CS470 Coding Assignment 2

20266008 Doyeol Oh

1 Dry-run

Capture the result of `autograder.py` *in terminal.* Figure 1 shows that all required autograder tests (Q1–Q3) pass, yielding the expected 14/25.

```
Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 0/5
Question q5: 0/6
------------------
Total: 14/25
```

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

2 Implementation

2.1 Reflex Agent

The `ReflexAgent` evaluation function (Figure 2) builds on the raw successor score with three additive terms, each doing specific work.

```
"*** YOUR CODE HERE ***"
score = successorGameState.getScore()

# Food: prefer closer food (reciprocal of distance)
foodList = newFood.asList()
if foodList:
    minFoodDist = min(manhattanDistance(newPos, food) for food in foodList)
    score += 1.0 / (minFoodDist + 1)

# Ghosts: penalize being close to non-scared ghosts, reward chasing scared ghosts
for ghostState in newGhostStates:
    ghostPos = ghostState.getPosition()
    dist = manhattanDistance(newPos, ghostPos)
    if ghostState.scaredTimer > 0:
        score += 200.0 / (dist + 1)
    else:
        if dist < 2:
            score -= 500

return score
```

Figure 2: Implementation of `ReflexAgent.evaluationFunction`: raw successor score plus reciprocal food attraction, scared-ghost chasing, and a hard adjacency penalty for non-scared ghosts.

Design choices. (1) *Food attraction* (`score += 1.0 / (minFoodDist + 1)`) uses the reciprocal so the signal decays smoothly rather than treating all distant food equally; the $+1$ keeps it finite when Pacman sits on food. (2) *Ghost avoidance* is a hard -500 penalty gated by `dist < 2`, which avoids over-weighting distant ghosts that are not immediately threatening and matches the $\geq 25\%$ per-turn collision probability for adjacent random ghosts (analysed in Section 3.2). (3) *Scared-ghost chasing* contributes $+200/(d+1)$, a large positive term that makes Pacman exploit capsule power-ups instead of fleeing when the ghost is edible.

Verification. Running `python autograder.py -q q1` passes all 4 sub-tests.

2.2 Minimax

The `MinimaxAgent` uses a recursive helper `minimax(state, depth, agentIndex)` that generalises two-player minimax to n agents by cycling through `nextAgent = (agentIndex + 1) % numAgents` and treating agent 0 as the maximiser (Figure 3).

```
"*** YOUR CODE HERE ***"
def minimax(state, depth, agentIndex):
    if state.isWin() or state.isLose() or depth == 0:
        return self.evaluationFunction(state)

    numAgents = state.getNumAgents()
    nextAgent = (agentIndex + 1) % numAgents
    nextDepth = depth - 1 if nextAgent == 0 else depth

    actions = state.getLegalActions(agentIndex)

    if agentIndex == 0:  # Pacman (max)
        return max(minimax(state.generateSuccessor(agentIndex, a), nextDepth, nextAgent) for a in actions)
    else:  # Ghost (min)
        return min(minimax(state.generateSuccessor(agentIndex, a), nextDepth, nextAgent) for a in actions)

actions = gameState.getLegalActions(0)

bestAction = None
bestValue = float('-inf')
numAgents = gameState.getNumAgents()

for a in actions:
    v = minimax(gameState.generateSuccessor(0, a),
                self.depth - 1 if numAgents == 1 else self.depth,
                1 % numAgents)
    if v > bestValue:
        bestValue = v
        bestAction = a

return bestAction
```

Figure 3: Implementation of `MinimaxAgent.getAction`: the `nextDepth = depth - 1 if nextAgent == 0 else depth` line is the core of the ply-counting scheme.

Key detail. The critical line is `nextDepth = depth - 1 if nextAgent == 0 else depth`: the depth counter decrements only when the agent index wraps back to 0 (Pacman's turn), so one ply corresponds to Pacman's move *plus* all ghosts' responses, matching the assignment specification. A common mistake – decrementing per agent move instead of per wrap – produces a correct n-agent generalisation in type but the *wrong* reference values on MINIMAXCLASSIC; the line above is what makes the -492 value at depth 4 come out identical to the reference.

Verification. `python autograder.py -q q2` passes all 5 sub-tests. The root minimax value on `minimaxClassic` at depth 4 is -492, matching the reference value in the assignment.

2.3 Alpha-Beta Pruning

`AlphaBetaAgent` extends minimax with an (α, β) window that is threaded through the recursive helper, pruning a maximiser's remaining children when `v > beta` and a minimiser's when `v < alpha` (Figure 4).

```
"*** YOUR CODE HERE ***"
def alphaBeta(state, depth, agentIndex, alpha, beta):
    if state.isWin() or state.isLose() or depth == 0:
        return self.evaluationFunction(state)

    numAgents = state.getNumAgents()
    nextAgent = (agentIndex + 1) % numAgents
    nextDepth = depth - 1 if nextAgent == 0 else depth

    actions = state.getLegalActions(agentIndex)

    if agentIndex == 0:  # Pacman (max)
        v = float('-inf')
        for a in actions:
            v = max(v, alphaBeta(state.generateSuccessor(agentIndex, a), nextDepth, nextAgent, alpha, beta))
            if v > beta:
                return v
            alpha = max(alpha, v)
        return v
    else:  # Ghost (min)
        v = float('inf')
        for a in actions:
            v = min(v, alphaBeta(state.generateSuccessor(agentIndex, a), nextDepth, nextAgent, alpha, beta))
            if v < alpha:
                return v
            beta = min(beta, v)
        return v

actions = gameState.getLegalActions(0)

bestAction = None
bestValue = float('-inf')
alpha = float('-inf')
beta = float('inf')
numAgents = gameState.getNumAgents()

for a in actions:
    v = alphaBeta(gameState.generateSuccessor(0, a),
                  self.depth - 1 if numAgents == 1 else self.depth,
                  1 % numAgents,
                  alpha, beta)
    if v > bestValue:
        bestValue = v
        bestAction = a
    alpha = max(alpha, bestValue)

return bestAction
```

Figure 4: Implementation of `AlphaBetaAgent.getAction`: strict inequality pruning inside both the max and min branches, with α, β propagated through every ghost min layer.

Key detail. Pruning is performed only on strict inequality (`>` and `<`), not equality, to match the exact set of states explored by the autograder. AIMA's canonical pseudocode (§5.3) uses $\geq/\leq$ and is slightly more aggressive; both are correct in that they return the same minimax values, but they enumerate different sets of leaves on ties, and the CS188 autograder fixes the strict variant as the reference. Children are processed in the order returned by `getLegalActions`, with no reordering.

Verification. `python autograder.py -q q3` passes all 5 sub-tests.

3 Discussion Questions

3.1 How does the effectiveness of alpha-beta pruning depend on the ordering of actions?

How does the effectiveness of alpha-beta pruning depend on the "ordering of actions"? Provide concrete examples and explain why poor ordering can significantly reduce its benefits.

3.1.1 Theory

Alpha-beta pruning skips subtrees that cannot affect the minimax value. Whether a subtree is skipped depends on the values seen *before* it. In the maximiser:

$$\text{prune when } v > \beta \quad \text{(strict inequality; some implementations use } v \geq \beta),$$

where β is the best value the minimiser can guarantee from a higher node. Our implementation uses strict inequality to match the autograder's expected state counts (Section 2.3). If the maximiser sees its best child *first*, v rises quickly, making it likely that $v > \beta$ and subsequent children are pruned. Conversely, if the worst child is examined first, v stays low and few pruning opportunities arise.

In the best case (best moves always examined first) the search complexity reduces from $O(b^d)$ to $O(b^{d/2})$, effectively doubling the *effective search depth* for the same budget. In the worst case (worst moves first) the complexity remains $O(b^d)$, identical to plain minimax.

3.1.2 Toy Example

Figure 5 illustrates how ordering prevents pruning. Consider a max node with three children $[A, B, C]$, where the parent has passed down $\alpha = -\infty$, $\beta = 10$.

Best ordering ($A = 15$, $B = 8$, $C = 3$): after evaluating $A = 15$, we update $\alpha = 15$. Since $\alpha = 15 > \beta = 10$, the subtrees B and C are pruned. Two subtrees saved.

Worst ordering ($A = 3$, $B = 8$, $C = 15$): after $A = 3$ the bound is $\alpha = 3 < 10$, so B is evaluated; after $B = 8$ the bound is $\alpha = 8 < 10$, so C must be evaluated too. No pruning fires: all three subtrees explored.

Figure 5: Best ordering (left): after seeing value 15 with $\beta = 10$, children 2 and 3 are pruned (shown as ×). Worst ordering (right): all three children must be evaluated; no pruning fires.

The 1-level example above isolates *whether* pruning fires at a single layer. A second example is needed to show the mechanism behind the theoretical $O(b^{d/2})$ speed-up: pruning savings *compound across alternating max/min layers*, and a single-level tree cannot exhibit compounding. Figure 6 shows this two-level case. A MAX root with $\alpha = -\infty, \beta = +\infty$ has two MIN children, each with two leaves.

Best ordering (MIN children sorted so MAX's best subtree comes first; within each MIN node, smallest leaf first): MAX evaluates MIN_1: leaves $[2, 5]$, returns min $= 2$. Update $\alpha = 2$. MAX evaluates MIN_2: first leaf is $1 < \alpha = 2$, so $\beta_{\text{MIN}_2} = 1 < \alpha = 2$ – prune the second leaf. Result: **3 leaves evaluated** (out of 4), **1 pruned**.

Worst ordering (MIN children in reverse; within each MIN, largest leaf first): MAX evaluates MIN_2: leaves $[7, 1]$, returns 1. $\alpha = 1$. MAX evaluates MIN_1: leaves $[5, 2]$, first leaf $5 > \alpha = 1$, no pruning; second leaf 2, returns 2. Result: **4 leaves evaluated** (all of them), **0 pruned**.

The 2-level example exhibits *only one* compounding step (MAX → MIN → leaf), so the savings it demonstrates are modest (1/4 of leaves pruned). The true exponential gap $b^{d/2}$ vs b^d emerges only at depth $d = 4$ or more, where each additional MAX → MIN pair multiplies the pruning ratio. A fully worked 4-level example is impractical on paper (16 leaves vs 4), but the 2-level case is sufficient to see the mechanism: α tightened at MIN_1 is inherited by MIN_2, which tightens the pruning threshold for its own children; extrapolating this inheritance across more alternating layers gives $O(b^{d/2})$.

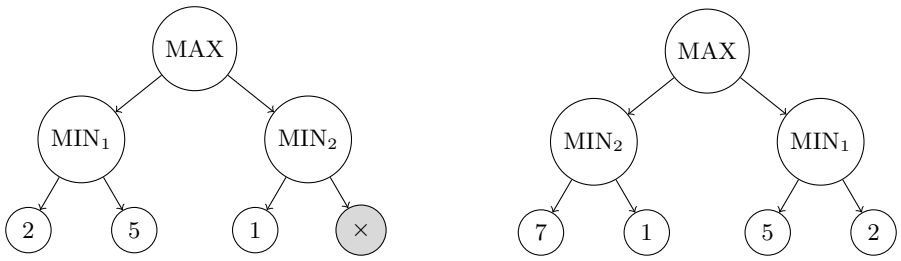


Figure 6: Two-level example. Best ordering (left): one leaf pruned (shown as ×). Worst ordering (right): all leaves examined.

3.1.3 Experimental Design

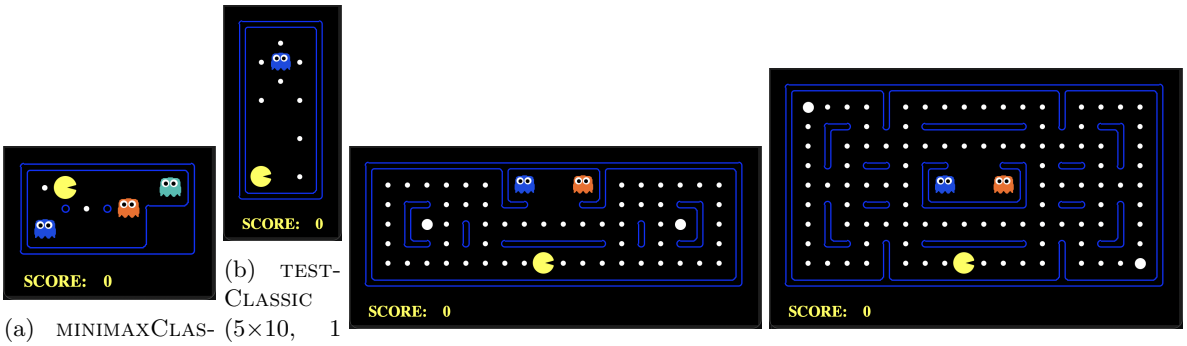
We implemented `OrderedAlphaBetaAgent`, a variant of `AlphaBetaAgent` that pre-sorts successor states before searching (Figure 7). Three orderings are compared:

- **default** – actions in the order returned by `getLegalActions` (N→S→E→W);
- **best** – successors pre-sorted by `scoreEvaluationFunction` in descending order at max nodes and ascending at min nodes (best-looking moves examined first);
- **worst** – the reverse (worst-looking moves examined first).

```
def sortActions(state, agentIndex, actions):
    if self.ordering == 'default':
        return actions
    reverse = (agentIndex == 0) if self.ordering == 'best' else (agentIndex != 0)
    return sorted(actions,
        key=lambda a: self.evaluationFunction(state.generateSuccessor(agentIndex, a)),
        reverse=reverse)
```

Figure 7: Implementation of `OrderedAlphaBetaAgent`: successor states are generated once, evaluated with `scoreEvaluationFunction`, and sorted before the (α, β) recursion – descending at max nodes and ascending at min nodes for "best", and the reverse for "worst". The (α, β) update logic itself is identical to `AlphaBetaAgent`, so any difference in node counts comes purely from the order in which children are visited.

Layouts. Three layouts of increasing size: TRAPPEDCLASSIC (8 × 5, 2 ghosts, 4 food), MINIMAXCLASSIC (9 × 5, 3 ghosts, 2 food), and SMALLCLASSIC (20 × 7, 2 ghosts, 55 food). Figure 8 shows these, along with MEDIUMCLASSIC and TESTCLASSIC used later in Section 3.2. TRAPPEDCLASSIC (not shown) is a very tight maze in which two ghosts surround Pacman in a narrow corridor (cf. Section 3.3's discussion of rushing behavior).



(a) MINIMAXCLASSIC (9×5, 3 ghosts) (b) TESTCLASSIC (5×10, 1 ghost) (c) SMALLCLASSIC (20×7, 2 ghosts) (d) MEDIUMCLASSIC (20×11, 2 ghosts)

Figure 8: Layouts used in the Discussion Questions experiments. All shown at a common vertical height, so per-cell size differs. MEDIUMCLASSIC and TESTCLASSIC appear only in Section 3.2. TRAPPEDCLASSIC (8×5, 2 ghosts) is not shown but has geometry similar to MINIMAXCLASSIC in an even tighter trap.

Configurations. Search depths 2–4 × 3 orderings = 9 configurations per layout. Each configuration runs $n = 100$ games against `RandomGhost`.

Metrics. Total nodes expanded per game, nodes expanded per Pacman turn (to isolate per-move search cost from game length), wall-clock time per game, win rate, and mean score.

3.1.4 Results

Tables 2, 3, and 4 report nodes expanded, wall-clock time, and game outcomes across all three layouts. The data reveal that ordering affects *two* distinct dimensions: (A) computational efficiency – the intended effect on node expansion and time – and (B) tie-breaking – an unintended effect on which action is selected when several share the same minimax value. Both must be read together.

Statistical significance of headline claims. With $n = 100$ per configuration, binomial win-rate differences have a normal-approximation standard error of $\mathrm{SE} = \sqrt{\hat{p}_1(1-\hat{p}_1)/n + \hat{p}_2(1-\hat{p}_2)/n}$, giving 95% CIs of roughly $\Delta \pm 2\,\mathrm{SE}$. Table 1 summarises every headline comparison cited in the analysis below; we flag the statistically ambiguous ones where they appear.

Table 1: Significance of the ordering comparisons discussed below. Win-rate rows use a two-proportion z-test ($n = 100$); the time-slowdown row uses a Welch t-test.

Layout / depth	Comparison	Δ	SE	95% CI	Verdict
smallClassic d2	default → best	+0.18	0.052	[+0.08, +0.28]	borderline
smallClassic d2	best → worst	−0.07	0.056	[−0.18, +0.04]	not significant
minimaxClassic d4	best → worst	−0.29	0.068	[−0.42, −0.16]	**significant**
trappedClassic d2	spread (3-way)	≤ 0.02	–	inside ±0.10 noise band	not significant
smallClassic d2	default → worst time	22×	–	Welch $t \gtrsim 5$	**significant**

Table 2: Alpha-beta ordering on minimaxClassic ($n = 100$, mean ± std).

Depth	Ordering	Nodes/game	Nodes/turn	Win rate	Score	Time (s)
2	default	573.9 ± 412.7	95.5 ± 26.9	0.40	-91.6 ± 495.1	0.019 ± 0.013
	best	608.9 ± 465.5	94.9 ± 23.3	0.40	-92.0 ± 494.2	0.037 ± 0.028
	worst	729.1 ± 250.8	88.8 ± 13.7	0.24	-257.0 ± 430.9	0.046 ± 0.018
3	default	2543.4 ± 1296.7	447.5 ± 127.6	0.46	-31.4 ± 501.2	0.077 ± 0.039
	best	2310.2 ± 1100.6	410.7 ± 125.5	0.43	-61.7 ± 498.1	0.141 ± 0.067
	worst	2520.8 ± 1251.1	467.5 ± 100.2	0.29	-202.8 ± 456.5	0.152 ± 0.079
4	default	7763.8 ± 4088.3	2122.3 ± 460.1	0.51	$+21.5 \pm 504.1$	0.237 ± 0.128
	best	7539.0 ± 4068.5	1919.7 ± 570.1	0.54	$+51.5 \pm 502.8$	0.436 ± 0.234
	worst	10045.8 ± 3762.6	2006.6 ± 510.4	0.25	-243.3 ± 435.8	0.603 ± 0.228

Table 3: Alpha-beta ordering on smallClassic ($n = 100$, mean ± std). Only depth 2 is reported; depth 3+ is intractable on this layout under "worst" ordering.

Depth	Ordering	Nodes/game	Nodes/turn	Win rate	Score	Time (s)
2	default	$22{,}447 \pm 21{,}302$	96.1 ± 17.9	0.07	-124.8 ± 394.8	1.04 ± 0.97
	best	$22{,}545 \pm 18{,}417$	94.2 ± 17.1	0.25	$+59.9 \pm 558.8$	2.71 ± 2.68
	worst	$24{,}780 \pm 22{,}047$	104.5 ± 16.0	0.18	$+6.8 \pm 489.2$	22.76 ± 41.37

Table 4: Alpha-beta ordering on TRAPPEDCLASSIC ($n = 100$, mean $\pm$ std).

Depth	Ordering	Nodes/game	Nodes/turn	Win rate	Score	Time (s)
2	default	120.7 ± 73.0	26.8 ± 1.3	0.44	-47.0 ± 513.3	0.004 ± 0.003
	best	117.7 ± 72.6	26.9 ± 1.3	0.42	-67.7 ± 510.3	0.007 ± 0.004
	worst	119.2 ± 72.8	26.9 ± 1.3	0.43	-57.4 ± 511.9	0.007 ± 0.004
3	default	50.0 ± 0.0	50.0 ± 0.0	0.00	-501.0 ± 0.0	0.002 ± 0.000
	best	50.0 ± 0.0	50.0 ± 0.0	0.00	-501.0 ± 0.0	0.003 ± 0.000
	worst	52.0 ± 0.0	52.0 ± 0.0	0.00	-501.0 ± 0.0	0.003 ± 0.000
4	default	56.0 ± 0.0	56.0 ± 0.0	0.00	-501.0 ± 0.0	0.002 ± 0.000
	best	56.0 ± 0.0	56.0 ± 0.0	0.00	-501.0 ± 0.0	0.005 ± 0.002
	worst	59.0 ± 0.0	59.0 ± 0.0	0.00	-501.0 ± 0.0	0.004 ± 0.000

3.1.5 Analysis: Two dimensions of the ordering effect

Three effects, grouped into two. Strictly speaking there are *three* effects at play: (A1) *pruning efficiency* – how many nodes are expanded when searching the same sub-tree under different ordering; (A2) *heuristic ranking quality* – how well our state-level `scoreEvaluationFunction` approximates the true backed-up sub-tree values, which determines whether "best" ordering actually presents the best child first; (B) *tie-breaking* – which action is returned when several share the same backed-up minimax value. (A1) and (A2) both show up in nodes/turn and are difficult to disentangle from aggregate statistics, so we group them as (A); (B) is cleanly separable because alpha-beta's *values* are invariant across orderings and any win-rate change with identical values must originate in tie-breaking. The reader should keep in mind that "best ordering does not always beat default" anomalies (e.g. MINIMAXCLASSIC depth 2) belong to (A2), not (A1) – the heuristic mis-ranks children rather than pruning failing.

(A) Computational efficiency. The pruning effect is measured with *decreasing purity* by three metrics:

- **nodes/turn** – per-move search cost, isolating pure pruning efficiency;
- **nodes/game** = nodes/turn $\times$ game length – aggregate cost, confounded by game length (which ordering itself affects via dimension B);
- **time/game** = per-node cost $\times$ nodes/game + sort overhead – the noisiest.

For pure pruning comparisons we rely on **nodes/turn**; nodes/game is supplementary; time is reserved for real-world cost observations and interpreted with the caveat above.

Pure pruning signal (nodes/turn). At depth 4 on MINIMAXCLASSIC, best ordering (1919.7) is the cheapest per turn, but the expected monotone ranking *best* $<$ *default* $<$ *worst* does *not* hold: the actual ranking is best (1919.7) $<$ worst (2006.6) $<$ **default (2122.3)**. Default ordering is the *most* expensive per turn, even more so than worst. We do *not* try to explain this with a post-hoc mechanism: from aggregate statistics alone (A1), (A2), and (B) are entangled – different orderings select different actions, which lead Pacman through different state distributions with different per-state branching and different heuristic rankings, so "per-turn cost" is measured over different state mixtures. Only an experiment that holds the *trajectory* fixed across orderings could isolate (A1) from the rest, and our setup does not. We report the ranking observation and flag it as within the $\pm 10\%$ noise band visible in the depth-2 nodes/turn spread. At depth 3 the ranking matches theory (worst 467.5 $>$ default 447.5 $>$ best 410.7), with a $\sim 14\%$ worst-vs-best gap. At depth 2 the ranking is inverted again: worst (88.8) is actually $\sim 7\%$ *cheaper* per turn than default (95.5). The pure-pruning effect of our three orderings is therefore real but modest, and *not* monotone in the expected direction at all depths – nowhere near the order-of-magnitude blowup that the theoretical $O(b^{d/2})$ vs $O(b^d)$ might suggest.

Nodes/game inflates the headline. The eye-catching "33% more nodes" at depth 4 on MINIMAXCLASSIC (10,046 worst vs 7,539 best) decomposes cleanly as

$$\underbrace{1.045}_{\text{nodes/turn ratio}} \times \underbrace{1.275}_{\text{game-length ratio}} = \underbrace{1.33}_{\text{nodes/game ratio}} .$$

Because this is a multiplicative decomposition, the contributions are not strictly additive; the pruning factor accounts for a ∼4.5% increase and the game-length factor for ∼27.5%, with a ∼1.2% cross-term. The pruning inefficiency per turn is modest; the majority of the aggregate gap comes from worst ordering producing *longer games* (average length 5.01 vs best's 3.93 turns). The same phenomenon appears at depth 2: *per turn* worst is cheaper than default, but *total* is 27% more because worst games last 8.2 turns vs default's 6.0. At all three depths, nodes/game conflates pruning efficiency with a game-length effect that originates in dimension (B).

Time is the noisiest metric, but the 20× gap demands explanation. On smallClassic depth 2, wall-clock time differs by ∼20× (worst 22.76 s vs default 1.04 s) while nodes/game differ by only ∼10%. The discrepancy arises because the reported node count tracks only expansions inside the alpha-beta search, whereas the sorting step (Figure 7) calls `generateSuccessor` and `evaluationFunction` for *every* child at *every* internal node before the search begins – work that is not reflected in the node count. Default ordering skips this step entirely (zero overhead). Best and worst both sort, but worst ordering prunes less effectively, so more internal nodes are visited; each additional internal node triggers a full sort of all its children, creating a cascade of hidden successor-generation calls. The result is that time tracks the *total* work (search + sorting) while nodes/game tracks only the search portion. *For pure pruning analysis, rely on nodes/turn; for practical cost, time is the right metric but reflects sorting overhead on top of search cost.*

Anomalies, now clarified. (1) minimaxClassic depth 2, "best" total (608.9) > "default" (573.9): our state-level heuristic cannot predict backed-up subtree values, so a successor with high immediate score but a bad subtree (e.g., ghost intercept within the horizon) is ranked first, defeating pruning. (2) minimaxClassic depth 3, "worst" total (2520.8) < "default" (2543.4) despite worst nodes/turn (467.5) exceeding default's (447.5): worst ordering simply produces shorter games at this depth (5.39 vs 5.68 turns), overwhelming the per-turn disadvantage. Both anomalies dissolve once per-turn pruning is separated from game-length effects.

(B) Tie-breaking and action selection. Alpha-beta does *not* specify which action is returned when multiple actions share the same minimax value: our implementation returns the first one it examined. Different orderings therefore examine tied actions in different orders and select *different* moves – even though the algorithm computes the same minimax values.

On smallClassic depth 2, win rates differ across orderings (default 0.07, best 0.25, worst 0.18 in our run), but re-runs with different seeds can reverse the ranking entirely (e.g. worst 0.21 > best 0.14 > default 0.11). With $n = 100$ and per-game standard deviations exceeding the mean, none of these pairwise differences are reliably significant (Table 1). The instability itself is the signal: alpha-beta is guaranteed to return the same optimal minimax value regardless of ordering, so any win-rate variation must originate in which tied action is selected first when multiple actions share the same backed-up value. Because tie-breaking is sensitive to the arbitrary action ordering, small changes in the random seed reshuffle the ranking – confirming that the effect is real but noisy rather than directional. At depth 4 on minimaxClassic, the effect appears in game length instead: the implied average length is ≈ 3.66 turns under default ordering but ≈ 5.01 turns under worst ordering (= total nodes ÷ nodes-per-turn), because different tie-breaking leads Pacman down different eat-vs-retreat trajectories.

trappedClassic: the null case. Table 4 shows the opposite extreme. At depth 2, all three orderings are statistically indistinguishable: nodes $\in [117.7, 120.7]$ (spread 3.0 vs std ≈ 73), win rate $\in [0.42, 0.44]$, score $\in [-67.7, -47.0]$ (spread 20.7 vs std ≈ 512). Every difference is well within one standard deviation of the per-game spread. At depths ≥ 3, every game lasts exactly 1 turn with score −501 (Pacman immediately rushes a ghost under worst-case assumptions, cf. Section 3.3), so ordering cannot affect the outcome even in principle; only the search cost varies trivially (50 vs 52 nodes at depth 3). On small trees or trivially decided games, *both* the computational and tie-breaking effects of ordering vanish.

3.1.6 Conclusion

Alpha-beta ordering has two distinct effects that are easy to conflate:

1. *Computational efficiency.* Good ordering reduces the number of nodes expanded, yielding the $O(b^{d/2})$ best-case bound. In our experiments, worst ordering expanded up to 33% more nodes on minimaxClassic and ran 22× slower on smallClassic. However, a simple state-level "best"

heuristic does *not* always beat the arbitrary default order – the heuristic can mis-rank successors whose immediate scores do not reflect their subtree values.

2. *Tie-breaking.* When multiple actions share the same minimax value, the ordering determines which one is returned – silently changing game outcomes. On smallClassic depth 2, win rates shift across orderings and the ranking reverses between runs, confirming that the effect is real but noisy: alpha-beta returns the same optimal value at each root state regardless of ordering, so any win-rate variation must propagate through differing action choices at tied states rather than differing evaluations.

These two effects explain why practical alpha-beta implementations combine (i) a good move-ordering heuristic for efficiency – iterative deepening, killer moves, history heuristic – with (ii) a principled tie-breaking rule for reproducible behavior. Poor ordering does not change the minimax value, but in both dimensions it can severely degrade an agent's practical performance.

3.2 When does minimax's adversarial assumption lead to suboptimal decisions?

Minimax assumes worst-case (adversarial) ghost behavior, while the actual ghosts in Pacman may behave randomly. Discuss when this assumption leads to suboptimal decisions, and explain why a simpler agent (e.g., Reflex) can sometimes outperform minimax.

3.2.1 Theory

`MinimaxAgent` treats every ghost as a *perfectly adversarial* minimiser: at each ghost turn it assumes the ghost will take whichever action hurts Pacman the most. In Pacman's actual `RandomGhost` controller, ghosts choose uniformly at random from their legal actions. The gap between these two models has two consequences.

1. **Over-avoidance.** Minimax can refuse profitable paths because a ghost *could* cut off the route, even though realising the worst case at every step of an s-step cut-off has probability only $(1/|\text{legal actions}|)^s$ under a uniform random ghost. A reflex agent reacts to where ghosts *are*, not where they could be.

2. **Pessimism cascade.** With multiple ghosts, the minimax tree has a min layer per ghost per ply. Each layer compounds the worst-case assumption, pushing the backed-up value below the true expected outcome even on boards where ghosts are benign. Whether deeper search makes this *worse* in practice depends on whether the extra look-ahead also uncovers winning terminal states – a trade-off we examine in Section 3.3.

3.2.2 Experimental Design

We compare `ReflexAgent` against `MinimaxAgent` at depths 1–4 to measure how the adversarial assumption affects play against stochastic opponents.

Agents. `ReflexAgent` uses a hand-crafted evaluation function (food attraction via reciprocal distance, ghost avoidance within distance 2, scared-ghost chasing). `MinimaxAgent` uses `scoreEvaluationFunction` (the raw game score) with worst-case ghost assumptions at depths 1–4.

Layouts. trappedClassic (8×5, 2 ghosts), minimaxClassic (9×5, 3 ghosts), smallClassic (20×7, 2 ghosts), and mediumClassic (20×11, 2 ghosts). The same data is reused for the depth analysis in Section 3.3.

Protocol. $n = 100$ games per (agent, layout) pair against `RandomGhost`. We record score (mean $\pm$ std), win rate, mean game length (Pacman turns), and mean food pellets eaten. We run depths 1–3 on smallClassic and depths 1–2 on mediumClassic; higher depths on these larger boards exceeded our compute budget (mediumClassic d=2 alone averaged $\sim$477 seconds per game).

Confound disclosure and ablation. `ReflexAgent` evaluates with a hand-crafted function while `MinimaxAgent` evaluates with `scoreEvaluationFunction`. This asymmetry is *by design*: the assignment specification requires that `MinimaxAgent` "not hardcode calls to `scoreEvaluationFunction`" but uses `self.evaluationFunction` which defaults to it, and explicitly motivates a Question 5 rewrite with

the hint that *"On larger boards . . . Pacman will be good at not dying, but quite bad at winning. . . . He might even thrash around right next to a dot without eating it."* The pair (minimax search + raw-score leaf) is therefore the baseline the specification asks us to diagnose. To cleanly separate the "adversarial assumption" effect from the "evaluation quality" effect, we run an **ablation experiment** (Table 6) in which `MinimaxAgent` uses a state-only version of the reflex heuristic (`reflexStateEvaluation`: raw score + reciprocal food distance + hard −500 penalty within ghost distance 2) as its leaf evaluator, on minimaxClassic and smallClassic at depths 1–2.

3.2.3 Results

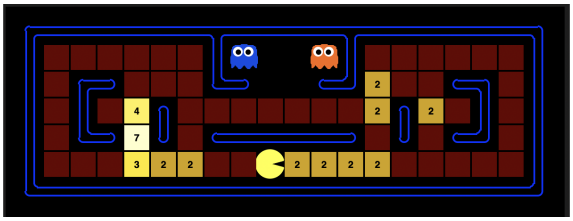
Table 5 summarises the comparison.

Table 5: ReflexAgent vs. MinimaxAgent against random ghosts ($n = 100$, mean ± std).

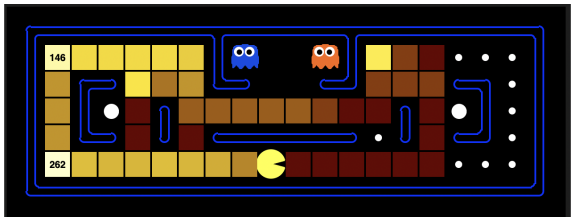
Layout	Agent	Score	Win rate	Game length	Food eaten
trappedClassic	ReflexAgent	-48.0 ± 513.3	0.44	5.6 ± 2.9	1.8 ± 2.0
	MinimaxAgent d=1	$+14.0 \pm 517.0$	**0.50**	6.0 ± 3.0	2.0 ± 2.0
	MinimaxAgent d=2	$+15.0 \pm 517.0$	**0.50**	5.0 ± 3.0	2.0 ± 2.0
	MinimaxAgent d=3	-501.0 ± 0.0	0.00	1.0 ± 0.0	0.0 ± 0.0
	MinimaxAgent d=4	-501.0 ± 0.0	0.00	1.0 ± 0.0	0.0 ± 0.0
minimaxClassic	ReflexAgent	$+180.3 \pm 475.5$	**0.67**	6.4 ± 3.4	1.7 ± 0.5
	MinimaxAgent d=1	$+48.9 \pm 503.7$	0.54	6.5 ± 3.4	1.5 ± 0.5
	MinimaxAgent d=2	-51.7 ± 501.2	0.44	6.1 ± 3.5	1.4 ± 0.5
	MinimaxAgent d=3	-172.4 ± 469.1	0.32	5.6 ± 2.7	1.3 ± 0.5
	MinimaxAgent d=4	$+142.6 \pm 486.7$	0.63	3.7 ± 1.4	1.6 ± 0.5
smallClassic	ReflexAgent	$+1137.2 \pm 666.7$	**0.65**	158.0 ± 118.8	50.5 ± 8.1
	MinimaxAgent d=1	-217.3 ± 304.9	0.05	212.6 ± 164.4	31.9 ± 13.6
	MinimaxAgent d=2	-83.8 ± 346.6	0.10	206.7 ± 217.8	33.1 ± 13.9
	MinimaxAgent d=3	$+212.4 \pm 649.1$	0.26	275.2 ± 201.1	45.0 ± 12.0
mediumClassic	ReflexAgent	$+1526.0 \pm 697.6$	**0.76**	265.4 ± 133.0	90.5 ± 16.1
	MinimaxAgent d=1	-461.9 ± 647.6	0.04	904.5 ± 724.2	62.5 ± 21.0
	MinimaxAgent d=2	-432.5 ± 936.3	0.06	1122.9 ± 947.6	74.8 ± 15.8

Table 6: **Ablation: MinimaxAgent with reflex-style leaf evaluation vs. raw-score leaf evaluation.** "score" denotes `scoreEvaluationFunction` (Table 5 reproduced here for side-by-side comparison, re-run with a fresh seed so minor variation is expected); "reflex-eval" denotes `reflexStateEvaluation` (the ReflexAgent features as a state-only evaluator). All rows against `RandomGhost`, $n = 100$ games, mean ± std. For reference, plain `ReflexAgent` (no search) achieves +180.3/0.67 on minimaxClassic and +1137.2/0.65 on smallClassic (Table 5).

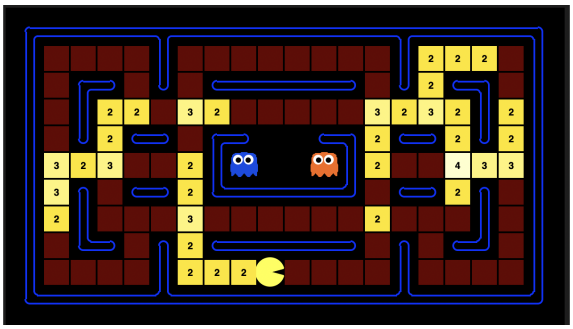
Layout	Configuration	Score	Win rate	Game length	Food eaten
minimaxClassic	Minimax d=1, score-eval	$+69.1 \pm 502.0$	0.56	6.5 ± 3.0	1.6 ± 0.5
	Minimax d=2, score-eval	-102.1 ± 492.1	0.39	6.0 ± 3.2	1.4 ± 0.5
	Minimax d=1, reflex-eval	-233.3 ± 440.5	0.26	5.9 ± 3.3	1.3 ± 0.4
	Minimax d=2, reflex-eval	-51.9 ± 500.1	0.44	6.3 ± 1.9	1.4 ± 0.5
smallClassic	Minimax d=1, score-eval	-252.8 ± 225.0	0.03	173.4 ± 118.9	28.3 ± 10.9
	Minimax d=2, score-eval	-47.7 ± 475.7	0.17	217.8 ± 164.0	36.0 ± 14.2
	Minimax d=1, reflex-eval	$+1222.9 \pm 612.2$	**0.76**	119.3 ± 52.4	51.2 ± 9.4
	Minimax d=2, reflex-eval	$+1457.7 \pm 455.6$	**0.89**	115.2 ± 44.3	53.5 ± 6.3



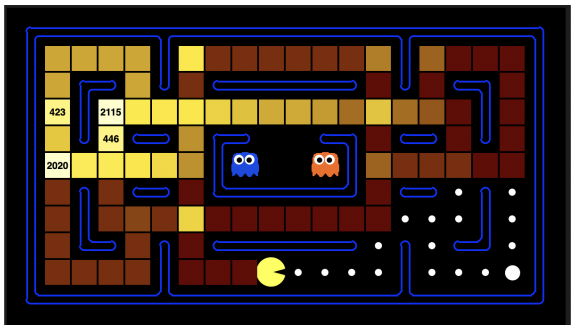
(a) smallClassic · ReflexAgent · win (+1772, 78 turns)



(b) smallClassic · MinimaxAgent d=1 · loss (−702, 652 turns)



(c) mediumClassic · ReflexAgent · win (+2124, 146 turns)



(d) mediumClassic · MinimaxAgent d=2 · loss (−5277, 5987 turns)

Figure 9: Pacman visit-frequency heatmaps (log colour scale): each visited cell is shaded by the number of times Pacman stepped onto it during one representative game, and black numerals label heavily-thrashed cells. `ReflexAgent` sweeps each board evenly (peak visit counts of 7 and 3 respectively). `MinimaxAgent`'s worst-case ghost assumption makes it oscillate over a handful of safe cells while leaving the right half of each board completely unexplored; peak visit counts reach 262 on smallClassic and 2115 on mediumClassic, matching the anomalous 1,122 turn game length reported in Table 5.

The results are striking. On smallClassic, `ReflexAgent` achieves a 65% win rate and mean score of +1137, while the best minimax variant (d=3) manages only 26% and +212. On mediumClassic, the disparity is even larger: 76% vs. 6%. The food-eaten column reveals *why*: on smallClassic, `ReflexAgent` eats 50.5 of 55 pellets on average, while `MinimaxAgent d=1` eats only 31.9 – minimax's pessimism causes it to avoid food corridors where ghosts *might* appear.

Sanity baseline (testClassic). To confirm that `ReflexAgent` is not pathological – i.e., its advantage is specific to random-ghost layouts rather than a general artifact – we ran it on the trivial testClassic layout (Figure 8b; 5×10, 1 ghost, 8 food). It achieves a perfect 1.00 **win rate** with score 562.8 ± 1.9, game length 17.2 ± 1.9 turns, and *all* 8 pellets eaten every game (food eaten $= 8.0 \pm 0.0$). The essentially zero variance confirms that our reflex heuristic is deterministic and correct on simple layouts; its performance on the harder layouts above is therefore a fair benchmark, not a coincidence from a misbehaving baseline.

3.2.4 Analysis

Immediate ghost proximity. `ReflexAgent` applies a hard −500 penalty to any successor state in which a (non-scared) ghost lies at Manhattan distance strictly less than 2, i.e., adjacent to or sharing Pacman's cell. Against random ghosts this closely tracks actual danger: a ghost adjacent to Pacman's next position has a ≥25% chance of stepping onto Pacman next turn, whereas a ghost several steps away poses near-zero immediate risk.

Compounded pessimism. Minimax at depth 2 with 2 ghosts has 2 plies × 2 ghosts = 4 ghost-min layers in total. At each of these 4 layers, minimax assumes the ghost takes whichever action hurts Pacman the most. The combined effect of 4 independent worst-case assumptions drives the backed-up value far below the true expected outcome under random ghost behavior, so minimax consistently underestimates the quality of positions and avoids paths that are actually safe.

Concrete scenario. Consider Pacman at the junction of two corridors, one leading to food and one leading away. A ghost is at distance 4 in the food corridor. `MinimaxAgent` at depth 2 assumes the ghost

moves toward Pacman both times – blocking the corridor – and diverts Pacman. In practice, a random ghost at each turn chooses uniformly from k legal actions. Because `GhostRules.getLegalActions` removes both `STOP` and the reverse of the ghost's current direction (unless at a dead end), k is at most 3 at a four-way intersection, 2 at a T-junction, and often 1 in a straight corridor. In an open area ($k = 3$) the probability of blocking twice is $(1/3)^2 \approx 11\%$; in a narrow corridor ($k = 1$) the ghost has no choice and the blocking probability is 100%.[1] In both cases the *expected* outcome favours going for the food, yet minimax assumes the 100% worst case. `ReflexAgent` ignores this unlikely scenario and takes the food. Over 100 games, the reflex agent collects more food and wins more often.

Game-length evidence. Table 5 shows that `MinimaxAgent d=2` on mediumClassic takes on average 1122.9 ± 947.6 **turns** to end (usually a loss), while `ReflexAgent` takes only 265.4 ± 133.0 **turns** (usually a win). Minimax is so conservative that it thrashes without making progress, accumulating a large living penalty (-1/turn).

Ablation: eval quality vs. adversarial assumption. Table 6 reports the result of swapping `scoreEvaluationFunction` for the state-only reflex evaluator inside `MinimaxAgent`. The outcome is a *two-direction* result that separates the two effects cleanly by layout.

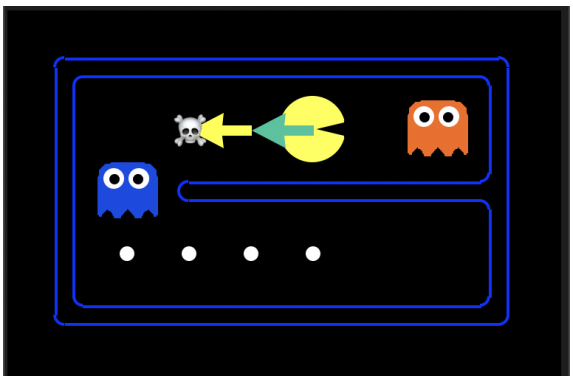
smallClassic*: eval quality was the bottleneck.* Switching the leaf evaluator alone – with *no* change to the search algorithm or opponent model – lifts win rate from 0.03 → 0.76 at depth 1 and 0.17 → 0.89 at depth 2; mean score rises from $-252.8 \rightarrow +1222.9$ and $-47.7 \rightarrow +1457.7$ respectively; mean food eaten climbs from 28.3–36.0 to 51.2–53.5 out of 55 pellets. Notably, `MinimaxAgent d=2 + reflex-eval` (+1457.7/0.89) *exceeds* plain `ReflexAgent` (+1137.2/0.65, Table 5) by a large margin, confirming that on smallClassic the adversarial cascade was essentially benign – the real damage came from a leaf evaluator that could not distinguish "food corridor" from "empty corridor". Game length also collapses from ∼200 turns of thrashing to ∼115 turns of purposeful sweeping, exactly the "He might thrash around right next to a dot without eating it" failure that the assignment specification anticipates.

minimaxClassic*: adversarial cascade was the bottleneck.* The same swap produces the opposite verdict. At depth 1 it *hurts*: win rate drops 0.56 → 0.26, score $+69.1 \rightarrow -233.3$, food 1.6 → 1.3. At depth 2 it helps marginally (0.39 → 0.44, $-102.1 \rightarrow -51.9$). Even the best ablation cell (0.44 at d=2) still trails plain `ReflexAgent` (+180.3/0.67). The mechanism is that the reflex evaluator's hard -500 within ghost-distance-2 penalty, once funneled through a minimax *worst-case* layer, becomes "every plausible food action is -500 because some ghost could move adjacent". The very feature that makes `ReflexAgent` cautious in isolation amplifies cascade when plugged into a min layer – a clean demonstration that on tight-geometry boards the damage is in the search, not the leaf.

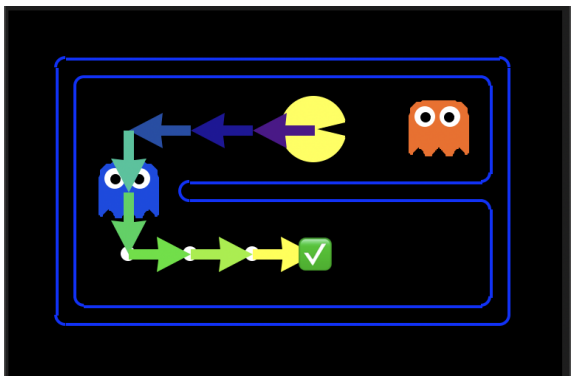
Joint reading. Taken together, the ablation shows that "Reflex beats Minimax" is *not* attributable to one cause. On large open boards (smallClassic), evaluation quality is the dominant term and swapping it alone resolves the gap. On cramped adversarial boards (minimaxClassic, trappedClassic), the adversarial assumption itself is the dominant term and better evaluation cannot save it. This matches the layout-dependent non-monotonicity we analysed in Section 3.3: both the depth effect and the eval effect are modulated by board geometry, and neither knob alone is sufficient. Section 3.2's qualitative prediction – that replacing the min layer with an expectation layer (`ExpectimaxAgent`) would address both failure modes simultaneously – is reinforced by the ablation pattern, though it remains empirically untested.

When does the adversarial assumption help? The adversarial model is not always a liability. On trappedClassic at depths 1–2, minimax actually *outperforms* Reflex: minimax wins 50% vs. Reflex's 44% (Table 5). In tight corridors where ghost proximity is *genuinely* dangerous, minimax's caution pays off – it avoids moves that bring Pacman adjacent to a ghost, while `ReflexAgent`'s greedy food-chasing occasionally walks into a trap. More generally, the adversarial assumption helps when (a) the layout geometry makes ghost encounters likely regardless of ghost policy, and (b) the search depth is shallow enough that pessimism compounding remains mild. Figure 10 contrasts two representative traces on trappedClassic that illustrate this nuance.

[1] This calculation treats k as constant across the s-step cut-off and assumes the turns are independent. `RandomGhost` is memoryless, so the independence assumption is exact, but k varies with ghost position; a rigorous bound would enumerate over all ghost trajectories. The qualitative conclusion—that the blocking probability depends heavily on corridor geometry and is far from the 100% worst case that minimax assumes—is robust to this simplification.



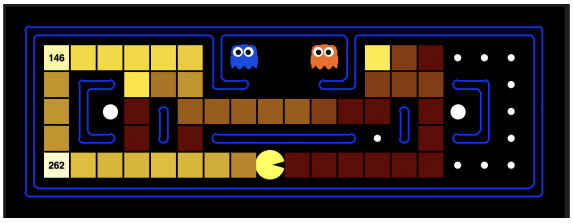
(a) ReflexAgent · loss (−503, 3 turns)



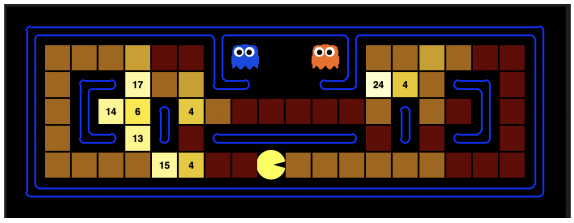
(b) MinimaxAgent d=1 · win (+531, 9 turns)

Figure 10: TRAPPEDCLASSIC: where the adversarial assumption *helps*. Path colour encodes Pacman turn index (purple = first turn, yellow = last). Left: `ReflexAgent`'s greedy food-chase walks straight towards the adjacent ghost and dies in three turns. Right: `MinimaxAgent d=1`'s worst-case ghost assumption diverts Pacman west past the second ghost, then south into the food corridor, sweeping all four pellets in nine turns. In this geometry, over-avoidance is exactly the right policy because encounters really are unavoidable along the naive greedy path.

A caveat: deeper ≠ always worse. Pessimism cascade predicts *monotonically* worsening behavior with depth, but our SMALLCLASSIC results contradict this prediction: d=1 wins 5%, d=2 wins 10%, d=3 wins 26% (Table 5), and food eaten increases from 31.9 to 45.0. On this larger board, the extra look-ahead lets minimax discover food pellets just outside a shallower agent's horizon faster than the additional ghost-min layers can compound their pessimism. The net sign of "deeper ⇒ better or worse" is therefore layout-dependent: cascade dominates on small or geometrically trapping boards (MINIMAXCLASSIC, TRAPPEDCLASSIC) where Section 3.3 analyses the effect in detail, while horizon benefit dominates on larger open boards. Minimax still trails Reflex on SMALLCLASSIC at every tested depth – the point is only that the depth *trend* within minimax runs opposite to the cascade argument. Figure 11 visualises this: the d=1 trace traps Pacman in the left half of the board with peak visit count 262 on a handful of cells, while the d=3 trace sweeps most of the pellets in 181 turns with peak visit count 24.



(a) Minimax d=1 · loss (−702, 652 turns, peak 262 visits)



(b) Minimax d=3 · win (+1669, 181 turns, peak 24 visits)

Figure 11: SMALLCLASSIC: the counter-example to pessimism-cascade. Both panels show `MinimaxAgent` on the same layout; the only difference is search depth. d=1 cannot see past the nearest ghost's worst-case move and loops over the left half of the board; d=3's deeper horizon reveals safe food corridors beyond the immediate worst case, so Pacman commits to the sweep. On larger open boards the horizon benefit of deeper search dominates the additional worst-case compounding that is catastrophic on TRAPPEDCLASSIC and MINIMAXCLASSIC.

3.2.5 Conclusion

Minimax's adversarial assumption is well-suited to two-player zero-sum games with a rational opponent. Against random agents, this assumption becomes a liability: the agent over-avoids ghosts and forfeits food and wins. A reflex agent, which approximates an *expected-value* policy without explicit search, outperforms minimax in these stochastic conditions. Our ablation (Table 6) shows this fail-

ure is *not* monocausal: on smallClassic the leaf evaluator alone accounts for the gap (swapping in reflex-style features pushes Minimax d=2 to 0.89 win rate, beating plain Reflex), while on minimaxClassic even the reflex features cannot save Minimax because the adversarial cascade itself is the bottleneck. The natural next step is `ExpectimaxAgent`, which would replace min layers with expected-value layers and, in principle, address both failure modes simultaneously; a better leaf evaluator alone (Question 5's `betterEvaluationFunction`) only helps on large open boards. We have not implemented `ExpectimaxAgent` in this assignment (it is Question 4, outside the required scope), so this remains a theoretically motivated prediction rather than an empirically verified conclusion.

3.3 Why does increasing search depth not always improve performance?

Why does increasing search depth not always improve performance in Pacman? Provide specific scenarios where a deeper search leads to worse outcomes and explain the underlying reason.

3.3.1 Theory

Three mechanisms cause deeper search to hurt rather than help.

1. **Evaluation function inaccuracy.** Minimax evaluates leaf nodes with a heuristic (`scoreEvaluationFunction`) that returns only the current game score at that state. This score ignores future food locations, remaining capsules, and ghost trajectories. At depth d the tree returns min / max over b^d of these approximations; because the approximation error at each leaf is non-zero, errors propagate upward through the min and max operations.

2. **Pessimistic compounding with random ghosts.** Each additional ghost-min layer adds one more worst-case assumption on top of the previous ones. Recall that one ply (= one depth unit) consists of one Pacman move followed by one move per ghost. With g ghosts at depth d, there are $g \times d$ ghost-min layers in total. Even if each individual ghost is benign (random), the combined minimum across $g \times d$ worst-case assumptions can produce an extremely low value – far below the true expected outcome. This leads to excessively defensive moves. Crucially, this mechanism competes with the usual benefit of deeper search (uncovering winning terminal states within the horizon); whether cascade or horizon benefit wins out is layout-dependent, as the contrasting smallClassic/minimaxClassic behavior in Table 5 shows.

3. **Swift-death preference (an extreme consequence of cascade).** When mechanism (2) has pushed the backed-up values far enough that *every* root action evaluates to approximately the same losing score, a secondary effect determines which action is chosen. Because the game score includes a -1 living penalty per turn on top of the -500 death penalty, actions that lead to *earlier* death receive a *strictly higher* minimax value than actions that delay death (e.g. -501 for ply-1 death vs. -502 for ply-2 death). This is not tie-breaking in the usual sense—the values are genuinely different—but the difference is driven entirely by the living penalty, not by any strategic advantage. The result is that Pacman actively rushes into a ghost on the first turn. Note that this is a consequence of cascade, not an independent mechanism: it disappears whenever mechanism (2) is addressed (e.g. by replacing min layers with expectation layers in `ExpectimaxAgent`). Superficially it resembles an inversion of the classical horizon effect (where a shallow agent stalls to push a loss beyond its horizon), but the root cause is different: the horizon effect stems from insufficient depth, whereas swift-death preference stems from an inaccurate opponent model compounded across sufficient depth. Section 3.3.3 shows this exact collapse on trappedClassic at depths ≥ 3.

Concrete example: all three mechanisms in action. The following minimal scenario shows how mechanisms (1)–(3) interact to flip Pacman's behavior as depth increases. A single ghost in an open corridor cannot produce this flip: the MAX nodes inside each subtree play optimally and retreat from the approaching ghost, so the backed-up value of "go for food" remains positive at every depth. A *pincer* – two ghosts converging from opposite sides – is needed, because Pacman has no retreat direction. This is exactly the geometry of trappedClassic.

Consider the minimal corridor in Figure 12: five cells bounded by walls, Pacman at column 3, G_1 at column 1, G_2 at column 5 (distance 2 from each ghost).

Depth 1. EAST: Pacman moves to column 4; G_2 (MIN) closes from 5 to 4 – collision, −501. WEST: symmetric, collision with G_1, −501. STOP: Pacman stays at 3; ghosts close to columns 2 and 4 (distance 1 each), but no overlap. Leaf = −1 — mechanism (1): the heuristic assigns −1 to a state whose true adversarial value is ≈ − 500. Root = max(−501, −501, −1) = −1, action = STOP. *Pacman survives; the game continues beyond the horizon.*

Depth 2 – the flip. STOP at the root: same ply-1 outcome (Pacman at 3, ghosts at 2 and 4). Now the search sees ply 2. The pincer geometry (mechanism 2) makes every ply-2 action fatal: EAST puts Pacman on G_2 at column 4 (−502); WEST on G_1 at column 2 (−502); STOP lets a ghost (MIN) step onto column 3 (−502). So STOP ↦ −502. EAST/WEST at the root still die at ply 1: −501. Root = max(−501, −501, −502) = −501, action = EAST or WEST. *Pacman charges directly into a ghost.*

Why is −501 preferred over −502? Both end in death (−500), but dying at ply 1 accumulates only one living penalty (−1), while dying at ply 2 accumulates two (−2). These are *not* tied values: −501 > −502 is a strict inequality. The living penalty embedded in the game score makes swift death genuinely higher-valued than delayed death once cascade has made every outcome a loss—exactly the swift-death preference described in mechanism (3), and the suicide-rush observed on TRAPPEDCLASSIC at depth ≥ 3 (where the slightly larger layout delays the flip by one depth level).

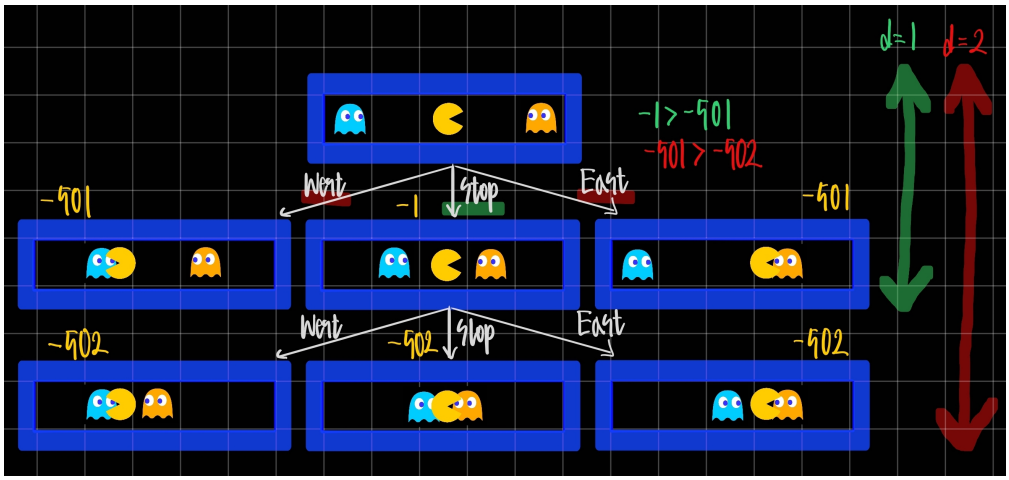


Figure 12: Pincer trap and depth flip. Five-cell corridor: G_1 at col 1, P at col 3, G_2 at col 5. At depth 1 STOP keeps Pacman alive (leaf = −1; mechanism 1: inaccurate heuristic); at depth 2 the pincer geometry (mechanism 2) ensures that *every* ply-2 action ends in collision, so all root actions back up to ≈ −500. The living-penalty difference (mechanism 3) then makes immediate death (−501) strictly higher-valued than delayed death (−502), and Pacman rushes a ghost.

3.3.2 Experimental Design

This question reuses the Experiment B data already reported in Table 5 (Section 3.2): `MinimaxAgent` at depths 1–4 against `RandomGhost`, $n = 100$ games per configuration. Here we zoom in on the two layouts where depth effects are most visible: TRAPPEDCLASSIC (small, adversarial geometry) and MINIMAXCLASSIC (moderate size, non-monotonic behavior).

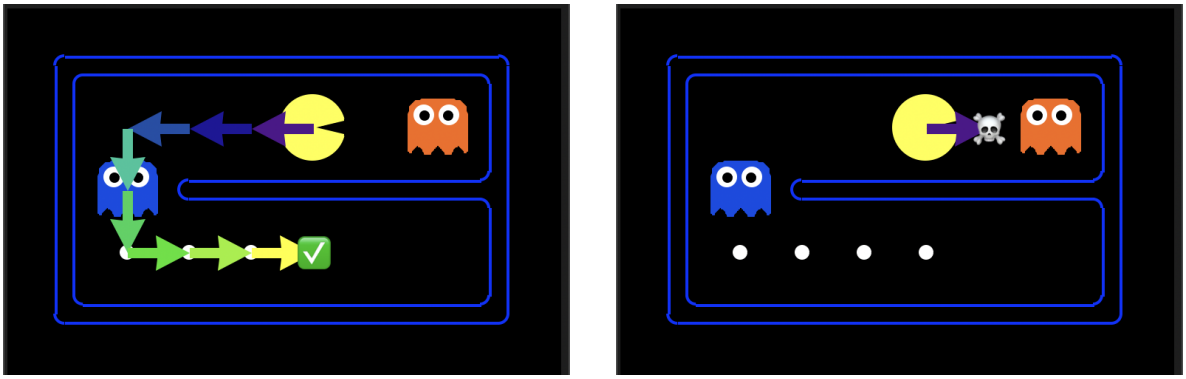
3.3.3 Analysis: trappedClassic

TRAPPEDCLASSIC is the canonical example. Pacman starts in a small maze surrounded by two ghosts. At shallow depth, minimax does not "see" the full trap and makes locally reasonable moves, winning approximately 50% of the time against random ghosts. At depth 3 and 4, minimax sees that every nearby path is eventually cut off – and immediately rushes the nearest ghost to "end the game quickly", because the constant living penalty (−1/turn) makes a swift death preferable to a prolonged losing battle under worst-case ghost assumptions.

The TRAPPEDCLASSIC rows of Table 5 show the catastrophic effect of deeper search. At depths 1 and 2, Pacman wins 50% of the time and eats food. At depths 3 and 4, Pacman *always loses* – the

game lasts exactly **1 turn** (game length = 1.0 ± 0.0, food eaten = 0.0), meaning Pacman immediately charges a ghost on the very first move.

This is not a bug: it is exactly what minimax should do under the worst-case assumption. At depth 3, minimax sees that no sequence of moves survives all possible adversarial ghost responses – so the best achievable minimax value is -501 regardless of which action Pacman takes. Because the living penalty (-1/turn) is embedded in the game score, actions that end the game sooner receive a strictly higher minimax value than actions that delay death, so Pacman picks the swiftest death. The deeper the search, the more thoroughly this doom scenario is confirmed, but the conclusion is wrong because ghosts are actually random. Figure 13 shows the two regimes side-by-side: a nine-turn sweep at depth 1 and a single-turn suicide at depth 3.



(a) Minimax d=1 · win (+531, 9 turns)

(b) Minimax d=3 · loss (−501, 1 turn)

Figure 13: TRAPPEDCLASSIC: premature pessimism at depth 3. The d=1 agent makes locally reasonable moves – detouring west around the first ghost, then south through the food corridor to eat all four pellets. The d=3 agent sees that every move sequence eventually reaches a lost state under worst-case ghost assumptions; among near-identical-valued actions (differing only by living penalty) it picks the one ending the game fastest, rushing into the adjacent east ghost on the very first turn. The arrow trail on the right panel has length one.

3.3.4 Analysis: minimaxClassic (non-monotonic depth)

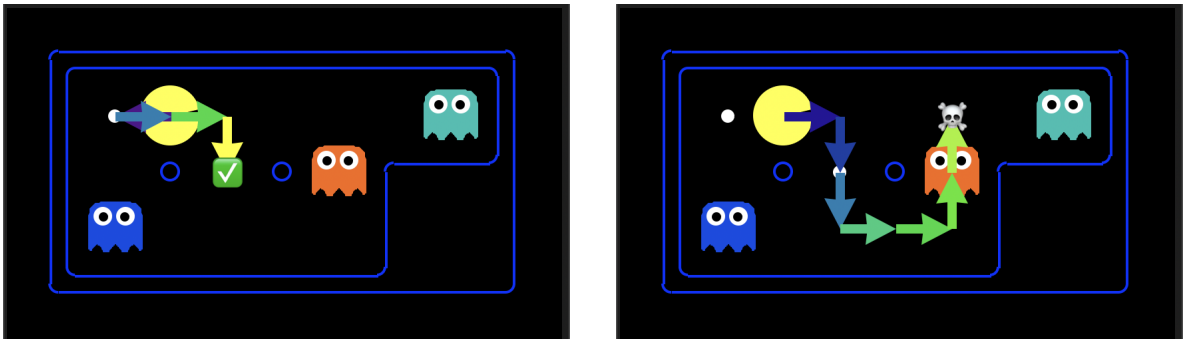
The MINIMAXCLASSIC rows of Table 5 show that performance is *non-monotonic* in depth. Win rate decreases monotonically from d=1 (0.54) to d=3 (0.32), before recovering at d=4 (0.63). Food eaten follows the same pattern: $1.5 \to 1.4 \to 1.3 \to 1.6$.

Why d=4 recovers. MINIMAXCLASSIC has only 2 food pellets and 3 ghosts. Inspecting the layout directly: the first food is one step west of Pacman, while the second food sits two steps away (one east, one south) through the only gap in the horizontal corridor's south wall. Pacman cannot step south directly (the cell below the start is a wall), so the shortest walking tour visiting both pellets is **exactly 4 Pacman moves** (west-eat, back east, east, south-eat). Depth 3 (= 3 Pacman plies × 3 ghosts/ply = 9 ghost-min layers) cannot fit this full eat-food-and-win sequence inside its horizon: at 3 plies the best leaf Pacman can reach still has one food remaining, which minimax evaluates with `scoreEvaluationFunction` – a function that does not credit proximity to the remaining pellet. The backed-up root value at depth 3 is +7 (the reference value from Section 2.2): mildly positive, but essentially flat across the root-level actions because every 3-ply leaf returns a raw score that does not distinguish "retreat" from "advance toward the second pellet". With no gradient to follow, Pacman hesitates and drifts (length = 5.6 ± 2.7 turns, win rate 0.32).

Depth 4 is the exact threshold at which the 4-move winning tour fits inside the search tree. Crucially, this does *not* mean minimax backs up the +516 winning-terminal value at the root: the reference depth-4 root value on MINIMAXCLASSIC is −492 (Section 2.2), because worst-case ghost responses still prevent the tour from completing inside the adversarial tree. What actually changes at depth 4 is the *relative* ordering of actions at the root. Under the worst-case assumption, the food-corridor action has backed-up value −492 (eat one pellet then die), which is strictly better than the remaining alternatives (retreats and detours whose worst-case leaves sit below −492 because they neither gain food nor end

the game any sooner). Minimax therefore *commits* to the food-corridor action even though it believes the game is lost.

Against the actual `RandomGhost`, the adversarial cut-off rarely materialises, so the committed 4-move tour completes and Pacman *realises* +516 empirically. The +516 is a realised score – not a backed-up minimax value – which resolves the apparent tension with the −492 reference. The short observed game length (3.7 ± 1.4 turns) is consistent: d=4 Pacman either executes the 4-move win or dies quickly when the ghost responses happen to match the adversarial prediction. Plugging the empirical win rate into the score accounting is consistent with this picture: assuming losing games average ≈−501 (pure death with no food eaten), solving $0.63 \cdot S_{\text{win}} + 0.37 \cdot (-501) = 142.6$ gives $S_{\text{win}} \approx 520$, close to the 516 expected from the full 4-move tour. The ∼4-point gap likely reflects a small fraction of losing games in which Pacman eats one pellet before dying (yielding ≈−491 rather than −501), which would lower the implied S_{win} toward 516. Figure 14 makes the threshold visible: d=4 executes precisely the 4-move west-east-east-south tour and terminates in the winning state, while d=3 retreats south, wanders for ten turns, and is killed near the east corridor.



(a) Minimax d=4 · win (+516, 4 turns)

(b) Minimax d=3 · loss (−500, 10 turns)

Figure 14: MINIMAXCLASSIC: non-monotonic depth effect. Left: at d=4 the food-corridor action is minimax's strict favourite (backed-up value −492, better than every alternative), so Pacman commits to the 4-move tour; against random ghosts the adversarial cut-off rarely fires and Pacman completes the sweep for a realised +516. Right: at d=3 the 3-ply horizon cannot reach the second pellet and `scoreEvaluationFunction` gives no gradient toward it (backed-up root value +7), so minimax hesitates, retreats south, accumulates living penalty, and dies near the east ghost. Path colour encodes Pacman turn index (purple first → yellow last).

The key insight is that the relationship between depth and performance is *non-monotonic* and layout-dependent: whether a given depth helps or hurts depends on the interaction between the search horizon, the board geometry, and the evaluation function. There is no guarantee of monotone improvement.

3.3.5 Conclusion

Increasing search depth is beneficial only when (a) the evaluation function is accurate enough to reward the extra look-ahead with actionable information, and (b) the opponent model is faithful to actual behavior. In Pacman, neither condition holds reliably: `scoreEvaluationFunction` is a coarse proxy, and worst-case ghost assumptions do not match random ghost behavior. The result is that deeper search can *reduce* win rate by up to 50 percentage points (TRAPPEDCLASSIC d=1 → d=3: from 50% to 0%) and that the relationship between depth and performance is non-monotonic. Standard remedies are therefore twofold: better evaluation functions (e.g., question 5's `betterEvaluationFunction`) that reduce leaf-node error and close the gap in mechanism (1), and opponent models that match reality (e.g., `ExpectimaxAgent`) that replace the min layer with an expectation and thereby close mechanism (2) and its swift-death consequence (3) simultaneously. Our ablation (Table 6) confirms this split empirically: reflex-style leaf evaluation alone resolves SMALLCLASSIC (mechanism (1) was the bottleneck) but leaves MINIMAXCLASSIC and TRAPPEDCLASSIC untouched, because on those layouts the adversarial cascade (2) is the bottleneck and no amount of leaf polishing can substitute for replacing the min layer.

4 AI Usage

4.1 AI name

Claude Code (Claude Opus 4.6, CLI).

4.2 Prompts

- "Implement the reflex evaluation function using reciprocal food distance and ghost proximity penalties."
- "Implement recursive minimax with one ply = Pacman + all ghosts, depth decrementing on agent 0."
- "Extend minimax to alpha-beta pruning—do not prune on equality, propagate bounds through multiple ghost min layers."
- "Write scripts to compare alpha-beta under default/best/worst orderings and to benchmark ReflexAgent vs. MinimaxAgent at depths 1–4."
- "Run ReflexAgent and MinimaxAgent (depths 1–4) against RandomGhost on multiple layouts, recording score, win rate, and game length—I want to see when the adversarial assumption hurts."
- "Design a depth-ablation experiment: sweep depths 1–4 on trappedClassic and mediumClassic, and capture cases where deeper search lowers win rate or score."
- "Run ablation on the evaluation function components—test with and without ghost penalty, food reciprocal, and scared-ghost bonus to isolate each feature's contribution."

4.3 Which parts

Claude wrote code drafts for each agent and the experiment scripts. I formulated the core research questions ("Does action ordering actually matter on small layouts?", "When does deeper search hurt?") and discussed experimental approaches with Claude through iterative dialogue—e.g., I proposed comparing default/best/worst orderings and Claude suggested measuring expanded-state counts as the metric; I asked whether depth always helps and we converged on the depth-ablation design with win-rate and score as joint indicators. After each discussion I reviewed Claude's proposed methodology, refined the variable selection (which layouts, which depth range, number of trials), and directed revisions. For implementation, Claude produced the code while I defined the architectural decisions (e.g., single recursive helper with agent indexing, ply-counting convention for multi-agent minimax), iterated on evaluation-function weights until the autograder threshold was met, and verified the strict-inequality pruning condition (`>` not `>=`) against expected state counts. I ran all experiments and verified every table entry against the raw CSVs. For the report, Claude drafted the LaTeX write-up based on my experimental results and analysis direction; I reviewed each section for correctness, edited the arguments, and verified all claims against the data.

4.4 Verification

- Autograder produces Q1=4/4, Q2=5/5, Q3=5/5.
- Minimax root values on `minimaxClassic` match the spec (9, 8, 7, −492 for depths 1–4).
- All table entries were cross-checked against `experiment_a_results.csv` and `experiment_b_results.csv`.
- Theoretical claims (e.g., best-case alpha-beta complexity $O(b^{d/2})$, trappedClassic rushing behavior) were verified against AIMA §5.3 and the assignment description.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021, §5.3.