Project II Report for COM S 4/5720 Spring 2025: Pursue-Escape Planning with Dynamic Rollouts

Jesus Soto Gonzalez

Abstract— **Project II extends the concepts developed in Project I into a multi-agent pursue-escape environment. While Project I focused on building a path planning algorithm for static grid navigation tasks, Project II required reactive and adaptive decision-making against moving opponents. I took inspiration from Monte Carlo Tree Search techniques to develop an agent that dynamically simulates future moves and plans robust actions under uncertainty. The agent balances pursuit, evasion, and spatial control while introducing slight randomization to avoid predictability. This report details the design, strategies explored, full function-level implementation, and how lessons from Project I informed my approach.**

I. Introduction

In Project I, the A* path planning algorithm enabled me to find optimal paths through static grid environments by minimizing the cost function $f(n) = g(n) + h(n)$. A key lesson I learned was how combining real costs with heuristic estimates leads to efficient and optimal navigation in structured spaces.

In Project II, the challenge expanded significantly since agents must not only plan paths but react and develop strategies against dynamic opponents sharing the environment. This shift meant that A* planning was no longer sufficient. I needed a method capable of short-term tactical reasoning based on anticipated opponent behavior. This motivated me to use ideas inspired by Monte Carlo Tree Search and adapt them into a lightweight, shallow rollout version suitable for real-time multi-agent competition.

II. Strategy and Integration of Project I Concepts

Core principles kept from Project I:

- Using cost estimation heuristics to guide decision-making.
- Prioritizing valid moves by checking bounds and avoiding obstacles.
- Efficiently selecting among candidate actions.

However, unlike A* which plans a complete path, my Project II agent plans only the next immediate action based on simulated future outcomes. Instead of seeking global optimality, my agent seeks a local advantage at each step. This shift required:

- Simulating multiple future possibilities or rollouts per candidate move.
- Scoring outcomes using a heuristic that balances pursuit, evasion, and mobility.
- Introducing randomness to avoid deterministic behavior against opponents.

Thus, while A* used a single search tree, my Project II agent builds shallow, dynamic trees rooted at the current state each turn.

III. Development of Final Planner

A. Early Strategies

Initially, I experimented with multiple simpler strategies:

- **Pure Greedy Chase**: Always moving directly toward the pursued agent. This performed poorly because it ignored threats from the pursuer.
- **Defensive Escape**: Always maximizing distance from the pursuer. This led to passive behavior and frequent missed opportunities to win.
- **Static Rollout (Fixed Depth)**: Simulating fixed-length futures but not adapting depth based on danger. It performed better than greedy strategies but was inflexible under pressure.
- **Two-Step Lookahead**: Attempting to simulate both the current and next moves at once. However, this approach made assumptions about opponent moves and often introduced prediction errors.

I also experimented with similar strategies by adjusting heuristic weights and tuning parameters related to aggressiveness, defensiveness, and mobility. By testing these strategies against each other, I gained insights into the strengths and weaknesses of different approaches. This analysis helped me ensure that the dynamic, shallow MCTS-style planning provided an effective balance of flexibility, reactivity, and robustness, making it a strong fit for the pursue-escape task.

B. Final Strategy

The final agent uses a dynamic, shallow MCTS-style approach:

- For each candidate move, I simulate multiple rollouts (depth 3 or 8 depending on threat proximity).
- Each rollout simulates my agent's moves, the pursued agent's escape moves, and the pursuer's chase moves.
- Rollouts accumulate a heuristic score that balances pursuit, evasion, mobility, and positional safety.
- Among candidate moves, the one with the highest average score is selected, breaking ties randomly.

This evaluation balances competing goals dynamically based on current surroundings, producing adaptive and robust behavior.

IV. Implementation Details

A. *plan_action*

Each timestep, `plan_action()` evaluates all nine possible moves (stay put and eight directions). Illegal moves are filtered out using `is_valid_move()`. For each legal move, I simulate five random rollouts. The cumulative scores are averaged, and the move with the highest expected value is chosen, introducing random tie-breaking among equally good candidates.

B. *simulate*

The `simulate()` function executes a sequence of simulated actions:

- A random danger trigger (distance 3, 4, or 5) determines when to change the simulation depth.
- If escape options are few (3 or fewer valid moves), my agent switches to an escape mode, choosing moves to maximize distance from the pursuer.
- Otherwise, my agent greedily chases the pursued.
- A tiny 1% chance per step injects random moves to introduce unpredictability.
- Simulations end early if either my agent catches the pursued or is caught.

C. *pick_smart_move, pick_escape_move, pick_random_move*

Movement is selected during simulation:

- `pick_smart_move()` selects moves that decrease the Manhattan distance to a target.
- `pick_escape_move()` selects moves that increase distance from a threat.
- `pick_random_move()` randomly selects a legal move, used rarely to enhance resilience against predictable behavior.

D. *evaluate*

The `evaluate()` function computes the score for a position based on:

- $-1.1\times$ Manhattan distance to the pursued, encouraging capture.
- $+0.9\times$ Manhattan distance from the pursuer, encouraging safety.
- $+0.3\times$ center bonus to promote high-mobility positions.
- $-0.3\times$ wall penalty to avoid confinement near map edges.
- -2 penalty if within 2 units of the pursuer.

E. *count_valid_moves*

The `count_valid_moves()` function counts the number of legal moves available from a given position. Low counts signal potential danger or traps, triggering escape behavior during rollouts.

F. *is_valid_move*

The `is_valid_move()` function checks movement legality:

- Boundaries must not be crossed.
- Destination cells must not be occupied by walls.
- Diagonal moves are only permitted if both adjacent orthogonal squares are free, preventing illegal corner-cutting.

G. *manhattan_distance*

The `manhattan_distance()` function computes the sum of absolute differences between coordinates. It is used for pursuit and evasion heuristics and center bonus scoring.

H. *wall_penalty and center_bonus*

The `wall_penalty()` function adds penalties based on proximity to walls and map edges. The `center_bonus()` function computes a reward based on proximity to the grid center, encouraging higher mobility and more escape options.

V. Results and Observations

Through extensive testing across many maps, the final agent consistently:

- Escaped threats when cornered.
- Pursued targets efficiently when safe.
- Maintained central positioning and high mobility.
- Demonstrated robustness against both passive and aggressive opponents.

It outperformed earlier greedy, static, and two-step lookahead strategies by changing the rollout depth, switching between chasing and escaping, and avoiding traps.

VI. Conclusion

By extending the search and evaluation principles developed in Project I, I implemented a lightweight MCTS-inspired agent capable of responsive pursue-escape decision-making. Instead of rigidly following static paths, the final implementation adapted in real time, reacting to both threats and opportunities. Dynamic rollout depth, escape detection, randomized behavior injection, and a carefully tuned evaluation function enabled robust, competitive performance against a variety of opponent strategies.

References

1 P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, 1968.

2 C. Browne et al., "A Survey of Monte Carlo Tree Search Methods," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, 2012.