# Capture-the-Flag Pacman with Self-Play Tuned Heuristics: A Goal-Commit A$^\star$ Offense, Minimax Defense, and an Anti-Overfitting Verification Protocol

**Doyeol Oh (20266008)** [1 2]

## Abstract

A multi-agent capture-the-flag Pacman team built on classical search: a goal-commit ghost-aware A$^\star$ planner for offense and alpha-beta minimax for defense, both driven by a 42-feature linear evaluation tuned by population-based self-play with held-out zoo anchors. The submitted agent wins $40/40$ official grading games. A held-out audit at $n$=100 games per opponent across ten layouts (700 games, 0 runtime errors) initially exposed three coin-flip results; three targeted fixes (multi-invader defense, sliding-window opponent classifier, tie-break randomization) lifted the mean win rate from $64.9\%$ to $73.4\%$ and the worst-case opponent's win rate from $42\%$ (Mirror) to $50\%$ (Bait, with the worst-case opponent identity shifting). Five further candidate experiments — four algorithmic changes plus one expanded-zoo self-play retune — were prototyped and reverted after their $n$=100 audits showed in-distribution gains paid for by out-of-distribution losses. As an independent external calibration we ran a 792-game sweep against 19 third-party SOTA agents fetched from public GitHub repositories plus our internal zoo, 36 opponents $\times$ 11 layouts $\times$ 2 games with side-swap, yielding a $77.1\%$ aggregate win rate with 0 runtime errors. We argue a hand-inspectable heuristic plus self-play tuning is preferable to deep RL or MCTS variants under the contest's tight per-move budget and multiplicative error-decay penalty.

[1]Ulsan National Institute of Science and Technology (UNIST), Ulsan, Republic of Korea [2]Visiting (exchange) at School of Computing, KAIST, Daejeon, Republic of Korea. Correspondence to: Doyeol Oh <ohdoyoel@kaist.ac.kr>.

*Proceedings of the $38^{th}$ International Conference on Machine Learning*, PMLR 139, 2021. 

## 1. Introduction

Coding Assignment 3 of CS470 is a two-versus-two adversarial capture-the-flag variant of Pacman, adapted from the UC Berkeley CS188 mini-contest (1). Each team controls two agents that act both as ghosts on their home side and as Pacman on the opponent's side; the goal is to ferry food pellets from the enemy's half back to one's own border, while preventing the opponent from doing the same. Games last 1200 agent moves, each agent has at most one second per move, and three timeout warnings forfeit the game. The contest grades on a round-robin tournament against all other student submissions (win/draw/loss = 3/1/0 pt) with a multiplicative decay applied to the final score when the agent's error rate exceeds 10%, and zero points once errors exceed 30%.

These constraints — tight per-move budget, large state space, no opponent introspection, and a heavy error penalty — shaped our design choices. We implemented and tested Q-learning, UCT, vanilla MCTS, and a heuristic-guided MCTS variant (Section 3.5); none of these were shipped. Three observations led us to a deterministic heuristic agent instead. First, to obtain meaningful tree depth within the one-second budget, every MCTS variant we tried had to call the same hand-designed evaluator we already use as its rollout policy or leaf evaluator; once that is done, the marginal benefit over running a deterministic A$^\star$ directly on the same evaluator was small, and the additional simulation noise hurt rather than helped. Second, the workload that actually decides the contest — the student round-robin — is an *unseen distribution*, and a zoo-based held-out check (Section 2.4) is far cheaper as a generalization probe than retraining a parameterized policy each time the zoo changes. Third, a 42-dimensional weight vector is hand-inspectable and pairs naturally with that anti-overfitting verification protocol, whereas a neural policy would have required a separate validation regime that the contest's scoring rule does not reward. The submitted file (`your_best.py`, renamed to `20266008.py` for submission) combines four ideas:

- A **goal-commit offense** that selects a target food, capsule, or border cell, and routes there with ghost-aware A$^\star$.
- A **minimax defense** with alpha-beta pruning that respects a runtime deadline, falling back to a one-step linear evaluation when the deadline approaches.
- An **opponent-style classifier** that observes the first sixty turns of play, labels the opponent as `TURTLE`, `AGGRESSIVE`, `CAPSULE_RUSH`, or `BALANCED`, and adjusts role assignment accordingly.
- A **42-weight evaluation function** optimized by tournament self-play with zoo-anchor games to discourage strategy collapse.

The remainder of this report describes the three baseline progressions that scaffolded the design (Section 2), reports the official grading run alongside extended generalization measurements (Section 3), and discusses the design tradeoff that governed every late-stage decision: how to add capability without overfitting to the small fixed pool of opponents we could test against (Section 4).

## 2. Methods

We describe the four agents implemented for this assignment in order of increasing sophistication: the three required baseline agents `your_baseline1/2/3` (Sections 2.1–2.3), which serve as internal calibration ladders, and the final submitted agent `your_best` renamed to `20266008.py` for submission (Section 2.4). The submitted file `20266008.py` is the v2 version of `your_best.py` together with the small-map specialized weight pack and the Task A/B patches of Section 3.4 (search the source for "`NEW (Task A/B)`" to see the diffs); the legacy `your_best.py` kept in the repository is the pre-fix snapshot. Any one of these four agents alone would already satisfy the rubric's "three agents" requirement; we describe all four for completeness.

### 2.1. Baseline 1: Role-Based Reflex

`your_baseline1.py` implements a single-step reflex agent with team-role specialization. The lower-index agent plays offense: it chases the closest food on the opponent's side using maze distance, penalizes one-cell collisions with non-scared ghosts, and returns home when only two food pellets remain. The higher-index agent plays defense: it stays on its own side, intercepts any visible invader, and otherwise patrols the border. The action evaluator is a linear combination of features $\phi(s, a)$ including `distanceToFood`, `numInvaders`, `onDefense`, and `stop`. This matches the structure of the staff-provided `baseline.py` but adds the offense/defense role split and a lethal-ghost veto, both of which are absent from the original.

**Algorithm 1** Offensive turn for `your_best.py`

**Input:** game state $s$, weights $W$, dead-end depth map $D$
**Output:** action $a$
Compute opponent ghost set $G$ and scared set $S$
Read teammate's committed target from `TEAM_PLANS`
$t \leftarrow \textsc{PickTarget}(s, G, S, W, D)$
$\pi \leftarrow \textsc{GhostAwareAStar}(s, t, G, W)$
**if** $\pi$ unreachable **then**
  retry with the next-best target
**end if**
$a \leftarrow$ first action along $\pi$
Persist $t$ to `TEAM_PLANS`

### 2.2. Baseline 2: Return-Home and Border Awareness

`your_baseline2.py` extends Baseline 1 with a precomputed *border list* (the set of empty cells on the centerline that are not walls) and an explicit return-home subroutine. The offense agent now decides between *chase* and *return* modes based on the food it is carrying: once it has eaten three or more pellets, it routes to the nearest border cell instead of pursuing more food. The defender uses the same border list to patrol pinch points rather than walking along its own edge. This single change yields the bulk of the improvement over Baseline 1 because uncollected food is wasted: an agent killed while carrying ten pellets explodes the food back into the maze and the entire trip is undone.

### 2.3. Baseline 3: Dynamic Carry Threshold and Capsule Attraction

`your_baseline3.py` adds two features. First, the carry threshold becomes *dynamic*: it is computed as $\max(2,\ \text{food\_left}/k)$, reducing as the supply runs out so that the agent does not stall on the enemy side waiting for a target it cannot reach. Second, the offense agent gains a *capsule attraction*: when ghosts are visible, the cost function rewards proximity to power capsules, which both opens an attack window during the scared timer and removes the cheapest weapon the opponent has against a heavy-carrying Pacman. The defender is also upgraded to prioritize the highest-value invader (the one carrying the most food) when multiple invaders are visible.

### 2.4. Best Agent: Goal-Commit A$^\star$ + Minimax Defense

`your_best.py` replaces the one-step reflex with a *plan-then-execute* architecture; Algorithm 1 gives the high-level offensive loop.

**Goal-commit target selection.** The function PICKTARGET scores every food pellet, every capsule, and (when the agent must return) every border cell using $\sum_i w_i \, \phi_i$. The complete feature inventory of $\phi$ — the same vector reused by the A$^\star$ cost, the defensive evaluator, and the minimax leaf — is enumerated in Table 1; representative target-selection components are distance to the target, ghost proximity, dead-end depth (precomputed by an iterative leaf-peeling on the wall graph), a team-coordination penalty (so both attackers do not race to the same pellet), and a capsule bonus when ghosts are visible. Targets in dead-end corridors are penalized unless a round-trip safety margin against the closest ghost can be proven by a simple distance-budget calculation.

**Ghost-aware A$^\star$.** Once a target is chosen, the agent computes a path with A$^\star$ on the four-connected wall graph. The cost function is uniform plus a ghost-penalty term that grows sharply when a candidate cell brings the agent within one or two maze-distance cells of a non-scared ghost. The search has a $0.2$ s per-call wall-clock cap, additionally clamped to the global $0.85$ s deadline minus a $0.05$ s safety margin; if the cap is reached, the agent falls back to a one-step greedy evaluator that uses the same weight vector but operates on `generateSuccessor` states directly.

**Minimax defense.** When in defense and an invader is visible, the agent runs an alpha-beta minimax search rooted at the current state, where the agent maximizes and the chosen invader minimizes. The search depth is one entry of the same 42-component vector — `h_mm_depth` is stored as a real-valued weight so the self-play optimizer can tune it jointly with the rest, but it is cast to an integer (via `int()`) inside the search call. The search respects the same $0.85$ s deadline as the rest of the move; the leaf evaluation emphasizes invader-distance, capsule protection, and ghost-form bonus. *Honest disclosure.* Self-play's continuous estimate landed at `h_mm_depth` $\approx 0.149$ (`selfplay_v3_weights.json`); we ship a hand-rounded $0.10$ for the default-layout weight set and $0.19$ for the small-map variant. All three values cast to integer depth $0$, so the minimax tree degenerates to a one-ply evaluator under the same leaf function. Self-play converged toward $0$ because deeper search exhausts the per-move time budget on large layouts and gains nothing once both players use the same hand-tuned evaluator at the leaves. We retain the alpha-beta scaffolding so the depth knob can be tuned upward if the time budget ever changes.

**Opponent-style classifier.** Over the first sixty agent turns we accumulate three statistics: the fraction of opponent turns spent as a Pacman (invasion rate), the fraction of turns within five cells of one of our capsules (capsule approach rate), and the raw count of invader turns. We then label the opponent: `TURTLE` (very low invasion rate), `CAPSULE_RUSH` (high capsule approach), `AGGRESSIVE` (high invasion rate), or `BALANCED` otherwise. Against a `TURTLE` opponent we drop the second-agent's defensive role entirely and double up on offense.

*Table 1.* Composition of the 42-component weight vector `_WEIGHTS`. Names use the prefix convention `o_*` (offense step evaluator), `a_*` (A$^\star$ path cost), `t_*` (target selection), `d_*` (defense evaluator), `h_*` (structural threshold), `f_*` (higher-order feature). Every weight multiplies a single hand-defined scalar feature; this is what we mean by "hand-inspectable" in the introduction.

| # | Group — features |
|---|---|
| 10 | **Offense step** (`o_*`): food-left penalty, return-home distance, nearest-food distance, capsule distance when ghosts visible, ghost-collision while carrying / while empty, dead-end avoidance, scared-ghost chase, STOP / REVERSE penalties. |
| 4 | **Ghost-aware A$^\star$** (`a_*`): cost for sharing a cell with a ghost, for being within 1 cell, within 2 cells; and a per-cell cost for dead-end depth along the path. |
| 5 | **Target selection** (`t_*`): distance to candidate, round-trip-safe bonus, team-collision penalty, dead-end-corridor penalty, capsule-when-ghosts-visible bonus. |
| 6 | **Defense step** (`d_*`): own-Pacman penalty, invader count, invader distance, border-patrol weight, capsule-guard weight, STOP penalty. |
| 6 | **Structural** (`h_*`): carry-threshold divisor, ghost-return radius, defender-opening window, defend-on-lead margin, capsule-threat radius, minimax depth. |
| 11 | **Higher-order** (`f_*`): predicted-enemy-target weight, carry adjustment when winning / losing, endgame-time threshold, $1/d_g^2$ ghost repulsion, ghost-within-1 penalty, score–time urgency, food-cluster bonus, enemy-already-returned urgency, $1/d_f$ food pull, scared-dance defense. |
| **42** | **Total** |

**Self-play weight tuning.** The 42-dimensional weight vector $W$ is tuned offline by a population-based self-play loop (`train_selfplay.py`). Each generation, candidate weights play a round-robin tournament against each other plus a fixed set of five zoo anchors. Anchors prevent strategy collapse: without them the population converges to a strategy that beats only its own siblings (19). The fitness function is the tournament-style 3-1-0 score normalized by the number of games. The final submission uses the weights from generation "final" of the v3 run (`selfplay_v3_weights.json`, fitness $3.64$). Table 2 reports the loop's hyperparameters for reproducibility.

**Anti-overfitting verification.** Because the optimizer can only see the small fixed pool of zoo anchors, the resulting weights are always at risk of zoo-overfitting. To detect this we wrote three additional opponents (`zoo_holdout1/2/3`) whose strategies were intentionally distinct from the anchors — a scared-window exploiter, a never-return food-bomber, and a score-driven role switcher

*Table 2.* Self-play training hyperparameters used to produce the shipped weight vector `selfplay_v3_weights.json`.

| Hyperparameter | Value |
|---|---|
| Population size | 12 |
| Generations | 20 |
| Elite fraction | 0.25 |
| Self-play matches/generation | 3 random opponents per candidate |
| Anchor matches/generation | 5 zoos × 1 game × 12 candidates |
| Mutation $\sigma$ | $\max(0.2\,\lvert\mu\rvert,\ 0.5)$ Gaussian per coord |
| Anchor weight in fitness | 0.30 |
| Layouts | `defaultCapture`, `RANDOM1`, `RANDOM7` |
| Zoo anchors | `zoo_capsule`, `zoo_bait`, `zoo_astar`, `zoo_reflex`, `baseline` |
| Held-out (forbidden) | `zoo_holdout1/2/3` |
| Random seed | 42 |

— and forbade their use during training. During development the submitted weight vector was gated on a small-$n$ default-layout sweep against these holdouts. For this report we first re-evaluated the agent at $n$=50 games per opponent across ten layouts (all twelve standard maps in `layouts/` except `testCapture`, which has no food, and `bloxCapture`, which produces systematic ties that obscure agent comparison; column v1 of Table 5), and additionally introduced three new adversarial styles (a near-clone of our own architecture, a coordinated double-defense, and a score-defender) to probe failure modes that the original holdouts may have masked. After applying the targeted fixes of Section 3.4, the shipped agent (column v2) was re-audited at $n$=100 for tighter confidence intervals. Section 3 reports both rounds together with 95% Wilson intervals.

## 3. Results

### 3.1. Official Grading Benchmark

We ran the official grading command `python capture.py -r 20266008 -n 10`. This plays ten games against each of `your_baseline1/2/3` and the staff `baseline`, and writes `output.csv`. Table 3 reports the contents of `output.csv`; Figure 1 reproduces the raw spreadsheet for reference.

The agent finishes 40 wins out of 40 games with zero recorded errors, clearing the upper-bracket condition (51–100% win rate against `baseline.py`) and securing the full 40-point base credit.

### 3.2. Layout Generalization

To check that the agent does not depend on a single layout we ran a twelve-layout sweep (all maps in `layouts/`)

*Table 3.* Official grading benchmark: `output.csv` produced by `python capture.py -r 20266008 -n 10`, ten games per opponent. The submitted agent wins every matchup with a +1.0 average winning rate and a +9.6 mean score across the four opponents, and registers zero runtime errors. `Num_Win` reaches the maximum value of 4, and `Avg_Winning_Rate` the maximum value of +1.0.

| OPPONENT | AVG. WINNING RATE | AVG. SCORE |
|---|---|---|
| YOUR_BASELINE1 | +1.0 | +8.6 |
| YOUR_BASELINE2 | +1.0 | +11.2 |
| YOUR_BASELINE3 | +1.0 | +8.2 |
| BASELINE | +1.0 | +10.4 |
| NUM_WIN | 4.0 | — |
| AVG_WINNING_RATE | +1.0 | — |
| AVG_SCORE | — | +9.6 |

| | your_best(red) |
|---|---|
| | <Average Winning Rate> |
| your_base1 | 1 |
| your_base2 | 1 |
| your_base3 | 1 |
| baseline | 1 |
| Num_Win | 4 |
| Avg_Winning_Rate | 1 |
| | <Average Scores> |
| your_base1 | 8.6 |
| your_base2 | 11.2 |
| your_base3 | 8.2 |
| baesline | 10.4 |
| Avg_Score | 9.6 |

*Figure 1.* Raw `output.csv` spreadsheet, included for reference. Numeric content is identical to Table 3. The label "`baesline`" in the raw CSV is a staff-side typo in the grading script; we render it as `baseline` elsewhere.

with side-swap to remove starting-color bias, four games per layout per side; this gives forty-eight games per opponent. We also sampled eight procedurally generated random mazes via `mazeGenerator`. Table 4 summarizes the per-opponent aggregate; per-layout numbers are dominated by the four ties on `testCapture`, a 12×6 micro-map on which neither team has enough food to score, so we exclude it from later strategy-generalization sweeps.

*Table 4.* Layout generalization: `your_best.py` versus the staff baseline across the twelve standard layouts (forty-eight games) and across eight procedurally generated random mazes (twenty-four games). A clear absence of layout overfitting is observed: the agent performs *better* on unseen random mazes than on the seen layouts.

| LAYOUT POOL | WIN % | AVG. SCORE | ERR. % |
|---|---|---|---|
| 12 STANDARD LAYOUTS | 91.7 | +25.7 | 0.0 |
| 8 RANDOM-SEED MAZES | 95.8 | +20.3 | 0.0 |

### 3.3. Strategy Generalization (Held-out Opponents)

The most informative evaluation is against opponents that were not seen during weight tuning. Table 5 reports the deeper audit (v1 at $n$=50, v2 re-audit at $n$=100) across ten layouts with side-swap, including three additional adversarial styles introduced for this report. Win rates carry 95% Wilson confidence intervals (20) so the reader can distinguish genuine margin from sample-size noise.

**Reading the table.** The v1 picture was honest rather than flattering: the agent crushed *Holdout 2* and the new *Swarm* (both at 90%), but was at coin-flip parity against *Mirror* (42%), *Holdout 3* (46%), and the split-attack *Bait* (48%). The original development gate during implementation was an $n$=4 default-layout sweep against the three holdouts (which always returned 100%); the wider $n$=50 ten-layout v1 audit reported in column v1 revealed that the development gate had been over-optimistic. Three targeted fixes were applied (Section 3.4), giving column v2: *Holdout 3* climbed from 46% to 66% (+20 pp; the sliding-window classifier now tracks role-switching opponents), *Mirror* climbed from 42% to 65% (+23 pp; tie-break randomization breaks the deterministic stalemate), *Holdout 1* went from 72% to 77%, and *Staller* went from 66% to 74%. The remaining concern is *Bait* at 50% (CI $[40, 60]$), where the first-agent multi-invader defense narrowed the gap by only +2 pp at this larger $n$; coordinated split-attacks remain the agent's residual structural weakness. Mean win rate across the seven held-out opponents is 73.4% at v2 (vs. 64.9% at v1), with zero runtime errors across 700 games.

*Table 5.* Held-out opponent audit. Column v1 is the $n$=50 ten-layout snapshot of the agent before the targeted robustness fixes; column v2 is the shipped agent (after three fixes: multi-invader defense for split-attacks, sliding-window opponent classifier for role-switching opponents, and tie-break randomization for symmetric-architecture opponents) re-evaluated at $n$=100 games per opponent (10 layouts × 10 games per layout, alternating which side plays our agent for side-swap) with 95% Wilson confidence intervals. AVG. SC. and the CIs are for v2. Holdout 1 is a capsule-window exploiter; Holdout 2 is a never-return food-bomber; Holdout 3 dynamically switches both agents between offense and defense based on score. Of the four rows in the lower group, MIRROR, SWARM, and STALLER are post-hoc stress tests added during this audit (MIRROR is a near-clone of our own architecture with untuned weights, SWARM is a coordinated double-defense pursuit, and STALLER is a score-defender that holds its lead); BAIT is a coordinated split-attack that was a training anchor for the v3 weight tuner but is reported here because it remains the agent's residual structural weakness despite training exposure. Errors are zero across all 700 v2-audit games, demonstrating that the deadline-aware fallback path keeps the error-rate decay term inactive.

| OPPONENT | v1 % | v2 % | 95% CI (v2) | AVG. SC. | ERR. |
|---|---|---|---|---|---|
| HOLDOUT 1 (CAP.) | 72 | 77 | [68, 84] | +9.9 | 0 |
| HOLDOUT 2 (BOMB) | 90 | 89 | [81, 94] | +22.5 | 0 |
| HOLDOUT 3 (SW.) | 46 | 66 | [56, 75] | +8.2 | 0 |
| MIRROR (NEW) | 42 | 65 | [55, 74] | +0.7 | 0 |
| SWARM (NEW) | 90 | 93 | [86, 97] | +27.4 | 0 |
| STALLER (NEW) | 66 | 74 | [65, 82] | +9.2 | 0 |
| BAIT (SPLIT-ATTACK) | 48 | 50 | [40, 60] | −5.8 | 0 |

### 3.4. Targeted Robustness Fixes

Three small, narrowly-scoped fixes between v1 and v2 produced the gains in Table 5; they are listed here so the reader can map each gain to a concrete change.

- **Multi-invader defense (*Bait* $+2$ pp).** The role selector now lets the first agent join defense when there are two invaders simultaneously, the agent is currently a ghost with no carry, and the agent is meaningfully closer to one invader than the partner is. Without this rule, a single defender cannot cover the two halves of a coordinated split-attack. The effect is small at $n$=100: *Bait* sits structurally near $50\%$ on this architecture and the multi-invader rule moves the floor by only two points; coordinated split-attacks remain the residual weakness.

- **Sliding-window opponent classifier (*Holdout 3* $+20$ pp).** The opponent style classifier was a one-shot label committed at turn 60. We now keep an 80-turn ring buffer and re-classify every 30 turns. This is what catches the *Holdout 3* role-switching strategy — the static label could not follow it; the rolling window does.

- **Tie-break randomization (*Mirror* $+23$ pp).** The target picker and greedy fallback now randomize among near-best candidates (within a small score margin). The *Mirror* match was an underdetermined symmetric game where deterministic tie-breaking trapped both agents in cycles; small randomness breaks the symmetry without changing the underlying weights, lifting the win rate from coin-flip to a measurable margin.

### 3.5. Internal Variant Comparison

We also evaluated several earlier development variants that explored different architectures (Q-learning, UCT, MCTS, potential fields, and an alternative tuning `your_best_rlv2`). Table 6 summarizes the pairwise four-game matchups. Four of the five variants (`qlearn`, `uct`, `mcts`, `potential`) hover around the 50% mark against the submitted agent on the default layout, which at $n$=4 is statistically indistinguishable from a coin flip and supports our introduction's claim that none of the architectural alternatives delivered a clear margin. The fifth, `rlv2`, is the only variant that wins cleanly on the default layout (4–0); however, it falls to 75% (3–4) on the held-out opponent `zoo_holdout1`. We treat this as a textbook demonstration of overfitting — gains on a narrow distribution at the cost of out-of-distribution robustness — and discuss it in Section 4.

*Table 6.* Internal variant comparison on the default layout, four games each. With $n$=4 per matchup the upper four variants are statistically indistinguishable from chance against the submitted agent; `rlv2` is the only variant with a clean default-layout margin, and it is also the only one that collapses on a held-out opponent. This pattern — in-distribution gain bought with out-of-distribution loss — is what motivates shipping the static heuristic agent.

| VARIANT (`YOUR_BEST_*`) | VS SUBMITTED | VS HOLDOUT 1 |
|---|---|---|
| `QLEARN` (Q-LEARNING) | $\sim 50\%$ | — |
| `UCT` (UCT) | $\sim 50\%$ | — |
| `MCTS` (VANILLA MCTS) | $\sim 50\%$ | — |
| `POTENTIAL` (POTENTIAL FLD.) | $\sim 50\%$ | — |
| `RLV2` (ALT. TUNING) | $100\%$ (4/4) | $75\%$ (3/4) |

### 3.6. Negative Results: Experiments That Did Not Ship

After v2 was settled, five additional candidate experiments were prototyped and audited at the same $n$=100 scale. None produced a clear net win over v2 and all were reverted; we document them here so future work can avoid the same dead ends.

**(i) MIRROR-mode aggressive override.** When the classifier detected a long score stalemate, we forced the carry-threshold to 1 and boosted capsule priority. The change lifted *Mirror* by up to $+18$ pp, but cost a comparable amount on *Holdout 3* and *Bait*, leaving the sum-of-win-rates within noise.

**(ii) Max-flow bottleneck camping.** A Ford–Fulkerson chokepoint search positioned the defender on the cell whose removal disconnects the most food from the border (the signature move of the Pacamon team (3)). The $n$=100 audit gave $+6$ pp on *Holdout 1* and $+3$ pp on *Mirror* but $-8$ pp on *Bait*, since split-attacks simply go around the bottleneck. A defensive fall-back flag (disable bottleneck once two simultaneous invaders are seen) recovered *Bait* by $+4$ pp at the cost of $-9$ pp on *Holdout 1* — a different trade, same net.

**(iii) Blind-spot avoidance.** Appending each ghost's predicted next cell to the A$^\star$ ghost-cost set (also from (3)) over-penalized symmetric-equilibrium moves and lost $\sim 25$ pp on *Mirror* in an $n$=20 smoke test, ruling the change out before a full audit.

**(iv) Layered endgame defense.** Tightening the lead-protection thresholds (SCORE $\geq 5 \wedge$ TL $< 600$, etc.) lost $-15$ pp on *Mirror* and $-3.4$ pp on the mean at $n$=100, because going defensive earlier surrendered offense tempo.

**(v) Expanded-zoo self-play retune.** We re-ran the population-based self-play optimizer with all 13 zoos folded

into the training pool (including the post-hoc *Mirror*, *Swarm*, *Staller* and Holdouts 1–3), reasoning that v2's five-anchor training set might be the bottleneck. Self-play fitness rose from 3.64 to 5.56 in two generations and plateaued for six more. To test generalization we picked four zoos (`zoo_chaser`, `zoo_camper`, `zoo_safe_return`, `zoo_minimax_off`) that had been used for neither v2's nor v5's training and ran both v2 and v5 against them at $n$=100 each. Both achieved an identical mean of 76.5% on this unseen set — no generalization gain. On the in-distribution 7-opponent audit, *Mirror* dropped by $-13$ pp and *Holdout 3* by $-10$ pp (clear regression on opponents v5 had trained on). The retuner's fitness function over-weighted training-pool wins relative to actual head-to-head margin, so the "higher-fitness" weights specialized in ways that did not transfer.

**Common pattern.** The shape across all five negative results is the same as the `rlv2` result of Section 3.5: in-distribution gains trade against out-of-distribution generality, and the cleanest defensible weight-and-rule set is the one that resisted the temptation to specialize.

### 3.7. External SOTA Calibration: 792-Game Sweep

To calibrate where the submitted agent sits relative to the broader international literature, we fetched all open-source Pacman capture-the-flag agents we could find on GitHub that drop into our framework without external dependencies. After removing those that failed to import (TensorFlow-trained policies, Python-2 syntax, custom `baselineTeam.py` dependencies, deprecated framework versions), 19 third-party agents remained. We combined them with our 17 internal zoo opponents (the five training anchors, the three original holdouts, the four post-hoc adversarial styles of Section 3.3, and five additional internal probes) and ran a sweep of $36 \times 11 \times 2 = 792$ games with side-swap, across all eleven non-degenerate layouts in the `layouts/` directory (we re-include `bloxCapture` here because under a heterogeneous SOTA pool its tie rate is itself a calibration signal rather than noise).

**Aggregate.** 611 wins, 69 draws, 112 losses — 77.1% win rate, 14.1% loss rate, and *zero* runtime errors across the full 792 games. The mean per-game score margin is $+22.1$.

**Per-opponent tiers.** Table 7 groups the opponents by win rate against the submitted agent. The "tough" tier ($\leq 50\%$ win rate at $n$=22 games each) contains five opponents: three third-party (`boyanzhang`, `infinityglow`, `iangalvez`) and two internal post-hoc adversaries (`bait`, `mirror`). The strongest is `boyanzhang` (a COMP90054 student team) which holds us to 27% wins, followed by `infinityglow` (Team Alpha, COMP90054 ranked 8/88 (5)) at 45%. Five further opponents sit in the 51%–70% band and twenty-six opponents at $\geq 71\%$.

*Table 7.* Per-opponent calibration, $n$=22 games per opponent (11 layouts $\times$ 2 games with side-swap). Opponents binned by win rate against the submitted agent. The five opponents in the top group — three third-party (`boyanzhang`, `infinityglow`, `iangalvez`) and two internal post-hoc adversaries (`bait`, `mirror`) — are where the submitted agent's remaining gap to the absolute frontier lives.

| Tier (W %) | # | Notable members |
|---|---|---|
| $\leq 50\%$ | 5 | boyanzhang (27%), infinityglow (45%), bait, mirror, iangalvez |
| 51–70% | 5 | holdout3, camper, astar, sren1618, tianqipeng |
| 71–80% | 7 | 214607135, kkkkkaran, holdout1, chunyinlai, aryanluthra, capsule, chaser |
| 81–95% | 14 | dacphuc1993, mitchellgust, holdout2, swarm, devminjin, aryansamuel, yangxvlin, kurkkk, aggressive, turtle, minimax_off, staller, safe_return, reflex |
| 100% | 5 | random, Ritesh1909, dedgit, nomaanakhan, reesewills |
| **Total** | **36** | 611 W · 69 D · 112 L (77.1%, 0 errors) |

**Per-layout breakdown.** Win rates by layout: `defaultCapture` 90%, `distantCapture` 85%, `jumboCapture` 83%, `officeCapture` 82%, `alleyCapture` 81%, `mediumCapture` 81%, `strategicCapture` 76%, `fastCapture` 74%, `crowdedCapture` 71%, `bloxCapture` 65%, `tinyCapture` 61%. The grading-default layout `defaultCapture` is the strongest; the small-and-narrow layouts (`tinyCapture`, `bloxCapture`) are the weakest, partly because they produce many no-score draws regardless of opponent.

**What this calibration says.** 77.1% over 792 games against a heterogeneous pool that includes nine documented international course-tournament submissions places the agent in the international top-tier band rather than the absolute top. The five hardest opponents (`boyanzhang`, `infinityglow`, `bait`, `mirror`, `iangalvez`) cost the agent essentially all of its distribution losses; against the other thirty-one opponents the agent wins $\geq 80\%$ of games. If the actual KAIST student field contains agents at the level of `boyanzhang` or `infinityglow`, the contest will be decided in matchups against those one-or-two students; against everyone else the $+22$-point average margin and 0% error rate provide a comfortable floor.

## 4. Conclusion and Free Discussion

**Summary.** The shipped agent wins all 40 official grading games, generalizes to unseen layouts (95.8%), passes seven held-out adversarial strategies at a 73.4% mean win rate at $n$=100 (Sections 3.3–3.4), and reaches 77.1% on the 792-game external SOTA calibration sweep (Section 3.7), all with zero runtime errors. Five candidate post-v2 experiments were prototyped and reverted after $n$=100 audits showed in-distribution gains paid for by out-of-distribution losses (Section 3.6).

**Self-imposed Question 1: How can a 42-parameter linear heuristic, tuned on a small zoo, possibly generalize?** The mechanism, in our view, is the strong separation between the *decision-making structure* and the *weights*. The structure — A$^\star$ over a maze graph, alpha-beta minimax, dead-end depth peeling, role classification — encodes universal facts about the domain that hold for any opponent. The weights only adjust the *relative* importance of these structural signals. As long as the zoo is diverse enough to keep all weights in a non-degenerate range, the resulting policy continues to behave reasonably against any opponent that obeys the same structural facts. This is consistent with the observation that the food-bomber and double-defense holdouts are won at $\geq 90\%$ even though neither was used to compute a single gradient step. It is also consistent with the observation that an alternative tuning (`rlv2`) which gains margin against in-distribution opponents loses generalization on a held-out one: the additional in-distribution margin came at the cost of a structural feature, not just a numeric adjustment. The honest caveat is that this transfer is asymmetric: opponents whose strategy contradicts a structural assumption (*Mirror*, *Holdout 3*) sit near the 50% line. The structure-vs-weights split explains generalization within the structural envelope and bounds it outside.

**Self-imposed Question 2: Why not deep RL or full MCTS?** We did write Q-learning, UCT, MCTS, and self-play RL variants (`your_best_qlearn.py`, `your_best_uct.py`, `your_best_mcts.py`, etc.); none reached the submitted agent's level. The deeper reason — below the obvious one-second budget — is that even when we plugged our hand-designed evaluator *into* MCTS as a rollout policy or leaf evaluator (the natural fix for "rollouts dominated by random play"), the marginal benefit over running deterministic A$^\star$ directly on the same evaluator was small, and the additional simulation noise actually hurt; Table 6 captures this as the four "$\sim 50\%$" rows. A second, more pragmatic reason is error robustness: learned policies are brittle to error spikes — a single uncaught exception during a rollout produces a fatal time warning, and the contest decays our entire score multiplicatively beyond a 10% error rate. The heuristic agent has no neural network to crash, and a single try/except in `chooseAction` that defaults to a random legal action keeps the error rate at zero in every benchmark we ran.

**Self-imposed Question 3: What single design choice mattered most?** The held-out opponent set. Without it, an evaluator can only tell us whether a candidate beats the zoo, which it must — the optimizer was told to maximize that. The held-out opponents are the only defense against the optimizer's natural tendency to overfit. Late in development we tested a candidate that capped the carry threshold at five food pellets to reduce the variance of large-cargo runs. It maintained 100% on the baselines and the original three holdouts. A six-game mirror match against the unmodified agent on `jumboCapture` (alternating sides), however, gave the unmodified agent the per-game scores $+16, +64, +24, +50, +25, +0$ (5 wins, 1 tie, mean margin $+29.8$). We rejected the change. Without the mirror-on-large-layout test we would have shipped a regression that the baselines and holdouts could not have detected.

**Limitations and future work.** The agent has no model of the actual student field, only of the staff baselines, the zoo, and our own held-outs. The most plausible source of remaining error is therefore an opponent strategy class that we did not anticipate. Online weight adaptation (`your_best_adaptive.py`) was investigated as a hedge: in paired-difference testing against the same opponent set with shared random seeds, the adaptive variant produced game-for-game identical outcomes for all 40 paired games (mean and standard error of the per-game difference were exactly zero). With shared random seeds, identical actions imply that the online weight updates never crossed the granularity needed to flip even a single action choice within the 1200-move horizon, so the adaptive variant was behaviourally indistinguishable from the static one. We therefore submitted the static variant, which is simpler and more reproducible.

## Acknowledgements

The Pacman framework was developed at UC Berkeley by John DeNero and Dan Klein for the CS188 course.

**AI usage disclosure. (1) AI used.** Anthropic Claude (Opus 4.7, `claude-opus-4-7`) via the Claude Code CLI. **(2) Prompts given to the AI.** Three categories: (a) code review of `your_best.py` and `train_selfplay.py` for deadline violations and edge-case bugs; (b) prose polish of an already-drafted version of each section of this report; (c) boilerplate generation for the `benchmark.py` harness used in paired-difference testing. *AI-assisted parts in positive form:* (a) line-level review of `chooseAction` error handling and A$^\star$ deadline guards; (b) English

wording polish of the abstract and Sections 1–4; (c) `benchmark.py`, `sweep.py`, `holdout_eval.py`, and `apply_v5_weights.py` skeletons. **(3) Authored by the student.** The agent architecture (goal-commit A$^\star$ offense, alpha-beta minimax defense, 42-feature linear evaluator, opponent classifier, anti-overfitting protocol), all 42 weight values from the self-play optimizer, the training and held-out zoo opponents, the design of the empirical methodology including the negative-result and SOTA calibration experiments ($n$=100 ten-layout audit with 95% Wilson confidence intervals, side-swap, and the 792-game external sweep of Section 3.7), and every ship/revert decision — including the rejection of all five negative-result experiments of Section 3.6 — were authored by the student. AI assistance did not change any algorithmic decision. **(4) How AI responses were verified.** Every numeric value in the abstract, tables, and figures was independently regenerated by the student via `python capture.py -r 20266008 -n 10` and `python benchmark.py` on the local machine, with the actual output files (`output.csv` and the JSON benchmark dumps) preserved as ground truth. Only changes that did not regress the held-out audit and kept the runtime error rate at 0% were retained.

## References


[1] DeNero, J., and Klein, D. (2020). *Pacman Capture the Flag — CS188 Mini-Contest 1*. UC Berkeley CS188 course materials, https://inst.eecs.berkeley.edu/~cs188/sp20/minicontest1/.

[2] DeNero, J., and Klein, D. (2009). *Contest: Pacman Capture the Flag*. UC Berkeley CS188 (Spring 2009) project page, https://inst.eecs.berkeley.edu/~cs188/sp09/projects/contest/contest.html.

[3] Creed, A., et al. (2020). *Artificial Intelligence — Pacman Capture The Flag (Team Pacamon)*. COMP90054 AI Planning for Autonomy, University of Melbourne, 4th–5th place, 68.5% win rate over 7012 games, https://abhinavcreed13.github.io/projects/ai-team-pacamon/.

[4] Creed, A., et al. (2020). *ai-capture-the-flag-pacman-contest (Pacamon source)*. GitHub repository, https://github.com/abhinavcreed13/ai-capture-the-flag-pacman-contest.

[5] Wang, R., et al. (2020). *COMP90054 Pacman Contest Project (Team Alpha, ranked 8/88)*. GitHub repository, https://github.com/infinityglow/COMP90054-Pacman-Contest-Project.

[6] Bachfischer, M. (2021). Reflections on Pacman AI Competition. COMP90054 personal blog, https://bachfischer.me/posts/2021/02/reflections_on_pacman_ai_competition/.

[7] Hammarstedt, J., et al. (2021). *PacmanAI: Pacman Capture the Flag AI (KTH DD2438 inter-class tournament winner; pure heuristic A$^\star$ + risk assessment)*. GitHub repository, https://github.com/jhammarstedt/PacmanAI.

[8] Roussel, L. (2024). *pacman-ctf-agents (CMSI 4320, 1st place in class final tournament; constrained Q-learning with particle filter)*. GitHub repository, https://github.com/loosh/pacman-ctf-agents.

[9] A., G. (2022). *Pacman Capture the Flag — multi-agent planning and Bayesian inference for competitive AI gameplay*. GitHub repository, https://github.com/95-ag/pacman-capture-the-flag.

[10] Sundberg, J. (2017). *Pacman-Tournament-Agent: a Deep Q-learning and Max-Flow agent (highest win/lose ratio in class)*. GitHub repository, https://github.com/jaredjxyz/Pacman-Tournament-Agent.

[11] Ren, S. (2020). *Pacman_AI_Agents: A$^\star$ Heuristic Search and PDDL Classic Planning for Pacman Capture the Flag*. GitHub repository, https://github.com/sren1618/Pacman_AI_Agents.

[12] Universitat Pompeu Fabra (2024). *Results for PACMAN Capture the Flag Tournaments*. UPF Pacman contest results portal, https://pacman-contest.upf.edu/.

[13] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

[14] Russell, S., and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson; ch. 5–6 for adversarial search and minimax with alpha-beta pruning.

[15] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

[16] Kocsis, L., and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *Proc. ECML 2006*, LNCS 4212, pp. 282–293.

[17] Sutton, R. S., and Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press; ch. 6 (TD methods) and 10 (function approximation) for approximate Q-learning background.

[18] Hoffmann, J. (2003). The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341.

[19] Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. (2017). Population based training of neural networks. arXiv preprint arXiv:1711.09846.

[20] Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212.

[21] Ford, L. R., and Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404.