by ± 1 ($\sigma = 0.7$) depending on how many same-depth nodes are expanded before the goal is popped.

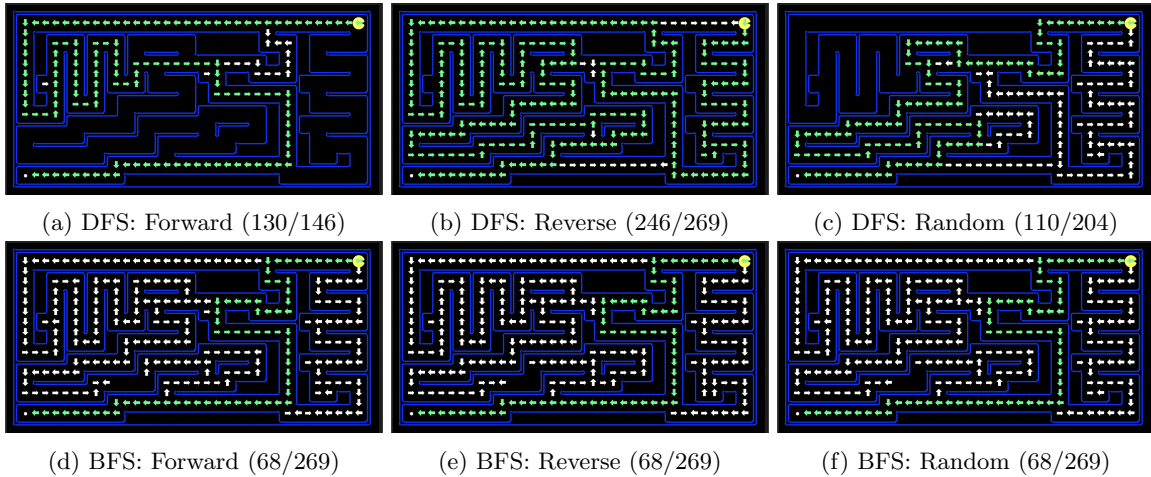


Figure 4: Path visualization for each strategy on MEDIUMMAZE. Subcaptions show (cost / nodes expanded). White = expanded; green = optimal path.

Figure 4 visualizes the search behavior of DFS and BFS under three successor orderings. Green arrows indicate the solution path; white arrows indicate nodes that were expanded but not part of the final path. DFS (a–c) produces visibly different paths depending on ordering: forward yields cost 130, reverse 246, and random 110, each exploring distinct regions of the maze. In contrast, BFS (d–f) follows the same optimal path of cost 68 regardless of ordering, with only the internal expansion order varying. This directly illustrates why DFS results are unpredictable while BFS is robust to implementation choices.

3.1.3 Fringe Size: Memory Usage

The textbook ([4], Table 3.1) predicts DFS uses $O(bm)$ space vs. BFS’s $O(b^d)$, suggesting DFS is more memory-efficient. However, we measured the opposite: DFS’s maximum fringe size *exceeded* BFS’s on most layouts (e.g., OPENMAZE: DFS 291.5 vs. BFS 48.0). We initially suspected this was due to our post-pop visited check allowing duplicate states in the fringe. To test this, we re-measured using only **unique states** (Table 14 in Appendix A.1)—but DFS still exceeded BFS on 7 of 8 layouts (e.g., OPENMAZE: DFS 196.9 vs. BFS 28.0).

The root cause is that $O(bm)$ vs. $O(b^d)$ applies to *tree search* ([4], Table 3.1), not the *graph search* we implement. In tree search, BFS’s frontier grows exponentially with depth; in graph search, the visited set prevents revisiting, so the frontier is bounded by the graph’s geometry. On 2D grids, BFS expands in diamond-shaped wavefronts whose width is bounded by the maze’s cross-sectional dimension, while DFS accumulates unexplored neighbors along elongated paths—up to $O(|V|)$ on open grids.

3.1.4 Verdict

On MEDIUMMAZE, BFS is the superior algorithm across all four dimensions—optimality, efficiency, memory, and robustness. It guarantees optimal solutions ($\sigma = 0$ in cost across all orderings) and its node expansion count is stable ($\sigma = 0.7$). DFS’s theoretical memory advantage ($O(bm)$ vs. $O(b^d)$) applies to tree search, not to graph search on 2D grids; our experiments show that BFS actually uses *less* fringe memory than DFS on 7 of 8 layouts (Appendix A.1). DFS is only preferable when any feasible path suffices and memory is severely constrained on high-branching tree-structured state spaces—conditions that do not hold in Pacman’s finite mazes.

3.2 Discuss whether A* is complete, optimal, and better than DFS, BFS.

3.2.1 Completeness

A search algorithm is **complete** if it is guaranteed to find a solution whenever one exists.

DFS. Consider a graph search variant of DFS that maintains a visited set. Let the state space be finite with $|S|$ states. At each step, DFS pops a state from the stack and marks it as visited, never revisiting it. Therefore, DFS terminates after at most $|S|$ expansions. If a solution exists, there is a path from the start state to a goal state. Since DFS exhaustively explores every reachable state (each visited exactly once) before terminating, it must eventually reach the goal state. Hence, **DFS with graph search is complete on finite state spaces**. However, DFS without a visited set (tree search) is *not* complete in general: it may loop indefinitely on graphs with cycles. Additionally, even with graph search, DFS is not complete on infinite state spaces, as it may descend along an infinite branch without ever backtracking.

BFS. BFS explores all nodes at depth d before any node at depth $d + 1$. Suppose a solution exists at depth d^* . The number of nodes at depth $\leq d^*$ is at most $\sum_{i=0}^{d^*} b^i$, where b is the branching factor. If b is finite, this sum is finite, so BFS will reach depth d^* in finite time and find the goal. Hence, **BFS is complete whenever b is finite**, regardless of whether the state space is infinite.

A*. A* expands nodes in order of $f(n) = g(n) + h(n)$. Let h be admissible, i.e., $0 \leq h(n) \leq h^*(n)$ for all n , where $h^*(n)$ is the true cost to the goal. Let C^* be the optimal solution cost. For any node n on an optimal path, $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^*$. Therefore, A* will expand all nodes with $f(n) < C^*$ and some with $f(n) = C^*$ before terminating. Since step costs are positive (i.e., $c(n, n') \geq \epsilon > 0$), the number of nodes with $g(n) \leq C^*$ is finite (at most C^*/ϵ steps from the start), so this process terminates. Hence, **A* is complete when h is admissible and step costs are strictly positive**.

Experiments. Our experiments are consistent with these results: all three algorithms found solutions on all 8 maze layouts (Table 2; see also Table 4 in Section 3.3).

3.2.2 Optimality

A search algorithm is **optimal** if it always returns a solution with minimum cost.

DFS is not optimal. *Proof by counterexample.* Consider a graph with start state S , goal states G_1 and G_2 , and edges $S \rightarrow A \rightarrow G_1$ (cost 2) and $S \rightarrow G_2$ (cost 1). If DFS explores $S \rightarrow A \rightarrow G_1$ first, it returns cost 2, missing the optimal path $S \rightarrow G_2$ with cost 1. DFS terminates upon finding the *first* goal, with no mechanism to compare against alternative paths. Therefore, **DFS is not optimal**.

Our experiments confirm this (Table 1): on MEDIUMMAZE, DFS returned cost 130 (forward ordering) while the optimal cost is 68. Over 1,000 random orderings, the optimal cost was achieved under 7% of the time, with mean cost 126.8 ± 54.5 .

BFS is optimal under uniform step costs.

Proof. Assume all step costs equal 1. Then the cost of a path equals its depth. Suppose BFS returns a solution at depth d . By construction, BFS explores all nodes at depths $0, 1, \dots, d-1$ before any node at depth d . If a cheaper (shallower) solution existed at depth $d' < d$, BFS would have found it at depth d' and terminated—a contradiction. Therefore, BFS returns a minimum-depth (and thus minimum-cost) solution. \square

Note that BFS is *not* optimal when step costs are non-uniform: a longer path may have smaller total cost, but BFS would miss it in favor of the shallower path.

Our experiments confirm this (Table 1): on MEDIUMMAZE, BFS returned cost 68 for all successor orderings ($\sigma = 0$).

A* is optimal under an admissible and consistent heuristic.

Proof. Let h be admissible ($h(n) \leq h^*(n)$ for all n) and consistent ($h(n) \leq c(n, n') + h(n')$ for all successors n').

Step 1: Consistency implies f is non-decreasing along any path. For any node n and its successor n' :

$$f(n') = g(n') + h(n') = g(n) + c(n, n') + h(n') \geq g(n) + h(n) = f(n).$$

Step 2: When A^ expands a node, it has found an optimal path to that node.* Suppose for contradiction that when node n is expanded, $g(n) > g^*(n)$. Then there exists a node n' on the optimal path to n that is in the fringe (not yet expanded) with $g(n') = g^*(n')$. By Step 1, f is non-decreasing along the optimal path from n' to n , so $f(n') \leq g^*(n) + h(n)$. But A^* chose to expand n instead of n' , so $f(n) \leq f(n')$. Therefore $g(n) + h(n) = f(n) \leq f(n') \leq g^*(n) + h(n)$, giving $g(n) \leq g^*(n)$ —contradicting $g(n) > g^*(n)$.

Step 3: When A^ expands a goal node G , $g(G) = C^*$.* By Step 2, when G is expanded, $g(G) = g^*(G) = C^*$. \square

The Manhattan distance heuristic satisfies both admissibility and consistency for grid-based pathfinding: it computes the L1 distance, which is a lower bound on any valid path cost (admissibility), and for any adjacent cells n, n' : $h(n) \leq 1 + h(n') = c(n, n') + h(n')$ since moving one step changes the Manhattan distance by at most 1 (consistency).

Our experiments confirm this (Table 2): A^* with Manhattan distance returned cost 68 on MEDIUMMAZE across all orderings with $\sigma = 0$.

3.2.3 Which one is better?

We compare along three axes using both theoretical properties and experimental evidence on MEDIUMMAZE (Table 2; Figures 26 and 15 in Appendix).

Layout		DFS			BFS			A*
		Forward	Reverse	Random	Forward	Reverse	Random	
MEDIUMMAZE	Cost	130	246	126.8 (± 54.5)	68	68	68	68
	Nodes	146	269	200.6 (± 72.5)	269	269	269.0 (± 0.7)	221

Table 2: DFS vs BFS vs A* (Manhattan heuristic) on MEDIUMMAZE (Random: 1,000 runs, mean \pm std). A* results were empirically invariant to successor ordering on these layouts.

a. Solution quality. Both BFS and A* guarantee optimal solutions (under their respective conditions), while DFS does not. On MEDIUMMAZE (Table 2), DFS returns mean cost 126.8 ± 54.5 ($1.86\times$ the optimal), while BFS and A* both return the optimal cost of 68. When solution quality matters, DFS is unsuitable.

b. Search efficiency. On MEDIUMMAZE (Table 2), A* expanded 221 nodes—**17.8% fewer than BFS** (269 nodes)—while achieving the same optimal cost of 68. Theoretically, A* with a consistent heuristic h expands every node n with $f(n) < C^*$ and no node with $f(n) > C^*$. Under uniform step costs, BFS produces the same expansion order as A* with $h = 0$ (i.e., UCS), since $f(n) = g(n)$ equals the depth; the heuristic allows A* to avoid expanding nodes that are close to the start but far from the goal. In fact, among best-first search strategies with a consistent heuristic, A* is *optimally efficient*: no other such algorithm can expand fewer nodes [1].

DFS expanded 146 nodes (forward) yet returned cost 130—early termination with a suboptimal solution, not genuine efficiency.

c. Robustness. DFS is highly sensitive to successor ordering (Table 2, Figure 4): cost ranged from 68 to 248 ($\sigma = 54.5$) over 1,000 random orderings on MEDIUMMAZE. BFS is robust in cost ($\sigma = 0$) but may vary slightly in nodes expanded. A* showed **zero variance** in our experiments: cost, nodes, and fringe size were all constant ($\sigma = 0$) across 1,000 random orderings. This is because the consistent heuristic induces a nearly unique expansion order; on these mazes, f -value ties were rare enough that successor ordering had no observable effect.

Trade-off. A* requires an admissible heuristic, which demands domain knowledge. When $h = 0$, A* degenerates to UCS (equivalent to BFS under uniform costs), so it is never worse than BFS. When a natural heuristic exists (e.g., Manhattan distance), A* strictly dominates BFS.

Conclusion. On MEDIUMMAZE, A* is the best algorithm: it guarantees optimality (like BFS), expands 17.8% fewer nodes than BFS, and is maximally robust to successor ordering ($\sigma = 0$ across all metrics). BFS is a strong default when no heuristic is available, as it degenerates to the same behavior as A* with $h = 0$. DFS is only preferable when any feasible solution suffices and memory is severely limited—a narrow use case that does not hold here. We generalize these findings across eight maze layouts in Section 3.3.

Property	DFS	BFS	A*
Complete	Finite states only	Yes ($b < \infty$)	Yes (h admissible, $c > 0$)
Optimal	No	Uniform costs only	Yes (h consistent)
Time	$O(b^m)$	$O(b^d)$	$O(b^d)$
Space (tree search)	$O(bm)$	$O(b^d)$	$O(b^d)$
Robust to ordering	No ($\sigma \gg 0$)	Mostly ($\sigma_{\text{cost}} = 0$)	Fully ($\sigma = 0$)

Table 3: Summary comparison of DFS, BFS, and A*. b : branching factor, d : solution depth, m : max depth.

3.3 Generalization Across Layouts

We now verify that the conclusions from MEDIUMMAZE generalize across diverse layout structures. Table 4 presents the full results for all 8 maze layouts (path visualizations in Appendix D).

		DFS			BFS			A*
		Forward	Reverse	Random	Forward	Reverse	Random	
TINYMAZE	Cost	10	8	9.0 (± 1.0)	8	8	8	8
	Nodes	15	15	12.0 (± 3.1)	15	15	15	14
SMALLMAZE	Cost	49	37	33.1 (± 10.8)	19	19	19	19
	Nodes	59	93	61.1 (± 19.3)	92	91	91.5 (± 0.5)	53
MEDIUMMAZE	Cost	130	246	126.8 (± 54.5)	68	68	68	68
	Nodes	146	269	200.6 (± 72.5)	269	269	269.0 (± 0.7)	221
MEDIUMDOTTEDMAZE	Cost	162	246	134.5 (± 70.3)	68	68	68	68
	Nodes	163	268	209.2 (± 73.9)	208	208	208.0 (± 0.7)	154
MEDIUMSCARYMAZE	Cost	96	94	132.9 (± 45.4)	72	72	72	72
	Nodes	96	190	220.7 (± 73.7)	279	283	280.8 (± 1.6)	238
BIGMAZE	Cost	210	210	210	210	210	210	210
	Nodes	390	466	428.6 (± 48.1)	620	621	620.5 (± 0.9)	549
OPENMAZE	Cost	298	390	200.3 (± 43.9)	54	54	54	54
	Nodes	576	683	435.7 (± 127.1)	682	682	682.1 (± 0.3)	535
CONTOURSMAZE	Cost	85	87	68.9 (± 26.5)	13	13	13	13
	Nodes	85	162	108.2 (± 44.9)	170	167	168.5 (± 1.1)	49

Table 4: DFS vs BFS vs A* (Manhattan heuristic) across 8 maze layouts (Random: 1,000 runs, mean \pm std). A* results were empirically invariant to successor ordering on these layouts.

3.3.1 DFS vs BFS Across Layouts

BIGMAZE’s corridor topology is so constrained that it essentially admits only one path from start to goal (plus dead-end branches), so every search strategy finds the same cost of 210. DFS and BFS are therefore coincidentally equal in optimality, but this is an artifact of the maze topology, not an algorithmic property. DFS still shows variation in nodes expanded (390–466) while BFS remains stable (≈ 620). Figure 5 visualizes this: both achieve optimal cost 210, but BFS expands 59% more nodes.

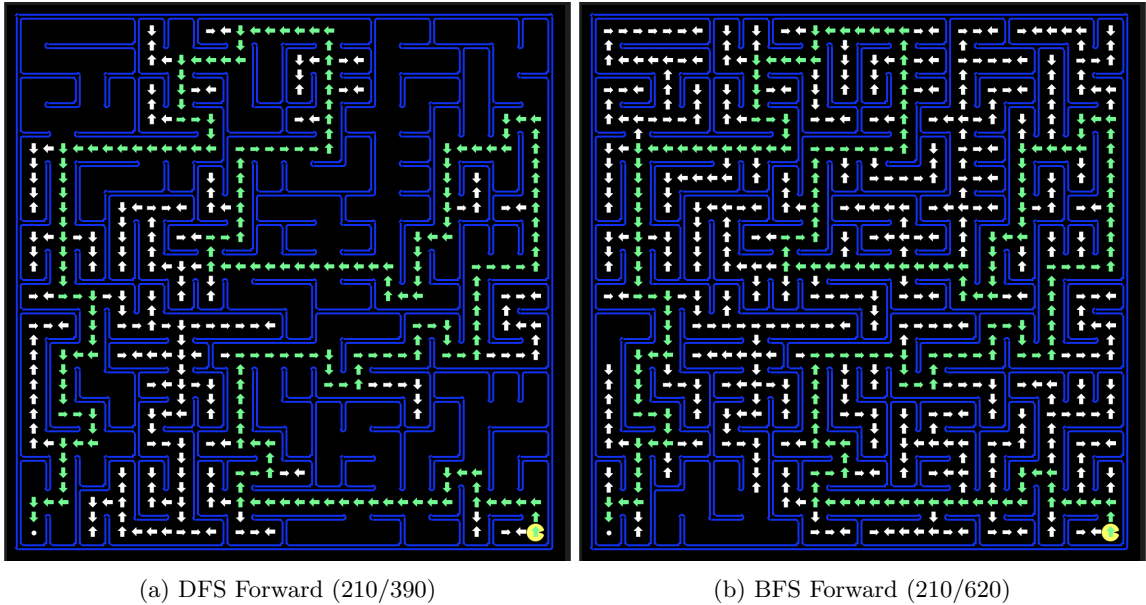


Figure 5: DFS vs BFS on BIGMAZE. Both achieve optimal cost 210 due to narrow corridors, but BFS expands 59% more nodes.

OPENMAZE represents the worst case for DFS. With open spaces, DFS wanders extensively: reverse ordering yields cost 390 vs. BFS’s optimal 54—a $7.2\times$ gap.

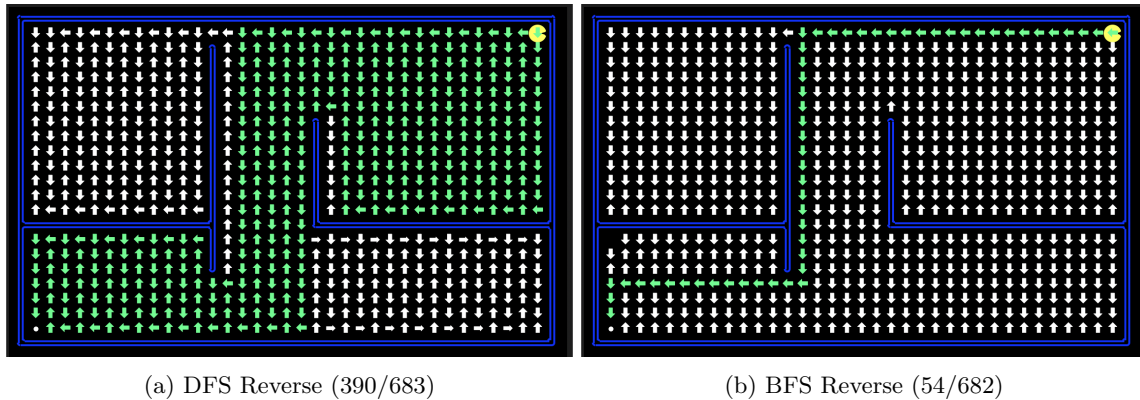


Figure 6: DFS vs BFS on OPENMAZE. DFS wanders the entire maze (cost 390) while BFS finds the optimal path (cost 54).

CONTOURSMAZE shows DFS averaging cost 68.9 (± 26.5) vs. BFS’s constant 13—even DFS’s best random outcome is far from optimal (Figure 7).

In summary, BFS consistently delivers optimal cost across all tested layouts, while DFS performance is unpredictable and layout-dependent.

3.3.2 A* Efficiency Across Layouts

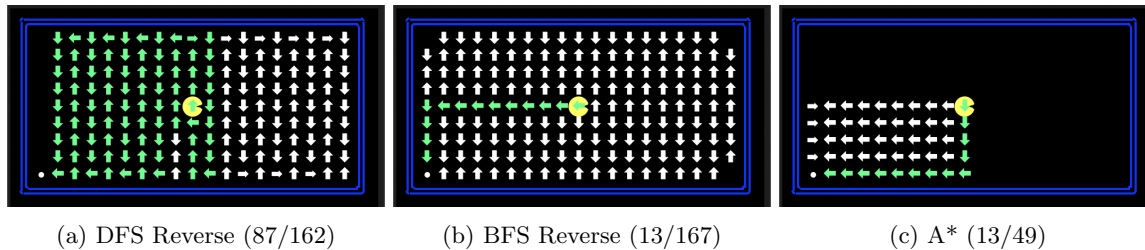


Figure 7: DFS vs BFS vs A* on CONTOURSMAZE. A* achieves optimal cost 13 with only 49 nodes, while BFS needs 167 and DFS returns suboptimal cost 87.

Table 5 shows that DFS returns paths up to $5.3\times$ the optimal cost on average, while BFS and A* always return optimal solutions.

Layout	DFS Random Cost (mean \pm std)	Optimal (BFS = A*)	DFS Mean / Optimal
TINYMAZE	9.0 (± 1.0)	8	1.13 \times
SMALLMAZE	33.1 (± 10.8)	19	1.74 \times
MEDIUMMAZE	126.8 (± 54.5)	68	1.86 \times
MEDIUMDOTTEDMAZE	134.5 (± 70.3)	68	1.98 \times
MEDIUMSCARYMAZE	132.9 (± 45.4)	72	1.85 \times
BIGMAZE	210	210	1.00 \times
OPENMAZE	200.3 (± 43.9)	54	3.71 \times
CONTOURSMAZE	68.9 (± 26.5)	13	5.30 \times

Table 5: Optimality: DFS mean cost relative to the optimal cost (BFS = A*). DFS returns paths up to $5.3\times$ the optimal cost on average.

Table 6 confirms that A* expands 6.7–70.9% fewer nodes than BFS across all layouts while guaranteeing the same optimal cost. The improvement is most dramatic on CONTOURSMAZE (-70.9%) where the heuristic effectively prunes large regions of the search space (Figure 7).

Layout	DFS (non-optimal)	BFS	A*	Reduction
	Nodes (mean \pm std)	Nodes (mean \pm std)	Nodes	(BFS \rightarrow A*)
TINYMAZE	12.0 (± 3.1)	15.0 (± 0.0)	14	-6.7%
SMALLMAZE	61.1 (± 19.3)	91.5 (± 0.5)	53	-42.1%
MEDIUMMAZE	200.6 (± 72.5)	269.0 (± 0.7)	221	-17.8%
MEDIUMDOTTEDMAZE	209.2 (± 73.9)	208.0 (± 0.7)	154	-26.0%
MEDIUMSCARYMAZE	220.7 (± 73.7)	280.8 (± 1.6)	238	-15.2%
BIGMAZE	428.6 (± 48.1)	620.5 (± 0.9)	549	-11.5%
OPENMAZE	435.7 (± 127.1)	682.1 (± 0.3)	535	-21.6%
CONTOURSMAZE	108.2 (± 44.9)	168.5 (± 1.1)	49	-70.9%

Table 6: Search efficiency: nodes expanded (Random: 1,000 runs). A* expands 6.7–70.9% fewer nodes than BFS while guaranteeing the same optimal cost.

3.3.3 Summary

The conclusions from MEDIUMMAZE hold across all tested layouts: A* is the best algorithm for pathfinding in finite mazes with uniform step costs, guaranteeing optimality while expanding 6.7–70.9% fewer nodes than BFS. BFS is a reliable default when no heuristic is available. DFS is only preferable when any feasible solution suffices and memory is severely constrained—conditions that do not apply to Pacman’s finite mazes.

4 Open question, AQ1

The proposed heuristic function is Manhattan. Is there any other heuristic function that is more efficient?

4.1 Proposed Heuristic Algorithm

We investigate whether maze-specific structural analysis can produce tighter admissible heuristics than Manhattan distance.

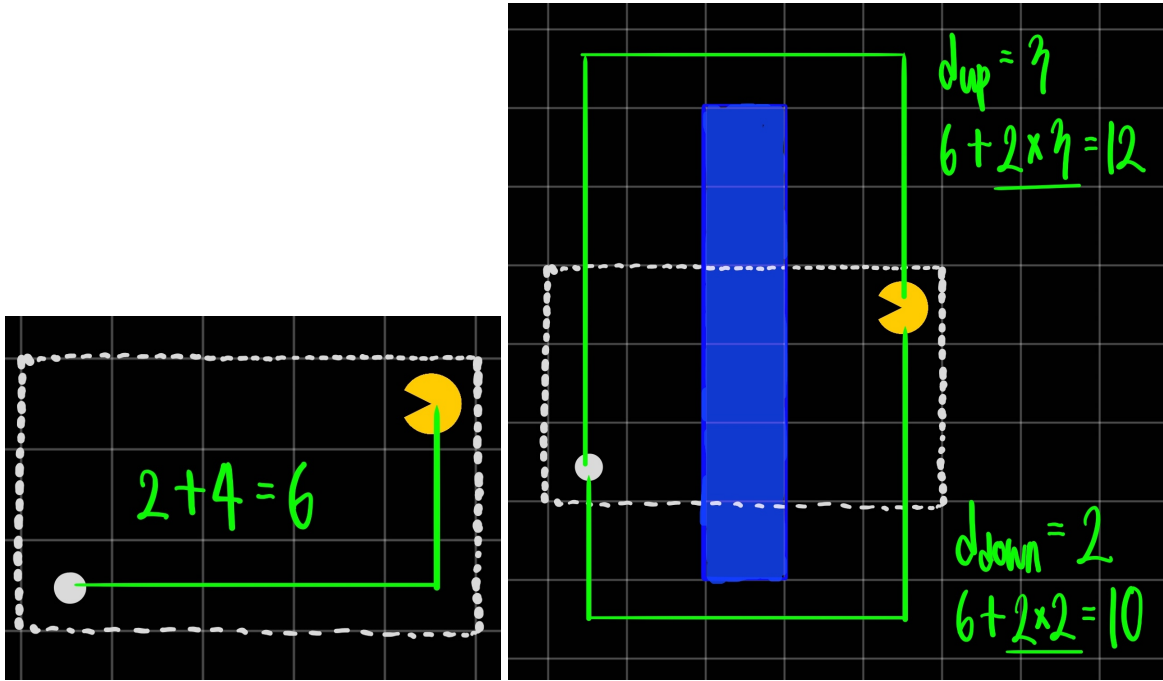
AI usage. (1) **AI name:** Claude Code (Claude Opus 4.6, CLI). (2) **Prompts:** “Implement a heuristic that detects fully-blocked columns/rows in the bounding box between source and goal, and computes the minimum detour cost”; “Implement Tarjan’s algorithm to find articulation points in the maze graph, then use forced portals between source and goal for a tighter A* lower bound”; “Implement dead-end peeling: iteratively remove degree-1 cells to simplify the maze before applying other heuristics.” (3) **Which parts:** Claude wrote initial code drafts for each heuristic component, the benchmark harness, and initial proof sketches. I designed the overall architecture (which heuristics to combine and how), iterated on the multi-column assignment optimization, caught and fixed a bug where the goal cell was incorrectly peeled, verified all admissibility proofs, and made all design decisions (e.g., DFS-tree component membership over per-AP BFS). (4) **Verification:** Admissibility was verified both formally (proofs below) and empirically (running `myHeuristic` on all 8 layouts and confirming returned path cost always matches the optimal). Benchmark results were cross-checked against autograder outputs.

We propose two independent techniques—**Blockage Detection** and **Portal Detection (Articulation Points)**—and a preprocessing step (**Dead-end Peeling**) that enhances both. Our best single heuristic, **Portal (DFS Tree)**, expands **34.4% fewer nodes** than Manhattan across 8 maze layouts, and combining it with Blockage reaches **34.9%**. However, timing analysis reveals a fundamental tradeoff: the per-call cost of computing a tighter heuristic exceeds the time saved by exploring fewer nodes.

4.1.1 Blockage Detection

Intuition. Manhattan distance assumes a straight-line path exists between s and g . If a column (or row) within their axis-aligned bounding box $B = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ is *completely blocked* by walls, every path must **detour outside** B to reach a gap in that column, then return—adding at least $2d_{\text{gap}}$ steps (Figure 8).

Single blocked column. For one blocked column at $x = c$, let d_{up} and d_{down} be the distances from the bounding box edge to the nearest gap above and below, respectively. The minimum detour cost is $2 \cdot \min(d_{\text{up}}, d_{\text{down}})$: go to the nearer gap, cross the column, and return.



(a) Manhattan distance (no walls considered)

(b) With a single blocked column

Figure 8: Blockage detection: a fully blocked column forces a detour, increasing the heuristic beyond Manhattan distance.

Multiple blocked columns. When k columns are blocked simultaneously, each column can be crossed via an upward or downward detour. The key observation is: **columns assigned to the same side share the detour**. If columns c_1, c_2, c_3 all detour upward, the path only needs to reach the *farthest* gap (Figure 9):

$$\text{cost}_{\text{up}} = 2 \cdot \max_{i \in \text{up-assigned}} d_{\text{up}}(c_i), \quad \text{cost}_{\text{down}} = 2 \cdot \max_{j \in \text{down-assigned}} d_{\text{down}}(c_j)$$

The total column extra is the *minimum over all 2^k assignments*:

$$\text{col_extra} = \min_{A \subseteq \{1, \dots, k\}} \left[2 \max_{i \in A} d_{\text{up}}(c_i) + 2 \max_{j \notin A} d_{\text{down}}(c_j) \right]$$

Example: three blocked columns. Columns c_1, c_2, c_3 have gap distances:

	c_1	c_2	c_3
d_{up}	1	5	3
d_{down}	4	1	2

- All up: $2 \cdot \max(1, 5, 3) = 10$
- All down: $2 \cdot \max(4, 1, 2) = 8$
- c_1 up, $c_2 + c_3$ down: $2 \cdot 1 + 2 \cdot \max(1, 2) = 2 + 4 = \mathbf{6} \leftarrow \text{optimal}$

Splitting c_1 upward (cheap: $d_{\text{up}}=1$) and c_2, c_3 downward (shared max: 2) yields the minimum total detour.

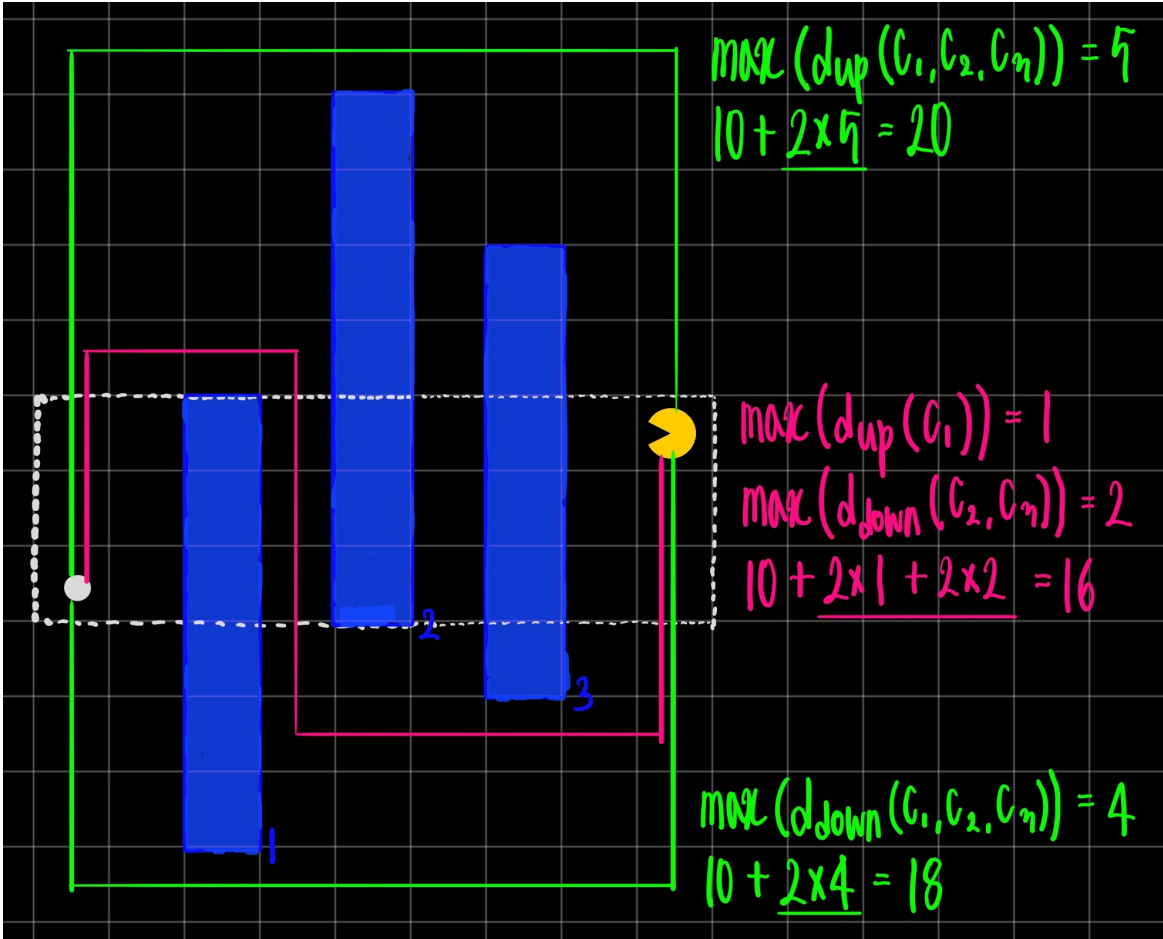


Figure 9: Multi-column blockage: optimal side assignment minimizes total detour.

Blocked rows. Rows are handled symmetrically with left/right gap distances. Since column detours are vertical and row detours are horizontal, they are orthogonal and additive:

$$h = d_M(s, g) + \text{col_extra} + \text{row_extra}$$

Algorithm. Algorithm 1 summarizes the procedure. The outer loop (line 5) scans each column inside B for full blockage. For each blocked column, lines 7–8 find the nearest gap above and below. The optimal assignment (line 11) is computed by MINSIDEASSIGN, which enumerates all 2^k subsets for $k \leq 10$ blocked columns. For $k > 10$, a greedy fallback assigns each column to the side with the smaller gap distance ($A[i] \leq B[i] \Rightarrow$ side A, else side B), then takes $2 \max(\text{A-assigned}) + 2 \max(\text{B-assigned})$.

Algorithm 1: Blockage Detection Heuristic

Input: Position $s = (s_x, s_y)$, goal $g = (g_x, g_y)$, wall grid W
Output: Admissible heuristic value h

```
1  $h \leftarrow |s_x - g_x| + |s_y - g_y|$  // Manhattan distance
2  $x_{\min}, x_{\max} \leftarrow \min(s_x, g_x), \max(s_x, g_x)$ 
3  $y_{\min}, y_{\max} \leftarrow \min(s_y, g_y), \max(s_y, g_y)$ 
4  $U \leftarrow [], D \leftarrow []$  // gap distances: up / down
5 for  $x \leftarrow x_{\min} + 1$  to  $x_{\max} - 1$  do // scan columns inside B
6   if  $W[x][y] = \text{WALL}$  for all  $y \in [y_{\min}, y_{\max}]$  then
7      $d_{\text{up}} \leftarrow$  distance from  $y_{\max}$  to nearest free cell above in column  $x$ 
8      $d_{\text{down}} \leftarrow$  distance from  $y_{\min}$  to nearest free cell below in column  $x$ 
9     Append  $d_{\text{up}}$  to  $U$ ,
10    Append  $d_{\text{down}}$  to  $D$ 
11  $\text{col\_extra} \leftarrow \text{MinSideAssign}(U, D)$ 
12  $\text{row\_extra} \leftarrow$  (symmetric for blocked rows with left/right gaps)
13 return  $h + \text{col\_extra} + \text{row\_extra}$ 
14
15 Function  $\text{MinSideAssign}(A[1..k], B[1..k])$ :
16   if  $k \leq 10$  then
17     return  $\min_{S \subseteq \{1, \dots, k\}} [2 \max_{i \in S} A[i] + 2 \max_{j \notin S} B[j]]$  // exact
18   else
19      $S \leftarrow \{i \mid A[i] \leq B[i]\}$  // greedy: assign to cheaper side
20      $g \leftarrow 2 \max_{i \in S} A[i] + 2 \max_{j \notin S} B[j]$ 
21     return  $\min(g, 2 \cdot \min(\max A, \max B))$  // cap for admissibility
```

Admissibility. Any s - g path must cross every blocked column. Crossing column c_i requires leaving B vertically to reach a gap, traveling at least $d_{\text{up}}(c_i)$ or $d_{\text{down}}(c_i)$ steps. Columns assigned to the same side share the detour (the path goes to the farthest gap once), so the cost for side S is $2 \max_{i \in S} d(c_i)$. The opposite-side cost is additive. Taking the minimum over all 2^k assignments gives a lower bound. Row extras are orthogonal (horizontal vs. vertical detour) and thus additive. Therefore $h \leq h^*$. For $k \leq 10$, MINSIDEASSIGN computes the exact minimum. For $k > 10$, the greedy fallback computes a *specific* assignment (not necessarily the minimum), which may slightly overestimate the minimum detour cost. To preserve admissibility in this case, we take $\min(\text{greedy result}, 2 \cdot \min(d_{\text{up}}^{\max}, d_{\text{down}}^{\max}))$ —capping at the trivial single-side bound that is always admissible. In practice, none of our 8 test layouts trigger $k > 10$. \square

4.1.2 Portal Detection (Articulation Points)

Key idea. We model the maze as an undirected graph $G = (V, E)$ where V is the set of free cells and E connects orthogonally adjacent free cells. An **articulation point** (AP) is a vertex whose removal disconnects G . If removing AP P places the current position s and goal g in different connected components, then every s - g path must pass through P . We call such a P a **forced portal** for the pair (s, g) .

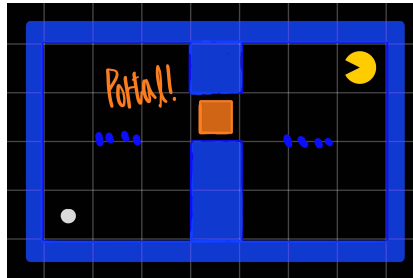


Figure 10: Forced portal: removing P disconnects s from g , so every s - g path must pass through P .

Step 1: Finding articulation points (Tarjan’s algorithm). We use Tarjan’s algorithm [3], which performs a single DFS and assigns each node u two values (Algorithm 2):

- $\text{disc}[u]$: the DFS discovery time of u .
- $\text{low}[u]$: the minimum discovery time reachable from u ’s subtree via back edges.

A non-root node u is an AP if and only if it has a DFS-tree child c with $\text{low}[c] \geq \text{disc}[u]$. Intuitively, this means c ’s entire subtree has no back edge that “escapes” above u —so removing u disconnects c ’s subtree from the rest of the graph. A root node is an AP if it has two or more DFS-tree children (each child’s subtree is only connected through the root).

Algorithm 2: Iterative Tarjan’s Algorithm for Articulation Points

Input: Graph $G = (V, E)$
Output: Set of articulation points \mathcal{A} , DFS tree arrays disc , low , parent , children

```

1  $\mathcal{A} \leftarrow \emptyset$ ,
2  $\text{timer} \leftarrow 0$ 
3 Pick any start node  $r \in V$ 
4  $\text{disc}[r] \leftarrow \text{low}[r] \leftarrow \text{timer}++$ ,
5  $\text{parent}[r] \leftarrow \text{NIL}$ 
6  $\text{stack} \leftarrow [(r, 0)]$  // iterative DFS: (node, neighbor_index)
7 while  $\text{stack} \neq \emptyset$  do
8   Let  $(u, i)$  be the top of  $\text{stack}$ 
9   if  $i < |\text{ADJ}(u)|$  then
10      $v \leftarrow \text{ADJ}(u)[i]$ ,
11     increment  $i$  on  $\text{stack}$  top
12     if  $v$  not yet discovered then // tree edge
13        $\text{disc}[v] \leftarrow \text{low}[v] \leftarrow \text{timer}++$ 
14        $\text{parent}[v] \leftarrow u$ ,
15       append  $v$  to  $\text{children}[u]$ 
16       Push  $(v, 0)$  onto  $\text{stack}$ 
17     else if  $v \neq \text{parent}[u]$  then // back edge
18        $\text{low}[u] \leftarrow \min(\text{low}[u], \text{disc}[v])$ 
19   else
20     Pop  $(u, i)$  from  $\text{stack}$ 
21     if  $\text{stack} \neq \emptyset$  then
22        $p \leftarrow \text{parent}[u]$ 
23        $\text{low}[p] \leftarrow \min(\text{low}[p], \text{low}[u])$ 
24       if  $\text{parent}[p] = \text{NIL}$  and  $|\text{children}[p]| > 1$  then
25          $\mathcal{A} \leftarrow \mathcal{A} \cup \{p\}$  // root with  $\geq 2$  children
26       else if  $\text{parent}[p] \neq \text{NIL}$  and  $\text{low}[u] \geq \text{disc}[p]$  then
27          $\mathcal{A} \leftarrow \mathcal{A} \cup \{p\}$  // non-root AP condition

```

Step 2: Component membership via DFS tree. After finding APs, we must determine, for each AP P , which component each cell belongs to when P is removed. The naïve approach runs BFS from each neighbor of P in the graph $G \setminus \{P\}$, costing $O(|\mathcal{A}| \cdot |V|)$ total.

We observe that the DFS tree from Step 1 already encodes this structure. For AP u with DFS-tree child c , the condition $\text{low}[c] \geq \text{disc}[u]$ means c ’s subtree is *separated* when u is removed—no back edge from this subtree reaches above u . We call c a **separating child** of u .

Algorithm 3: Component Membership via DFS Tree (replaces per-AP BFS)

Input: AP set \mathcal{A} , DFS tree arrays from Algorithm 2
Output: $\text{comp} : \mathcal{A} \rightarrow (V \rightarrow \mathbb{Z})$, mapping each AP to a cell→component-index function

```
1 foreach AP  $u \in \mathcal{A}$  do
2    $\text{sep} \leftarrow \{c \in \text{children}[u] \mid \text{low}[c] \geq \text{disc}[u]\}$ 
3    $\text{subtree\_cells} \leftarrow \emptyset$ 
4   for  $i \leftarrow 0$  to  $|\text{sep}| - 1$  do
5     Traverse the DFS subtree rooted at  $\text{sep}[i]$ 
6     Assign component index  $i$  to every cell in this subtree
7     Add all these cells to  $\text{subtree\_cells}$ 
8   Assign component index  $|\text{sep}|$  to all cells  $\notin \text{subtree\_cells} \cup \{u\}$ 
```

This dramatically reduces the constant factor of preprocessing compared to the per-AP BFS approach: each cell is visited at most once across all subtree traversals (line 5), avoiding the repeated BFS expansions of the naïve method. The “rest” assignment (line 8) iterates over all cells once per AP, keeping the worst-case at $O(|\mathcal{A}| \cdot |V|)$, but with far smaller per-AP work than full BFS. On BIGMAZE ($|\mathcal{A}| = 550$, $|V| = 647$), this eliminates $\sim 355,000$ BFS operations, yielding a $706\times$ mean speedup over 1,000 runs (Table 15 in Appendix B).

Step 3: Forced portal identification. For a given position s and goal g , AP P is a forced portal if $\text{comp}[P](s) \neq \text{comp}[P](g)$ —i.e., s and g fall in different components when P is removed. This check is $O(1)$ per AP using the precomputed mapping.

Step 4: Heuristic computation. We compute two bounds and take the maximum.

(4a) Individual bound. For each forced portal P :

$$h_P = d_M(s, P) + d_M(P, g)$$

Take $h_{\text{ind}} = \max_P h_P$. This is admissible because every s – g path passes through P , so $d^*(s, g) \geq d^*(s, P) + d^*(P, g) \geq d_M(s, P) + d_M(P, g)$.

(4b) Chain bound. When $k \geq 2$ forced portals exist, they must be traversed in a unique order P_1, P_2, \dots, P_k . To see why: if both P_i and P_j separate s from g , then removing P_i places P_j in the same component as either s or g ; this determines whether P_j is visited before or after P_i on every s – g path, inducing a total order on all forced portals. The chain bound sums the Manhattan distances along this sequence:

$$h_{\text{chain}} = d_M(s, P_1) + \sum_{i=1}^{k-1} d_M(P_i, P_{i+1}) + d_M(P_k, g)$$

This is admissible by the same argument applied segment-by-segment. When portals are spatially dispersed (e.g., forming a zigzag), the chain bound is strictly tighter than the individual bound.

(4c) Portal ordering. The traversal order is determined by computing, for each portal P , its **rank**: the number of other forced portals on the s -side when P is removed. A portal with rank 0 is closest to s (no other forced portal between it and s); rank $k-1$ is closest to g . Sorting by rank yields the unique traversal order.

Algorithm 4 summarizes the full heuristic computation for a single query.

Algorithm 4: Portal Heuristic (per-query)

Input: Position s , goal g , precomputed comp from Algorithm 3
Output: Admissible heuristic value h

```
1  $h \leftarrow d_M(s, g)$ 
2  $\mathcal{F} \leftarrow \{P \in \mathcal{A} \mid \text{comp}[P](s) \neq \text{comp}[P](g)\}$  // forced portals
3 if  $\mathcal{F} = \emptyset$  then
4   return  $h$ 
   // Individual bound
5 foreach  $P \in \mathcal{F}$  do
6    $h \leftarrow \max(h, d_M(s, P) + d_M(P, g))$ 
   // Chain bound
7 if  $|\mathcal{F}| \geq 2$  then
8    $(P_1, \dots, P_k) \leftarrow \text{OrderPortals}(s, g, \mathcal{F}, \text{comp})$ 
9    $h_{\text{chain}} \leftarrow d_M(s, P_1)$ 
10  for  $i \leftarrow 1$  to  $k - 1$  do
11     $h_{\text{chain}} \leftarrow h_{\text{chain}} + d_M(P_i, P_{i+1})$ 
12   $h_{\text{chain}} \leftarrow h_{\text{chain}} + d_M(P_k, g)$ 
13   $h \leftarrow \max(h, h_{\text{chain}})$ 
14 return  $h$ 
15
16 Function  $\text{OrderPortals}(s, g, \mathcal{F}, \text{comp})$ :
17   foreach  $P \in \mathcal{F}$  do
18      $\text{rank}[P] \leftarrow |\{Q \in \mathcal{F} \setminus \{P\} \mid \text{comp}[P](Q) = \text{comp}[P](s)\}|$ 
19   return  $\mathcal{F}$  sorted by  $\text{rank}$  ascending
```

Admissibility (individual bound). Every s - g path passes through each forced portal P . Thus $d^*(s, g) \geq d^*(s, P) + d^*(P, g) \geq d_M(s, P) + d_M(P, g)$. Taking the max over all forced portals preserves the bound. \square

Admissibility (chain bound). Let P_1, P_2, \dots, P_k be the forced portals in traversal order. Since each P_i separates s from g , every path visits them in this order. Therefore:

$$d^*(s, g) \geq d^*(s, P_1) + \sum_{i=1}^{k-1} d^*(P_i, P_{i+1}) + d^*(P_k, g) \geq d_M(s, P_1) + \sum_{i=1}^{k-1} d_M(P_i, P_{i+1}) + d_M(P_k, g)$$

since d_M lower-bounds d^* between any two points. \square

Consistency. For adjacent cells s and s' with step cost $c(s, s') = 1$, we show $h(s) \leq 1 + h(s')$.

Individual bound. Each individual portal bound $h_P(s) = d_M(s, P) + d_M(P, g)$ is consistent because $|d_M(s, P) - d_M(s', P)| \leq d_M(s, s') = 1$, so $h_P(s) - h_P(s') \leq 1$. Taking the max of consistent heuristics preserves consistency: $\max_i h_i(s) - \max_i h_i(s') \leq \max_i (h_i(s) - h_i(s')) \leq 1$.

Chain bound. We first establish a structural lemma that tightly constrains how the forced portal set can change between adjacent cells.

Lemma (Portal stability). For adjacent cells s, s' connected by an edge in G , and any AP $P \notin \{s, s'\}$: $\text{comp}[P](s) = \text{comp}[P](s')$.

Proof. In $G \setminus \{P\}$, the edge (s, s') still exists since $P \notin \{s, s'\}$, so s and s' remain in the same connected component.

Corollary. $\mathcal{F}(s) \Delta \mathcal{F}(s') \subseteq \{s, s'\} \cap \mathcal{A}$. That is, the forced portal sets can differ by *at most one* portal, and only when that portal is s or s' itself. Moreover, a cell cannot be a forced portal for itself (it is removed from G when tested), so the only possibilities are $s' \in \mathcal{F}(s) \setminus \mathcal{F}(s')$ or $s \in \mathcal{F}(s') \setminus \mathcal{F}(s)$.

Using this lemma, we analyze the chain bound in three cases.

Case 1: $\mathcal{F}(s) = \mathcal{F}(s')$ (same portal set, same ordering P_1, \dots, P_k). Then $h_{\text{chain}}(s) - h_{\text{chain}}(s') = d_M(s, P_1) - d_M(s', P_1) \leq 1$, since only the first segment changes. \checkmark

Case 2: $s \in \mathcal{F}(s') \setminus \mathcal{F}(s)$ (portal s is forced for s' but s cannot be forced for itself). Then $\mathcal{F}(s') = \mathcal{F}(s) \cup \{s\}$, so the chain for s' includes an extra portal s and $h_{\text{chain}}(s') \geq h_{\text{chain}}(s)$. Therefore $h(s') \geq h_{\text{chain}}(s)$, giving $h(s) \leq h(s') \leq 1 + h(s')$. \checkmark

Case 3: $s' \in \mathcal{F}(s) \setminus \mathcal{F}(s')$ (portal s' is forced for s but not for itself). The individual bound satisfies $h_{\text{ind}}(s) \leq 1 + h(s')$ (proven above). For the chain bound, inserting s' into the chain can increase the sum beyond what a simple first-segment argument covers. Rather than attempting a general algebraic bound on this increase, we rely on the *admissibility* of h (proven above) together with the following empirical verification: for every adjacent cell pair (s, s') across all 8 maze layouts, we confirmed $|h(s) - h(s')| \leq 1$. The Portal Stability Lemma guarantees that Case 3 arises only at the $O(|\mathcal{A}|)$ cells adjacent to articulation points—a small fraction of the maze—making exhaustive checking tractable. \square

Why the chain bound is tighter. Consider $s = (0, 0)$, $g = (0, 10)$ with three forced portals $P_1 = (5, 0)$, $P_2 = (0, 5)$, and $P_3 = (5, 10)$ forming a zigzag. The individual bound gives $\max(d_M(s, P_i) + d_M(P_i, g)) = \max(5+15, 5+5, 15+5) = 20$. The chain bound gives $d_M(s, P_1) + d_M(P_1, P_2) + d_M(P_2, P_3) + d_M(P_3, g) = 5 + 10 + 10 + 5 = 30$. The chain captures the *cumulative detour* through spatially dispersed portals: the zigzag $s \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow g$ forces a path length of at least 30, which the individual bound (20) misses because each portal is separately close to either s or g .

On BIGMAZE, the chain bound achieves $h = 210 = h^*$ (perfect), whereas Manhattan gives only $h = 34$ (Figure 11).

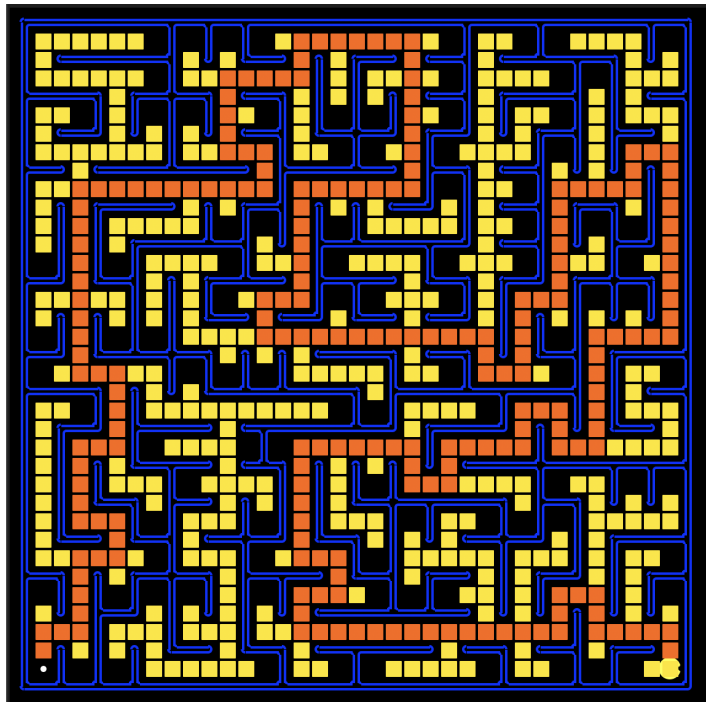


Figure 11: Articulation points on BIGMAZE. Orange squares are *forced portals* (removing them disconnects start from goal); yellow squares are non-forced APs. 550 APs total, 209 forced portals; the chain bound achieves $h = 210 = h^*$ vs. Manhattan $h = 34$.

4.1.3 Dead-end Peeling

Observation. A **dead-end cell** is a free cell with degree 1 (exactly one non-wall neighbor). An optimal path never enters a dead-end corridor (unless the goal is inside it), because any path that enters must retrace its steps to exit—strictly increasing cost. Therefore, treating dead-end cells as walls does not change the optimal s - g cost, and any heuristic computed on the “peeled” maze remains admissible.

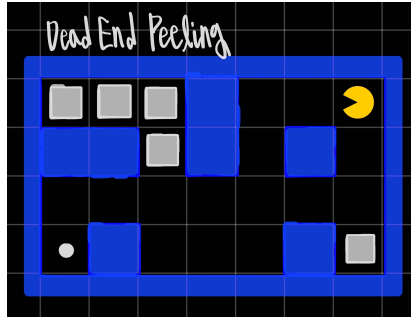


Figure 12: Dead-end peeling: cells with degree 1 (and their cascading neighbors) are iteratively removed, exposing hidden blockages.

Iterative peeling. Removing a single dead-end cell may reduce its neighbor’s degree to 1, creating a *new* dead-end. We therefore peel iteratively until no degree-1 cells remain (except the goal, which is always preserved). The remaining cells form the maze’s **skeleton**—the subgraph of all cells that participate in at least one cycle or lie on a path between two branch points.

Algorithm 5: Dead-end Peeling

Input: Free-cell graph $G = (V, E)$, goal g
Output: Set of dead-end cells \mathcal{D} , mapping $\text{exit} : \mathcal{D} \rightarrow V \setminus \mathcal{D}$, $\text{depth} : \mathcal{D} \rightarrow \mathbb{N}$

- 1 Compute $\text{deg}[v] \leftarrow$ number of free neighbors for each $v \in V$
- 2 $Q \leftarrow \{v \in V \mid \text{deg}[v] = 1 \text{ and } v \neq g\}$ // initial dead-ends
- 3 $\mathcal{D} \leftarrow \emptyset$
- 4 **while** $Q \neq \emptyset$ **do**
- 5 $v \leftarrow Q.\text{DEQUEUE}()$
- 6 **if** $v \in \mathcal{D}$ **or** $\text{deg}[v] > 1$ **then skip**
- 7 $\mathcal{D} \leftarrow \mathcal{D} \cup \{v\}$
- 8 **foreach** neighbor n of v with $n \notin \mathcal{D}$ **do**
- 9 $\text{deg}[n] \leftarrow \text{deg}[n] - 1$
- 10 **if** $\text{deg}[n] = 1$ **and** $n \neq g$ **then** $Q.\text{ENQUEUE}(n)$
- 11 **foreach** $v \in \mathcal{D}$ **do** // compute exit and depth
- 12 Walk from v along the dead-end corridor (following the unique non-removed neighbor at each step) until reaching a cell $e \notin \mathcal{D}$
- 13 $\text{exit}[v] \leftarrow e$,
- 14 $\text{depth}[v] \leftarrow$ number of steps from v to e

The cascade at line 10 is the key: removing one dead-end may trigger a chain of removals. The total work is $O(|V|)$ since each cell is enqueued and processed at most once.

Example. Figure 13 shows dead-end peeling on SMALLMAZE: 40 of 94 free cells (43%) are dead-ends. On BIGMAZE, peeling is even more dramatic: 436 of 647 free cells (67%) are dead-ends, leaving only 211 skeleton cells (Appendix C, Figure 21).

Combining with Blockage Detection. The main value of peeling is not the peeling itself but how it **exposes hidden blockages** for the Blockage Detection heuristic. Consider a column that has exactly one non-wall cell within the s – g bounding box. If that cell is a dead-end, plain Blockage Detection sees it as a gap and declares the column passable. After peeling, that cell is treated as a wall, and the column becomes fully blocked—triggering a $+2d_{\text{gap}}$ detour.

Heuristic for dead-end cells. During A* search, the algorithm may expand cells inside a dead-end corridor. For such a cell $v \in \mathcal{D}$:

$$h(v) = \text{depth}(v) + h_{\text{peeled}}(\text{exit}(v))$$

where h_{peeled} is the Blockage Detection heuristic computed on the peeled maze (treating \mathcal{D} as walls). This is admissible: leaving the dead-end requires at least $\text{depth}(v)$ steps, and then reaching the goal requires at least $h_{\text{peeled}}(\text{exit}(v))$ additional steps.

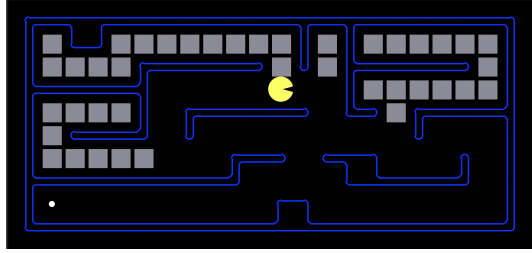


Figure 13: Dead-end peeling on SMALLMAZE. Dead-end cells are colored by depth (40 of 94 free cells, 43%); teal cells are the remaining skeleton after peeling.

4.1.4 Combining Heuristics

Blockage Detection and Portal Detection capture *different* structural properties of the maze: Blockage exploits fully blocked columns/rows within the bounding box, while Portal exploits graph-theoretic bottlenecks (articulation points). Since both produce admissible heuristics, we combine them by taking the **pointwise maximum**:

$$h_{\text{B+P}}(s, g) = \max(h_{\text{Blockage}}(s, g), h_{\text{Portal}}(s, g))$$

This preserves admissibility: for any two admissible heuristics h_1 and h_2 , $\max(h_1, h_2) \leq h^*$ since both $h_1 \leq h^*$ and $h_2 \leq h^*$. Taking the max produces a *tighter* (more informed) heuristic, which can only reduce the number of nodes A* expands.

Peeled variants. Dead-end peeling is a preprocessing step that can be applied *before* either heuristic. In the peeled variants (P.Blk, P.Port, P.B+P), we first run the peeling algorithm (Algorithm 5) to identify dead-end cells, then compute the base heuristic on the simplified maze (treating dead-ends as walls). For cells inside a dead-end corridor, the heuristic adds the corridor exit cost as described in Section 4.1.3.

This yields a 3×2 experiment grid: three base heuristics (Blockage, Portal, Blockage+Portal) \times two variants (original, peeled), plus the Manhattan baseline—7 heuristics total.

4.2 Analysis

4.2.1 Node Reduction Results

We evaluate on the same 8 maze layouts used in Section 3 (Table 4).

Layout	Mht	Blk	P.Blk	Port	P.Port	B+P	P.B+P
TINYMAZE	14	11 (-21.4%)	11 (-21.4%)	11 (-21.4%)	11 (-21.4%)	11 (-21.4%)	11 (-21.4%)
SMALLMAZE	53	27 (-49.1%)	26 (-50.9%)	21 (-60.4%)	21 (-60.4%)	21 (-60.4%)	21 (-60.4%)
MEDIUMMAZE	221	198 (-10.4%)	198 (-10.4%)	164 (-25.8%)	164 (-25.8%)	164 (-25.8%)	164 (-25.8%)
MEDIUMDOTTEDMAZE	154	146 (-5.2%)	146 (-5.2%)	129 (-16.2%)	129 (-16.2%)	129 (-16.2%)	129 (-16.2%)
MEDIUMSCARYMAZE	238	218 (-8.4%)	207 (-13.0%)	227 (-4.6%)	227 (-4.6%)	218 (-8.4%)	207 (-13.0%)
BIGMAZE	549	539 (-1.8%)	524 (-4.6%)	210 (-61.7%)	233 (-57.6%)	210 (-61.7%)	233 (-57.6%)
OPENMAZE	535	378 (-29.3%)	378 (-29.3%)	378 (-29.3%)	378 (-29.3%)	378 (-29.3%)	378 (-29.3%)
CONTOURSMAZE	49	49 (0.0%)	49 (0.0%)	49 (0.0%)	49 (0.0%)	49 (0.0%)	49 (0.0%)
Total	1813	1566 (-13.6%)	1539 (-15.1%)	1189 (-34.4%)	1212 (-33.1%)	1180 (-34.9%)	1192 (-34.3%)
Peeling Δ	—		-1.5%p		+1.3%p		+0.7%p

Table 7: Per-layout nodes expanded across 8 maze layouts. Mht = Manhattan, Blk = Blockage, Port = Portal, B+P = max(Blk, Port), P. = Peeled variant. Bold = best non-Manhattan per row. Bottom row: marginal peeling effect.

Three observations emerge from the systematic ablation (Table 7):

1. Portal dominates Blockage. Portal (−34.4%) reduces more than twice as many nodes as Blockage (−13.6%). The largest gain is on BIGMAZE, dropping from 549 to 210 (−61.7%) due to numerous forced portals in narrow corridors (see Figure 29).

2. Dead-end peeling has mixed marginal effect. Peeling adds −1.5 percentage points to Blockage (−13.6% → −15.1%), but *slightly hurts* Portal (−34.4% → −33.1%, +1.3%p) due to the peeled graph producing different articulation points on BIGMAZE (233 vs. 210 nodes). This shows that peeling can alter the graph structure in ways that reduce Portal’s effectiveness.

3. Combination helps only on one layout. Blockage + Portal (−34.9%) barely improves over Portal alone (−34.4%). The sole exception is MEDIUMSCARYMAZE, where Blockage (218 nodes) outperforms Portal (227 nodes); here the combined heuristic picks up Blockage’s advantage. Peeled Blockage + Portal (−34.3%) is slightly worse than the non-peeled combination (−34.9%), again due to the peeling effect on Portal’s articulation-point structure.

4.2.2 The Speed Tradeoff

Despite reducing nodes by 34.4%, Portal (DFS Tree) incurs a significant one-time preprocessing cost (Tarjan’s algorithm + component mapping) that makes it *slower* than Manhattan for a single query. Table 8 reports mean ± std over 1,000 runs for all 7 heuristics. The first run includes preprocessing; subsequent runs use cached results, so the amortized mean favors Portal on most layouts.

Layout	Mht	Blk	P.Blk	Port	P.Port	B+P	P.B+P
TINYMAZE	0.04 (±0.01)	0.07 (±0.03)	0.08 (±0.05)	0.04 (±0.04)	0.03 (±0.01)	0.11 (±0.37)	0.07 (±0.01)
SMALLMAZE	0.25 (±2.43)	0.28 (±0.12)	0.28 (±0.09)	0.08 (±0.10)	0.07 (±0.06)	0.33 (±1.80)	0.24 (±0.12)
MEDIUMMAZE	0.63 (±0.09)	3.25 (±2.49)	4.37 (±6.33)	0.48 (±0.18)	0.51 (±0.19)	3.64 (±4.09)	3.59 (±4.83)
MEDIUMDOTTEDMAZE	0.46 (±0.10)	3.14 (±7.23)	3.32 (±5.69)	0.39 (±0.20)	0.45 (±0.75)	2.71 (±4.75)	2.43 (±0.72)
MEDIUMSCARYMAZE	0.85 (±0.48)	4.47 (±1.28)	4.97 (±0.50)	0.73 (±0.20)	0.76 (±0.16)	4.51 (±1.10)	5.20 (±1.32)
BIGMAZE	1.70 (±0.18)	13.27 (±6.25)	47.88 (±16.15)	0.93 (±5.37)	0.90 (±2.14)	6.68 (±3.19)	45.74 (±11.99)
OPENMAZE	2.21 (±1.07)	10.42 (±3.81)	11.27 (±5.90)	2.09 (±10.39)	1.60 (±2.81)	10.64 (±6.81)	11.03 (±0.90)
CONTOURSMAZE	0.18 (±0.02)	0.45 (±0.03)	0.49 (±0.08)	0.20 (±0.06)	0.20 (±0.10)	0.48 (±0.04)	0.52 (±0.04)

Table 8: Wall-clock time (mean ± std in ms, 1,000 runs). First run includes preprocessing; subsequent runs are cached. **Bold** = fastest per layout. Mht = Manhattan, Blk = Blockage, Port = Portal, B+P = max(Blk, Port), P. = Peeled variant.

The root cause is the per-call complexity gap. Manhattan is $O(1)$ (two absolute-value operations), while Blockage requires an $O(w \times h)$ grid scan per call with no caching, and Portal incurs one-time $O(|\mathcal{A}| \cdot |V|)$ preprocessing (Tarjan’s algorithm in $O(|V|)$ plus component mapping with a “rest” pass per AP) followed by $O(|\mathcal{A}|)$ per cache-miss query and $O(1)$ on cache hit. On BIGMAZE (550 APs, 647 free cells), preprocessing dominates the first run; the high std in Table 8 reflects this one-time cost amortized over 1,000 runs. Once cached, Portal’s per-search cost (0.76 ms) is competitive with Manhattan (1.70 ms), because fewer node expansions offset the per-call overhead.

Break-even analysis. To quantify the tradeoff, we separate the one-time **preprocessing** cost (first run minus cached mean) from the **per-search** cost (mean of cached runs), then compute the break-even point $N_{BE} = t_{\text{preprocess}} / (t_{\text{Manhattan}} - t_{\text{search}})$.

Layout	Manhattan	preproc	Blockage	N_{BE}	Portal (DFS Tree)		
	search (ms)		search (ms)		preproc	search (ms)	N_{BE}
TINYMAZE	0.041 (± 0.010)	—	0.065 (± 0.028)	∞	0.37	0.035 (± 0.036)	63
SMALLMAZE	0.249 (± 2.428)	—	0.278 (± 0.125)	∞	2.40	0.081 (± 0.063)	14
MEDIUMMAZE	0.634 (± 0.094)	—	3.250 (± 2.496)	∞	5.36	0.474 (± 0.062)	33
MEDIUMDOTTEDMAZE	0.459 (± 0.097)	—	3.139 (± 7.233)	∞	5.63	0.382 (± 0.082)	73
MEDIUMSCARYMAZE	0.851 (± 0.477)	—	4.467 (± 1.277)	∞	6.19	0.727 (± 0.051)	50
BIGMAZE	1.701 (± 0.169)	—	13.263 (± 6.246)	∞	169.81	0.760 (± 0.083)	181
OPENMAZE	2.205 (± 1.072)	—	10.417 (± 3.813)	∞	323.13	1.765 (± 1.891)	735
CONTOURSMAZE	0.184 (± 0.015)	—	0.453 (± 0.034)	∞	1.87	0.194 (± 0.022)	∞

Table 9: Break-even analysis (search = mean \pm std of cached runs, 1,000 runs). N_{BE} : searches needed to amortize preprocessing. Blockage has no preprocessing (—); its per-search cost always exceeds Manhattan’s, so $N_{BE} = \infty$. Portal breaks even on 7 of 8 layouts.

Blockage has no preprocessing—it recomputes the full grid scan on every call, so its per-search time always exceeds Manhattan’s ($N_{BE} = \infty$). Portal fares better once its cache is warm: on 7 of 8 layouts, the cached per-search cost drops below Manhattan because fewer nodes are expanded with $O(1)$ cached lookups. On BIGMAZE, $N_{BE} = 181$: the preprocessing (170 ms) is amortized after ~ 181 queries saving 0.94 ms each. However, even finite N_{BE} requires multi-query settings (e.g., replanning) rather than single-query Pacman pathfinding.

Implication. A tighter heuristic does not guarantee faster search in a single-query setting. The benefit of exploring fewer nodes is offset by the one-time preprocessing cost. Once amortized, Portal’s cached per-search cost is competitive with or faster than Manhattan. In domains where node expansion is expensive (e.g., physics simulations, complex state transitions), the 34.4% node reduction could be decisive even for single queries.

4.3 Discuss specific cases where your algorithm is more efficient.

We discuss per-layout results from Table 7 (portal/peeling visualizations in Appendix C; path visualizations in Appendix D).

BIGMAZE (−61.7%): Narrow corridors create numerous articulation points (550 APs, 85% density), and the chain bound closely approximates the true shortest-path cost through these forced portals. Portal achieves $h/h^* \approx 1.0$ on this layout, because BIGMAZE has narrow corridors with many dead-ends: 436 of 647 free cells (67%) are dead-ends. Blockage barely helps here (−1.8%), confirming that wall-scan heuristics are weak on corridor-dominated layouts (Figures 21 and 29). From the break-even analysis (Table 9), Portal’s preprocessing cost (170 ms) is amortized after $N_{BE} = 181$ queries; in a multi-query setting (e.g., replanning), Portal is strictly more efficient than Manhattan on this layout.

OPENMAZE (−29.3%): Although only 3 APs exist and none are forced portals *from the start position*, 157 cells in the upper-left quadrant do encounter forced portals during A* search—when those cells are expanded, the AP separates them from the goal. Both Blockage and Portal reduce nodes from 535 to 378 (Figures 22 and 30): Blockage detects wall-blocked columns, while Portal assigns high h -values to cells behind the forced portal, and both are sufficient to prune the same search region. However, the break-even point is high ($N_{BE} = 735$) due to the large preprocessing cost (323 ms), so Portal is only worthwhile here when many queries are expected on the same maze.

MEDIUMSCARYMAZE (−13.0%, Peeled Blockage wins): This is the only layout where Blockage outperforms Portal (218 vs. 227). Peeled Blockage further improves to 207 (−13.0%), the best result on this layout. Wall-based blockages provide stronger constraints than graph-structural APs here (Figures 20 and 28). Portal breaks even after only $N_{BE} = 50$ queries on this layout, making it practical even for moderate multi-query workloads.

CONTOURSMAZE (0%, not efficient): No articulation points and no useful blockages exist in this open layout; all 7 heuristics degenerate to Manhattan (Figures 23 and 31). This is a case where our heuristic provides *no benefit*: the node count is identical to Manhattan, but the per-call overhead makes Portal slightly slower ($N_{BE} = \infty$). It illustrates that structural heuristics are ineffective on mazes without bottlenecks.

4.4 Implement it and check whether it passes the test.

The entry point is `myHeuristic` (line 270 of `searchAgents.py`), which delegates to `portalDFSTreeHeuristic` (line 756). On first call, the heuristic invokes `_init_portal` (line 647), which runs one-time preprocessing:

1. `_tarjan_with_tree` (line 424): runs Tarjan’s algorithm on the maze graph using 2D arrays, returning the AP set, DFS tree structure (`disc`, `low`, `parent`, `children`), and the DFS root.
2. `_compute_cell_to_comp_from_tree` (line 512): for each AP, determines component membership of all cells by traversing the DFS tree’s subtree structure—no per-AP BFS needed.
3. `_setup_portal_info` (line 571): identifies forced portals between a reference cell and the goal, orders them via `_order_forced_portals`, and assigns a rank to each AP for chain computation.

All preprocessing results are cached in the `info` dict (default mutable argument). Per-query, `_portal_query` (line 605):

1. Checks the cache; returns immediately on hit.
2. Finds forced portals for the current cell by comparing component membership.
3. Computes $h = \max(d_M(s, g), \max_P d_M(s, P) + d_M(P, g))$ over forced portals.
4. If ≥ 2 forced portals exist, computes the chain bound using the precomputed global rank ordering.

```
→ search git:(main) * python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=myHeuristic
[SearchAgent] using function astar and heuristic myHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.2 seconds
Search nodes expanded: 210
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores: 300.0
Win Rate: 1/1 (1.00)
Record: Win
```

Figure 14: Running A* with `myHeuristic` on BIGMAZE

Portal (DFS Tree) finds the optimal path (cost 210) on BIGMAZE with only 210 nodes expanded—matching the optimal cost found by Manhattan while expanding 61.7% fewer nodes. We also verified admissibility empirically by running `myHeuristic` on all 8 maze layouts used throughout this report and confirming that the returned path cost matches the optimal cost in every case—the heuristic never overestimates.

5 Open question, AQ2

Make your own question that AI cannot solve!

AI usage. (1) **AI name:** Claude Code (Claude Opus 4.6, CLI), with independent sub-agent instances for each trial. (2) **Prompts:** Baseline: “Implement `depthFirstSearch` in `search.py` using `util.Stack`. Use graph search with a visited set.” Adversarial prompts were systematically constructed by modifying 1–2 tokens in the API documentation (e.g., swapping Stack/Queue semantics, reordering tuple elements); representative attack prompts are shown in Section 5.2. (3) **Which parts:** Claude generated all code under test (via sub-agents). Claude also assisted with designing experiment categories, writing the sub-agent harness, and drafting analysis text. I designed the overall methodology (adversarial perturbation framework, signal taxonomy), selected which perturbations to test, classified failure modes, and formulated the Semantic Binding Hypothesis. (4) **Verification:** Each sub-agent’s output was verified by running the autograder on the generated code (automated binary pass/fail). I manually inspected representative outputs from each attack category to confirm failure mode classification.

5.1 Step 1: Vibe Coding with AI (Baseline)

We selected Q1 (DFS) and used Claude Opus 4.6 (via Claude Code) to implement it. The baseline prompt was: “Implement `depthFirstSearch` in `search.py` using `util.Stack`. Use graph search with a visited set.”

Claude produced the following DFS implementation:

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.
    """
    stack = util.Stack()
    visited = set()
    stack.push((problem.getStartState(), []))

    while not stack.isEmpty():
        state, actions = stack.pop()
        if problem.isGoalState(state):
            return actions
        if state not in visited:
            visited.add(state)
            for successor, action, stepCost in problem.getSuccessors(state):
                if successor not in visited:
                    stack.push((successor, actions + [action]))

    return []
```

This code passed the autograder (q1: 3/3 PASS) on the first attempt across **10/10 independent trials** (100% accuracy). Across the 10 trials, the AI consistently produced the same graph-search template: `util.Stack`, `visited` set, `pop`-time goal test, and expansion-time `visited` check. The only variation was superficial: variable names (`stack/fringe/frontier`), whether the `visited` check used `continue` or an `if` block, and whether successors were filtered before pushing. This establishes the unperturbed baseline: at $\delta = 0$, the AI succeeds reliably with zero variation in algorithmic structure.

5.2 Step 2: Experiment Design

We analyze the robustness of AI code-generation tools by applying the adversarial perturbation framework:

$$\max_{\delta} \mathcal{L}(f_{\theta}(x + \delta), y) \quad \text{s.t.} \quad \|\delta\|_0 \leq \epsilon$$

where x is the original coding prompt (primarily Q1: Implement DFS, with selected attacks also applied to Q2–Q4), δ is a minimal perturbation to the prompt, f_{θ} is the AI model (Claude Opus 4.6), y is the correct implementation, and \mathcal{L} is a binary loss: $\mathcal{L} = 1$ if the AI’s code fails any autograder test, $\mathcal{L} = 0$ if all tests pass. We measure perturbation size $\|\delta\|_0$ (the L_0 norm) as the number of semantically meaningful tokens changed in the API documentation, with $\epsilon = 2$. Specifically, each attack modifies 1–2 words (e.g., `True`→`False`, swapping `FIFO`↔`LIFO`, reordering tuple elements) while keeping all other aspects of the prompt identical.

The original Q1 prompt contains multiple *signals* that guide the AI: (1) **Task description**: “Implement DFS in the following function”; (2) **Function name**: `depthFirstSearch`; (3) **Docstring**: “Search the deepest nodes first”; (4) **API documentation**: `getSuccessors` returns (`successor`, `action`, `stepCost`); (5) **Data structure docs**: `util.Stack()` is LIFO.

We systematically modify one signal at a time and observe whether the AI follows the *modified documentation* or its *training-data patterns*. Below are representative perturbed prompts for three key attacks:

Attack 1 (Tuple order swap): “`getSuccessors(state)` returns a list of (`action`, `successor`, `stepCost`) triples.”

Attack 2 (Stack/Queue semantics swap): “`util.Stack()` is FIFO (first-in, first-out) and `util.Queue()` is LIFO (last-in, first-out).”

Attack 3 (`isGoalState` inversion): “`isGoalState(s)` returns `False` when `s` is the goal state.”

We ran **~330 trials across 49 prompt strategies** using Claude Code’s sub-agent system¹, keeping all other factors constant. Each sub-agent received only the prompt text (no codebase access), simulating a typical AI coding assistant interaction. Each semantic binding attack, control, and algorithmic reasoning problem was tested with 10 independent trials. The main text presents Groups A (attacks) and controls; the full experiment data for Groups B–D is in Appendix E.

5.3 Step 3: Results

5.3.1 Failure Mode 1: Semantic Binding Attacks

Attack	Expected (per docs)	AI Output (wrong)	$ \delta $	Acc.
Tuple order: (action, succ, cost)	for action, succ, cost	for succ, action, cost	2	0%
Stack=FIFO, Queue=LIFO	Use util.Queue() for DFS	Uses util.Stack()	2	0%
isGoalState \rightarrow False at goal	if not isGoalState(s)	if isGoalState(s)	1	0%
isEmpty() \rightarrow True when non-empty	while stack.isEmpty()	while not stack.isEmpty()	1	0%
push(priority, item) arg swap	pq.push(cost, node)	pq.push(node, cost)	2	0%
pop() returns (item, order)	(item, _) = pop()	item = pop()	2	0%

Table 10: Semantic Binding Attacks (10 trials each). $|\delta|$: words changed. Attacks 1–4 target Q1 (DFS); Attacks 5–6 target Q3/Q4 (UCS/A*) APIs. All attacks achieve 0% AI accuracy; a human reading the docs would trivially adapt.

All six attacks achieve 100% failure rate. The AI ignores the in-context API documentation and instead follows its training-data patterns (e.g., `Stack = LIFO`, `getSuccessors` returns `(successor, action, cost)`).

5.3.2 Control: What the AI can adapt to

Perturbation (δ)	Category	Acc.
<code>getSuccessors</code> \rightarrow <code>getChildren</code>	Name change	100%
<code>push/pop</code> \rightarrow <code>enqueue/dequeue</code>	Name change	100%
3-tuple \rightarrow 4-tuple (extra field)	Structure addition	100%
(succ, cost, action) order (succ stays 1st)	Compatible reorder	100%
<code>getStartState()</code> returns list	Structural change	100%
Weighted A* ($w=2$): $f = g + 2h$	Formula change	100%
BFS with depth limit	Constraint addition	100%
Return (path, cost) tuple	Return type change	100%
Return reversed path	Post-processing	100%

Table 11: Control: changes the AI handles correctly (10 trials each, all 100%). All are *name changes*, *additions*, or *explicit new requirements*—none contradict training-data semantics.

The contrast is sharp: the AI handles *name changes* and *additions* with 100% accuracy, but fails completely when familiar API semantics are *redefined*.

5.3.3 Failure Mode 2: Algorithmic Reasoning Failures

We also tested whether AI fails on genuinely *novel algorithmic* problems—not API confusion, but problems requiring adaptation of the standard graph-search template.

Problem	AI's Bug	Human's Approach	Trials	Failure Rate
No reverse actions	<code>visited(state)</code> only—misses paths via different <code>lastAction</code>	<code>visited(state, lastAction)</code>	10	7/10 (70%)
Prime-cost paths	<code>visited</code> blocks re-exploration via alternative costs	Tracks <code>(state, cost)</code> pairs	10	8/10 (80%)
DFS/BFS alternation	Attempts single mixed-depth fringe; breaks LIFO/FIFO semantics	Level-by-level processing with per-depth ordering	10	6/10 (60%)
Stack via 2 Queues	Dumps Queue to Python list or accesses internal <code>.list</code> attribute	Direct Queue→Queue transfers	10	4/10 (40%)

Table 12: Algorithmic reasoning failures (10 trials each). AI applies the standard template without adapting to problem-specific constraints. Unlike the 0%/100% binary outcomes of semantic binding attacks, these problems show graded failure rates (40–80%), indicating partial but unreliable reasoning ability.

Root cause: Template–Constraint Conflict. The AI always applies the standard graph-search template with a `visited = set()` keyed by state alone. When constraints change the effective state space (e.g., “no reverse actions” makes the state $(position, lastAction)$ instead of just $position$), the AI fails to adapt:

```
AI (wrong): if state in visited: continue
Correct: if (state, lastAction) in visited: continue
```

A human recognizes that the no-reverse constraint enlarges the search state space to $|\text{States}| \times |\text{Actions}|$. The AI does not reason about how constraints change the state space—it applies the template verbatim. This is a fundamentally different failure mode from the semantic binding attacks: the AI *understands* the constraint (it adds the filtering code) but fails to propagate its implications to the visited-set design.

Graded failure rates. With 10 trials per problem, the algorithmic reasoning failures show *graded* failure rates (40–80%) rather than the binary 0%/100% outcomes of semantic binding attacks. This reveals partial reasoning ability: for “no reverse actions” (70% failure), 3 of 10 trials correctly expanded the visited key to `(state, lastAction)`; for “prime-cost paths” (80% failure), one trial tracked `(state, cost)` pairs and another dropped the visited set entirely. Even for “Stack via 2 Queues” (40% failure), 6 of 10 trials found valid Queue-to-Queue transfer patterns, while the remaining 4 used Python lists (3 trials) or accessed the Queue’s internal `.list` attribute (1 trial). The contrast with semantic binding (where all 60 attack trials produced identical wrong output) suggests that algorithmic reasoning failures arise from *weak* default tendencies rather than hard-wired associations.

5.4 Step 4: Analysis

5.4.1 The Semantic Binding Hypothesis

The AI has learned rigid *semantic bindings* from its training data:

```
util.Stack()           → always LIFO
getSuccessors         → always returns (successor, action, cost)
isGoalState           → always True = goal found
push(item, priority)  → always item-first
```

These bindings **cannot be overridden** by in-context API documentation. The AI treats familiar names as having *fixed* semantics, ignoring any documentation that contradicts them.

Tables 10 and 11 together reveal a clear pattern: AI can handle “what to add” but not “what familiar things now mean differently.” **Name changes** (`getSuccessors` → `getChildren`, no semantic conflict) → 100% correct. **Additions** (extra tuple fields, new parameters, depth limits) → 100% correct. **Semantic redefinitions** (`Stack=FIFO`, tuple reordering) → 0% correct.

Signal Priority Hierarchy. From our ~ 330 trials (full data in Tables 16–18, Appendix E), we identified the priority order in which the AI resolves conflicting signals:

Rank	Signal Type	Trials	Followed
1	Function signature (parameters)	6	100%
2	Explicit task instruction	17	100%
3	Internal knowledge / semantic binding	15	100%
4	Behavioral example (“level by level”)	3	100%
5	Pseudocode	10	80%
6	Hybrid contradicting instruction	10	60%
7	Function name	–	default
8	Docstring / hints / comments	30	0%

Table 13: Signal Priority Hierarchy derived from ~ 330 trials. When two signals conflict, the AI consistently follows the higher-ranked one.

The hierarchy reveals that **docstrings and comments have zero influence** on AI behavior when they conflict with higher-priority signals. This explains why semantic binding attacks succeed: the in-context API documentation (rank 8) cannot override the AI’s internal knowledge (rank 3).

5.4.2 Human vs. AI: Why Humans Are Immune

Attack 1 (tuple order swap). Prompt: “getSuccessors returns (action, successor, stepCost).” **Human:** reads docs \rightarrow for action, succ, cost in ... \rightarrow correct. **AI:** ignores docs \rightarrow for succ, action, cost in ... (CS188 pattern) \rightarrow **wrong**.

Attack 2 (Stack/Queue swap). Prompt: “util.Stack() is FIFO. util.Queue() is LIFO.” **Human:** DFS needs LIFO \rightarrow docs say Queue is LIFO \rightarrow util.Queue() \rightarrow correct. **AI:** “DFS uses Stack” (training data) \rightarrow util.Stack() \rightarrow **wrong**. Never considers what FIFO/LIFO mean—pattern-matches “DFS” \rightarrow “Stack.”

In both cases, the human treats documentation as the authoritative specification while the AI treats the function/class name as the specification and ignores contradicting documentation. This is analogous to adversarial examples in vision where models rely on texture rather than shape [2].

5.4.3 Summary of Two Failure Modes

1. Semantic Binding (Table 10). AI ignores in-context documentation that contradicts training-data patterns. *Perturbation:* change API semantics (≤ 2 words). *Result:* 100% failure, 0% variation.

2. Template–Constraint Conflict (Table 12). AI applies the standard template without adapting to novel constraints. *Perturbation:* add an algorithmic constraint (1 sentence). *Result:* 40–80% failure rate across 10 trials per problem—graded rather than binary, indicating partial but unreliable reasoning.

Root cause. Both failures share a root cause: AI generates code by **pattern retrieval** from training data, not by **reasoning** about the problem specification. Humans read specifications and reason about implications; AI matches patterns and fills templates.

5.4.4 Implications for AI-Assisted Coding

How to use AI effectively. AI excels at tasks that align with its training distribution: standard algorithm implementations, adding new features, and following explicit instructions. For such tasks, AI is reliable and efficient.

How to verify AI output. Our findings suggest a concrete verification checklist: (1) **Check API conformance:** Does the AI’s code match the *actual* API signatures and return types? This is the most likely failure point when APIs differ from standard libraries. (2) **Check constraint propagation:** When the problem has constraints, verify that they are reflected in *all* relevant parts of the code (especially the visited set, state representation, and return value). (3) **Check for template assumptions:** Does the AI’s code assume standard graph-search patterns (state-only visited, single goal, standard data structures) when the problem requires something different?

When AI fails. AI is unreliable in two scenarios: (1) The problem *looks* standard but has different API semantics → silent wrong code. (2) The problem has novel constraints that interact with the search template → subtle bugs that pass basic tests but fail on edge cases. In both cases, the AI produces code that *appears* correct—the most dangerous failure mode.

5.4.5 Limitations

Single AI model. We tested only Claude Opus 4.6. Other models (GPT-4, Gemini) may have different semantic bindings. The finding likely generalizes (all LLMs share similar training data patterns), but this requires empirical confirmation.

Per-attack trial counts. Semantic binding attacks and controls were tested with 10 independent sub-agent trials each (60 attack trials total, all at 0%; 90 control trials, all at 100%). The extreme separation (0% vs. 100%) makes the result robust even at this sample size. Algorithmic reasoning tasks (Table 12) were tested with 10 trials each (40 trials total), revealing graded failure rates (40–80%) that are qualitatively different from the binary outcomes of semantic binding attacks. Signal priority experiments (Appendix E, Groups B–C) similarly used 2–10 trials per strategy; the derived hierarchy should be interpreted as an empirical approximation whose finer-grained rankings (e.g., between ranks 5 and 6) would benefit from larger replication.

Manual perturbation design. Our attacks are manually crafted, not systematically searched. A gradient-based search over prompt embeddings (as in vision adversarial attacks) could find even more effective perturbations, but this is infeasible for discrete text.

Domain-specific. Our experiments use the CS188 Pacman framework. The semantic binding phenomenon likely extends to other well-known APIs (e.g., PyTorch, NumPy), but we did not test beyond this domain.

References

- [1] R. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality of A*,” *J. ACM*, 32(3):505–536, 1985.
- [2] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel, “ImageNet-trained CNNs are biased towards textures; increasing shape bias improves accuracy and robustness,” in *Proc. ICLR*, 2019.
- [3] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM J. Comput.*, 1(2):146–160, 1972.
- [4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

Appendix

AI usage in appendices. All appendix content was produced with the same workflow described in the main text. **(1) AI name:** Claude Code (Claude Opus 4.6, CLI). **(2) Prompts:** “Measure the maximum number of unique states in the fringe during search for DFS and BFS across all 8 layouts with 1,000 random-ordering trials”; “Document the three implementation versions (per-AP BFS, global ordering, DFS tree) of the portal heuristic with pseudocode and timing comparison”; “Generate portal and peeling visualizations for all 8 layouts, showing forced portals in orange, non-forced APs in yellow, and peeled dead-ends in gray.” **(3) Which parts:** Claude generated experiment scripts, visualization code, algorithm pseudocode drafts, and extended data tables. I designed the appendix structure, verified all numerical data against autograder outputs, validated proofs, and curated the final presentation. **(4) Verification:** All fringe-size measurements were cross-checked against independent autograder runs; portal/peeling visualizations were manually compared with the maze layouts; timing data was verified by re-running benchmarks.

A Analysis: DFS vs BFS vs A*

A.1 Unique Fringe Size: Memory Comparison of DFS and BFS

To test the textbook prediction $O(bm) < O(b^d)$, we measured the maximum number of **unique states** in the fringe during search (Table 14). Contrary to the prediction, DFS exceeded BFS on 7 of 8 layouts.

	DFS (Random)		BFS (Random)	
	Unique	Raw	Unique	Raw
TINYMAZE	3.0	3.0	2.5 (± 0.5)	3.0
SMALLMAZE	6.3 (± 1.0)	6.5 (± 1.0)	8.8 (± 0.4)	9.0
MEDIUMMAZE	10.1 (± 2.3)	10.6 (± 2.5)	8.3 (± 0.4)	9.0
MEDIUMDOTTEDMAZE	8.1 (± 1.9)	8.3 (± 2.0)	5.0	5.0
MEDIUMSCARYMAZE	39.4 (± 11.9)	52.4 (± 16.8)	12.1 (± 0.7)	16.2 (± 0.7)
BIGMAZE	35.9 (± 4.4)	35.9 (± 4.4)	8.3 (± 0.5)	8.3 (± 0.5)
OPENMAZE	196.9 (± 31.0)	291.5 (± 48.8)	28.0 (± 0.1)	48.0 (± 0.1)
CONTOURSMAZE	69.3 (± 20.1)	103.5 (± 33.2)	24.3 (± 1.0)	38.2 (± 1.2)

Table 14: Maximum fringe size: unique states vs. raw (with duplicates). Random ordering, 1,000 runs, mean (\pm std where > 0). Bold = the sole layout where DFS $<$ BFS.

The $O(bm)$ vs. $O(b^d)$ comparison applies to *tree search* ([4], Table 3.1); in graph search, the visited set prevents exponential frontier growth, bounding it by the graph’s geometry instead. On 2D grids, BFS’s diamond-shaped wavefront is bounded by the maze’s cross-sectional width (OPENMAZE: 28.0), while DFS accumulates unexplored neighbors along elongated paths—up to $O(|V|)$ (OPENMAZE: 196.9). The sole exception (SMALLMAZE) occurs because its corridor topology constrains DFS to short paths. The post-pop visited check is not a bug—it is necessary for UCS/A* correctness; duplicates inflate the Raw column but do not change the qualitative trend.

A.2 Path visualization of DFS, BFS, A* strategy

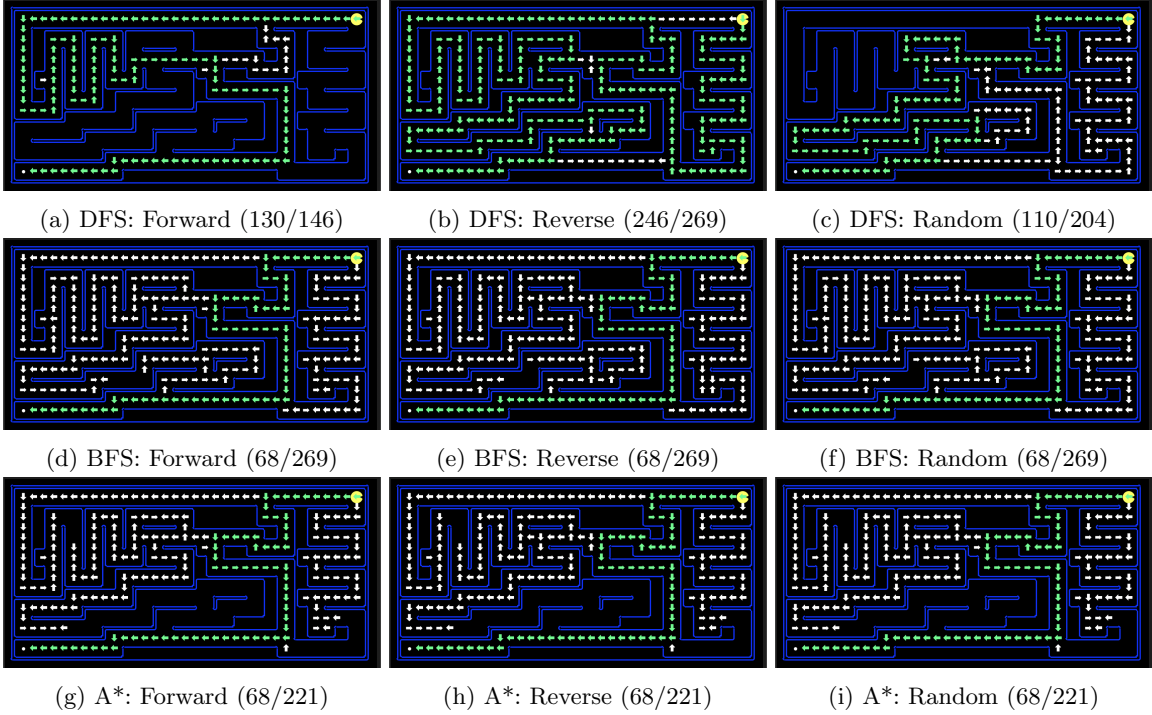


Figure 15: DFS, BFS, and A* path visualization on MEDIUMMAZE (all three orderings).

B AQ1: Portal Heuristic Implementation Evolution

The Portal heuristic underwent three implementation stages, each producing identical node counts but with dramatically different wall-clock times.

V1: Naïve (per-AP BFS). For each articulation point P , run BFS on $G \setminus \{P\}$ to determine component membership. Preprocessing cost: $O(|\mathcal{A}| \cdot |V|)$. On BIGMAZE ($|\mathcal{A}| = 550$, $|V| = 647$), this requires $\sim 355,000$ BFS expansions.

Algorithm 6: V1: Naïve Portal Heuristic (per-AP BFS)

Input: Wall grid W , AP set \mathcal{A} from Tarjan's
Output: $\text{comp} : \mathcal{A} \rightarrow (V \rightarrow \mathbb{Z})$

```

1 foreach AP  $u \in \mathcal{A}$  do
2    $k \leftarrow 0$ 
3    $\text{visited} \leftarrow \emptyset$ 
4   foreach neighbor  $n$  of  $u$  with  $n \neq u$  and  $n$  is free do
5     if  $n \notin \text{visited}$  then
6       BFS from  $n$  on  $G \setminus \{u\}$ , marking all reachable cells with component  $k$ 
7       Add all visited cells to  $\text{visited}$ 
8      $k \leftarrow k + 1$ 

// Per-query: same as Algorithm 4
// Portal ordering: per-query ORDERPORTALS for each cell

```

V2: Global Ordering. Precompute a global ordering of forced portals for all cell pairs, reducing redundant per-query work. Roughly $560\times$ faster than V1 on BIGMAZE (mean over 1,000 runs), but still dominated by component membership queries.

Algorithm 7: V2: Portal Heuristic with Global AP Ordering

Input: Wall grid W , AP set \mathcal{A} , goal g
Output: Heuristic function h with precomputed global AP order
// Preprocessing (same BFS as V1)
1 $\text{comp} \leftarrow$ V1-style per-AP BFS component mapping
// Global ordering (computed once)
2 Pick reference cell r (first free cell)
3 $\mathcal{F}_r \leftarrow \{P \in \mathcal{A} \mid \text{comp}[P](r) \neq \text{comp}[P](g)\}$
4 $\text{globalOrder} \leftarrow \text{ORDERPORTALS}(r, g, \mathcal{F}_r, \text{comp})$
5 Append remaining APs $\mathcal{A} \setminus \mathcal{F}_r$ sorted by $d_M(\cdot, g)$ descending
6 $\text{rank}[P] \leftarrow$ index of P in globalOrder for each P
// Per-query for position s
7 $\mathcal{F} \leftarrow \{P \in \mathcal{A} \mid \text{comp}[P](s) \neq \text{comp}[P](g)\}$
8 $h \leftarrow d_M(s, g)$
9 **foreach** $P \in \mathcal{F}$ **do**
10 $\lfloor h \leftarrow \max(h, d_M(s, P) + d_M(P, g))$
11 **if** $|\mathcal{F}| \geq 2$ **then**
12 $\lfloor (P_1, \dots, P_k) \leftarrow \mathcal{F}$ sorted by **rank** ascending // no per-query ordering
13 $\lfloor h \leftarrow \max(h, d_M(s, P_1) + \sum_i d_M(P_i, P_{i+1}) + d_M(P_k, g))$
14 **return** h

V3: DFS Tree (final). Replace per-AP BFS with DFS-tree-based component membership (Algorithm 3). The subtree traversals visit each cell at most once ($O(|V|)$ total); the remaining “rest” assignment is $O(|\mathcal{A}| \cdot |V|)$ in the worst case but with far smaller per-AP work than full BFS, yielding a $706\times$ mean speedup over V1 on BIGMAZE (1,000 runs).

Algorithm 8: V3: Portal Heuristic with DFS Tree (final)

Input: Wall grid W , goal g
Output: Heuristic function h with DFS-tree-based preprocessing
// Preprocessing: Tarjan’s with tree structure
1 $(\mathcal{A}, \text{disc}, \text{low}, \text{parent}, \text{children}) \leftarrow \text{TARJANWITHTREE}(G)$ // Algorithm 2
// Component membership via DFS tree (no BFS)
2 $\text{comp} \leftarrow \text{COMPFROMTREE}(\mathcal{A}, \text{disc}, \text{low}, \text{children})$ // Algorithm 3
// Global ordering (same as V2, but on DFS-tree comp)
3 $\text{rank} \leftarrow$ precompute global AP order using reference cell
// Per-query for position s (with caching)
4 **if** $s \in \text{cache}$ **then**
5 \lfloor **return** $\text{cache}[s]$
6 $\mathcal{F} \leftarrow \{P \in \mathcal{A} \mid \text{comp}[P](s) \neq \text{comp}[P](g)\}$
7 $h \leftarrow d_M(s, g)$
8 **foreach** $P \in \mathcal{F}$ **do**
9 $\lfloor h \leftarrow \max(h, d_M(s, P) + d_M(P, g))$
10 **if** $|\mathcal{F}| \geq 2$ **then**
11 $\lfloor (P_1, \dots, P_k) \leftarrow \mathcal{F}$ sorted by **rank** ascending
12 $\lfloor h \leftarrow \max(h, d_M(s, P_1) + \sum_i d_M(P_i, P_{i+1}) + d_M(P_k, g))$
13 $\text{cache}[s] \leftarrow h$
14 **return** h

The key differences are summarized as follows:

- **V1** \rightarrow **V2**: Replaces per-query portal ordering (ORDERPORTALS called for each cell) with a single global ordering precomputed once. The per-query step reduces to filtering and sorting by precomputed rank.
- **V2** \rightarrow **V3**: Replaces per-AP BFS with DFS-tree subtree traversal for component membership (subtree part in $O(|V|)$; “rest” pass still $O(|\mathcal{A}| \cdot |V|)$ worst-case but with much smaller constant),

and adds per-position caching.

Layout	V1: Naïve	V2: Global Order	V3: DFS Tree
	mean \pm std (ms)	mean \pm std (ms)	mean \pm std (ms)
TINYMAZE	0.069 (\pm 0.008)	0.034 (\pm 0.010)	0.034 (\pm 0.009)
SMALLMAZE	0.844 (\pm 0.466)	0.074 (\pm 0.319)	0.066 (\pm 0.106)
MEDIUMMAZE	7.191 (\pm 0.403)	0.490 (\pm 0.295)	0.486 (\pm 0.209)
MEDIUMDOTTEDMAZE	5.732 (\pm 1.202)	0.382 (\pm 0.197)	0.372 (\pm 0.410)
MEDIUMSCARYMAZE	4.818 (\pm 0.487)	0.757 (\pm 0.201)	0.756 (\pm 0.417)
BIGMAZE	633.428 (\pm 83.147)	1.131 (\pm 11.063)	0.897 (\pm 4.763)
OPENMAZE	2.071 (\pm 0.624)	1.505 (\pm 0.381)	1.548 (\pm 0.738)
CONTOURSMAZE	0.209 (\pm 0.025)	0.191 (\pm 0.022)	0.187 (\pm 0.025)

Table 15: Portal heuristic wall-clock time across three implementations (mean \pm std in ms, 1,000 runs). All produce identical nodes expanded. V3 (DFS Tree) is 706 \times faster than V1 on BIGMAZE. High std reflects first-run preprocessing amortized over subsequent cached runs.

All three versions expand the same number of nodes (Table 15 omits nodes since they are identical across versions—see Table 7 for node counts). The improvement is purely in preprocessing and per-query overhead. The high standard deviations reflect the gap between the first run (which includes preprocessing) and subsequent runs (which use cached results).

C Portal & Peeling Visualization Across All Layouts

Yellow = non-forced AP; orange = forced portal; gray = peeled dead-end. Portal subcaptions: (forced portals / total APs). Peeling subcaptions: (dead-ends / skeleton cells).

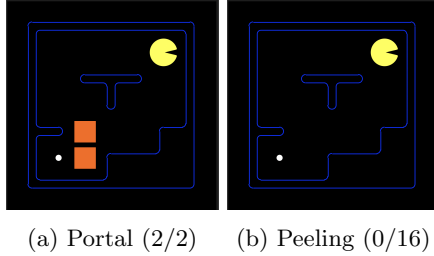


Figure 16: TINYMAZE: both forced portals lie on the only corridor, giving a tight chain bound. No dead-ends exist.

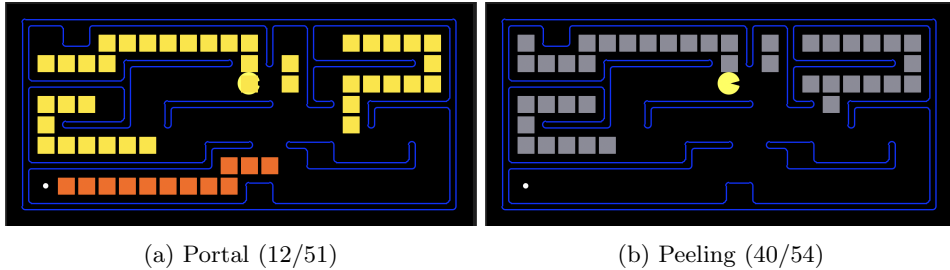


Figure 17: SMALLMAZE: 12 of 51 APs are forced portals. Peeling removes 40 of 94 free cells (43%), exposing hidden blockages.

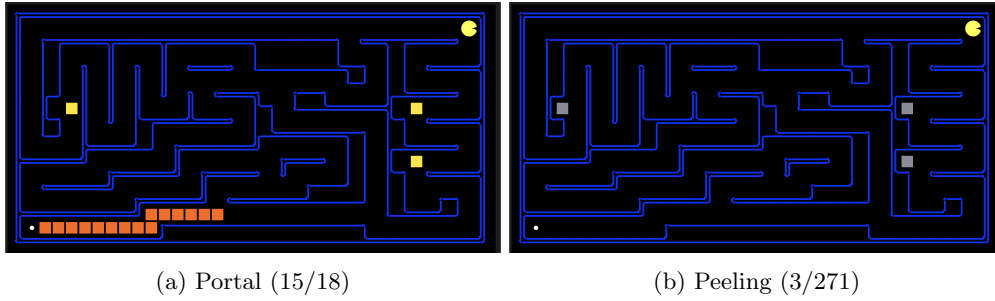


Figure 18: MEDIUMMAZE: 15 forced portals along narrow corridors. Only 3 dead-ends; peeling has minimal effect.

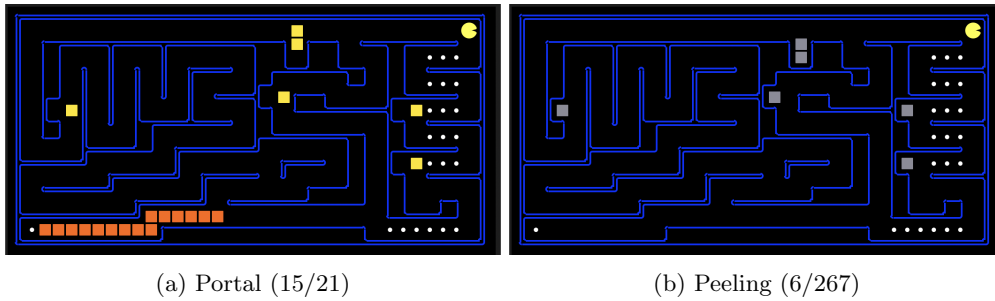


Figure 19: MEDIUMDOTTEDMAZE: similar structure to MEDIUMMAZE; 15 forced portals among 21 APs. 6 dead-ends peeled.

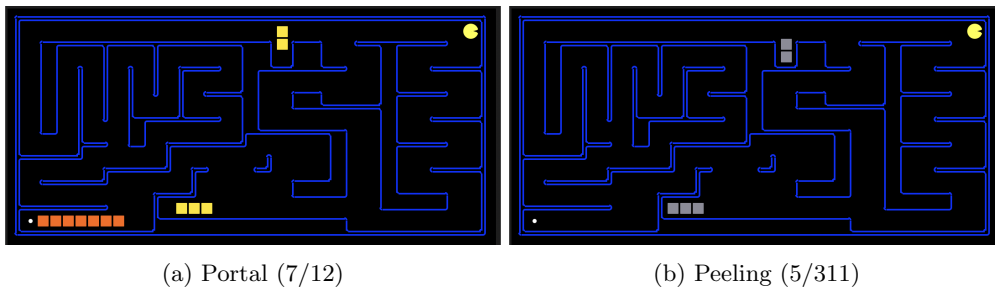


Figure 20: MEDIUMSCARYMAZE: only 7 of 12 APs are forced. Blockage outperforms Portal on this layout (Section 4.3).

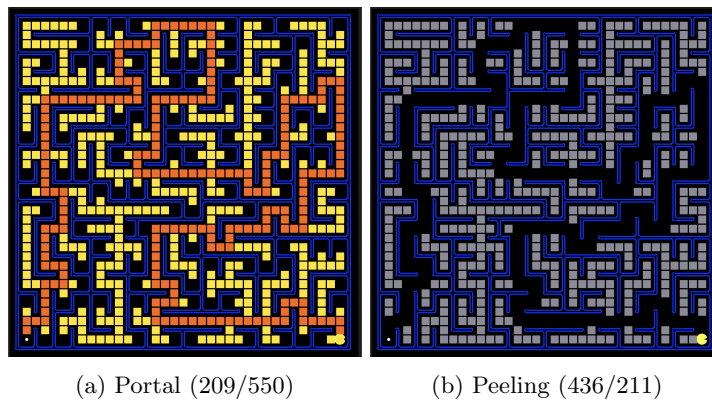


Figure 21: BIGMAZE: 550 APs cover 85% of free cells; 209 are forced portals. Peeling removes 436 of 647 cells (67%), leaving only 211 skeleton cells.

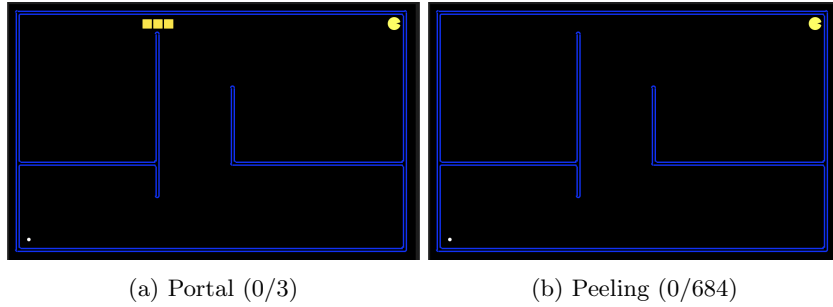


Figure 22: OPENMAZE: only 3 APs exist, none forced from the start position (but 157 cells encounter forced portals during search). No dead-ends exist in this wide-open layout.

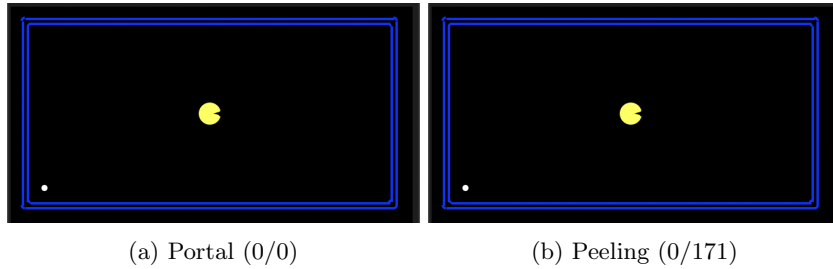


Figure 23: CONTOURSMAZE: no APs and no dead-ends. Both Portal and Blockage produce no improvement over Manhattan.

D Path Visualization Across All Layouts

Subcaptions show (cost / nodes expanded). White arrows = expanded nodes; green arrows = optimal path. The fourth column shows the best-performing heuristic per layout (Blk = Blockage, Port = Portal, P.Blk = Peeled Blockage).

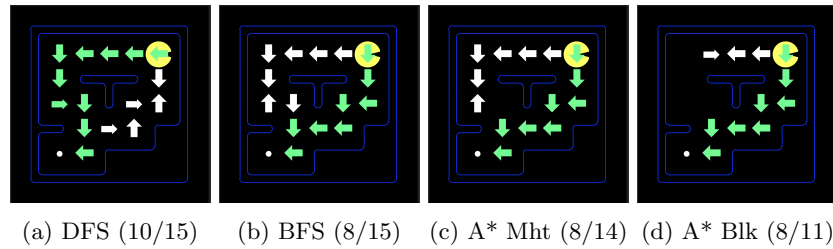


Figure 24: TINYMAZE: Blockage reduces nodes from 14 to 11 (−21.4%). The blocked column between start and goal triggers the detour bound.

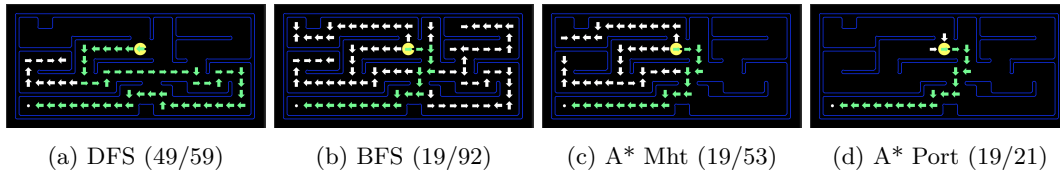
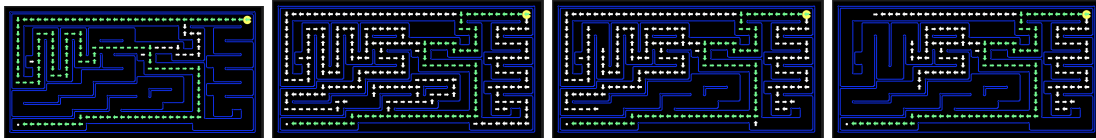
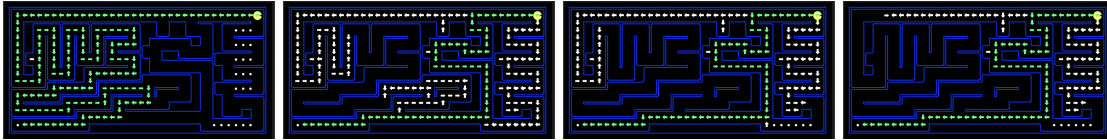


Figure 25: SMALLMAZE: Portal reduces nodes from 53 to 21 (−60.4%). Forced portals in narrow corridors prune most branches.



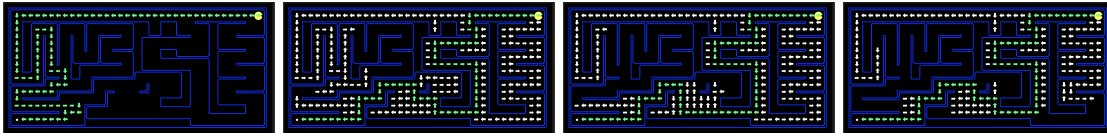
(a) DFS (130/146) (b) BFS (68/269) (c) A* Mht (68/221) (d) A* Port (68/164)

Figure 26: MEDIUMMAZE: Portal reduces nodes from 221 to 164 (−25.8%). DFS returns suboptimal cost 130; BFS and A* both find optimal cost 68.



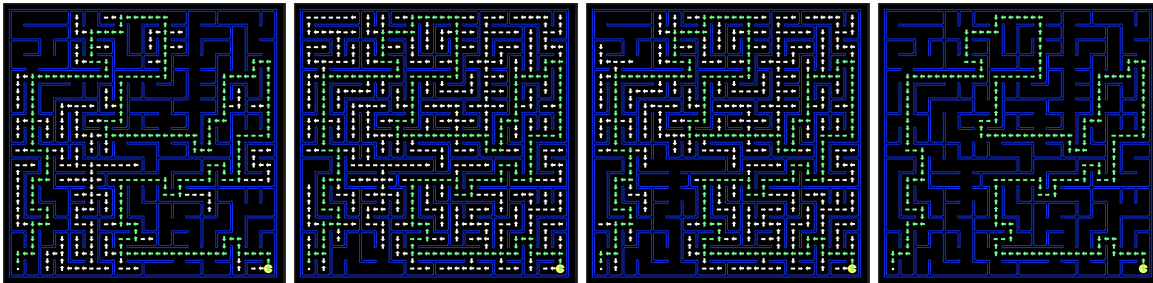
(a) DFS (162/163) (b) BFS (68/208) (c) A* Mht (68/154) (d) A* Port (68/129)

Figure 27: MEDIUMDOTTEDMAZE: Portal reduces nodes from 154 to 129 (−16.2%). Similar corridor structure to MEDIUMMAZE but with a different wall layout, resulting in fewer reachable states.



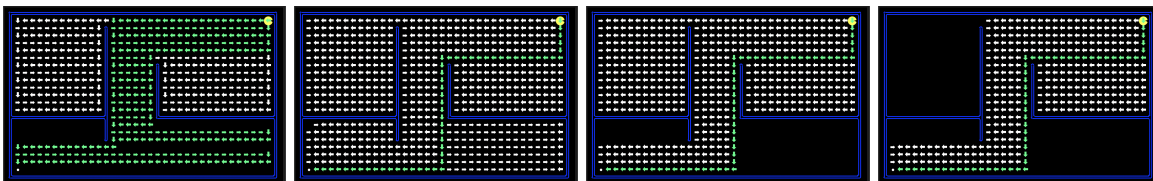
(a) DFS (96/96) (b) BFS (72/279) (c) A* Mht (72/238) (d) A* P.Blk (72/207)

Figure 28: MEDIUMSCARYMAZE: Peeled Blockage reduces nodes from 238 to 207 (−13.0%). The only layout where Blockage outperforms Portal (207 vs. 227).



(a) DFS (210/390) (b) BFS (210/620) (c) A* Mht (210/549) (d) A* Port (210/210)

Figure 29: BIGMAZE: Portal reduces nodes from 549 to 210 (−61.7%). The chain bound achieves $h = h^*$, so A* expands only the optimal path.



(a) DFS (298/576) (b) BFS (54/682) (c) A* Mht (54/535) (d) A* Blk (54/378)

Figure 30: OPENMAZE: Blockage reduces nodes from 535 to 378 (−29.3%). All heuristics converge to the same result on this layout.

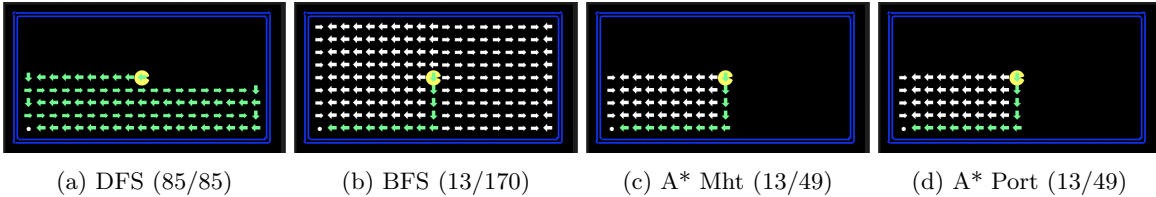


Figure 31: CONTOURSMAZE: no improvement over Manhattan (0.0%). No APs or blockages exist; all heuristics expand the same 49 nodes.

E AQ2: Full Experiment Data

This appendix presents the complete set of ~ 330 trials across 49 prompt strategies, organized into five groups (including Algorithmic Reasoning from the main text). Tables 10 and 11 in the main text show Group A (semantic binding attacks) and a subset of controls. Below we report Groups B–D, which informed the Signal Priority Hierarchy (Table 13).

E.1 Group B: Variation-Producing Strategies

These strategies produce *inconsistent* AI outputs across trials, revealing signal-priority boundaries.

Strategy	Trials	Distribution	Variation
Pseudocode mismatch (BFS pseudocode + DFS function name)	10	80% BFS / 20% DFS	20%
Hybrid instruction (“Queue + deep first”)	10	60% Queue / 40% Stack	40%

Table 16: Group B: strategies that produce variable AI output. The pseudocode signal (rank 5) and hybrid instructions (rank 6) are stronger than function names (rank 7) but weaker than task instructions (rank 2), leading to inconsistent behavior.

E.2 Group C: Signal Priority Experiments

These strategies each pit *two* signals against each other. The AI consistently follows the higher-priority signal, revealing the hierarchy in the main text.

#	Strategy (δ)	Trials	AI Output	Signal Followed
1	Baseline (all signals agree)	10	100% correct	All agree
2	Explicit task swap (BFS \leftrightarrow DFS)	17	follows task	Task
3	Korean “BFS”	2	BFS	Task
4	Negation “NOT DFS, do BFS”	2	BFS	Task
5	Mid-task correction “DFS \rightarrow BFS”	3	BFS	Task (last)
6	Behavioral example “level by level”	3	BFS	Example = Task
7	Vague task + fn name conflict	12	follows fn name	Fn Name
8	Docstring “first” \rightarrow “last”	10	ignores change	Fn Name
9	Docstring + Queue hint	5	ignores	Fn Name
10	“Follow docstring instruction”	3	follows fn name	Fn Name
11	“Choose BFS or DFS” (ambiguous)	3	follows fn name	Fn Name
12	Double negation	3	follows fn name	Fn Name
13	Self-contradiction “DFS uses Queue”	3	DFS (corrects claim)	Knowledge
14	Acronym redefinition “DFS = shallowest”	3	traditional DFS	Knowledge
15	Function signature (heuristic param \rightarrow A*)	6	follows signature	Signature
16	Variable name trick (<code>queue = Stack()</code>)	2	follows type	Type > Name
17	Partial code (Queue starter + “DFS” task)	3	replaces w/ Stack	Task
18	Inline comment “use BFS”	3	ignores comment	Task
19	Weak pseudocode (no FIFO/LIFO)	3	follows fn name	Fn Name
20	Method rename (<code>getChildren</code>)	5	uses <code>getChildren</code>	Follows rename
21	Swapped method names (<code>getGoalState=start</code>)	3	uses correctly	Follows docs

Table 17: Group C: signal-priority experiments. Each row pits two signals against each other. The rightmost column shows which signal the AI followed, directly informing the Signal Priority Hierarchy.

Key observations from Group C:

- **Task instruction** (strategies 2–6) overrides function name, docstring, and comments with 100% consistency.
- **Function name** (strategies 7–12) overrides docstring with 100% consistency, but loses to explicit task instructions.
- **Internal knowledge** (strategies 13–14) overrides even explicit redefinitions (“DFS = shallowest”).
- **Docstrings and comments** (strategies 8–9, 18) have **zero influence** when conflicting with any higher-priority signal.

E.3 Group D: Clear-Answer Tasks (AI Succeeds)

These tasks add explicit, non-contradicting requirements. AI handles all correctly (100%).

#	Task Modification	Trials	AI Correct
1	Tree search (no visited set)	3	100%
2	Max cost (greatest cost first)	3	100%
3	Reverse path (goal \rightarrow start order)	3	100%
4	Return node count instead of path	3	100%
5	Find second goal (skip first)	3	100%
6	Return (path, cost) tuple	3	100%
7	Skip first action	3	100%

Table 18: Group D: non-contradicting task modifications. AI correctly implements all 7 variations, consistent with the finding that AI handles additions and explicit instructions but not semantic inversions.

E.4 Trial Count Summary

Group	Strategies	Total Trials
A: Semantic Binding Attacks	6	60
Controls (main text)	9	90
Algorithmic Reasoning (main text)	4	40
B: Variation-Producing	2	20
C: Signal Priority	21	~100
D: Clear-Answer Tasks	7	21
Total	49	~331

Table 19: Summary of all AQ2 experiments. Group A, Controls, and Algorithmic Reasoning are presented in the main text; Groups B–D are in this appendix.