Project II Report for COM S 4720 Spring 2025: Pursue-Escape Planning Algorithm Development

Tanvi Mehetre[1] and Tejal Devshetwar[2]

I. INTRODUCTION

Project II extends from Project I as it extends into a complex multi-agent setting involving pursuit and evasion among three agents namely: Tom, Jerry and Spike. Each agent has two roles to play: pursuing one agent while simultaneously trying to evade another. Specifically:

- Tom is trying to catch Jerry
- Jerry is trying to catch Spike
- Spike is trying to catch Tom

Agents will be navigating a shared 2D grid world with obstacles are static. The goal of the project is to implement an intelligent planning algorithm that can make optimal movement decisions in this adversarial setting. The dynamic nature of the agents' goals and threats from other agents give rise to difficulty.

II. IMPLEMENTATION

Three agents are involved in the pursue-escape simulation—Tom, Jerry and Spike—each of which is controlled by a custom planner implementing an adversarial search algorithm. We are tasked with creating a planner algorithm which will be tested against other planner algorithms after submission. Our solution is modular, with the planner script using a planning template based on Minimax tree searching.

A. Overview

Our planning algorithm has the agent use a depth-limited Minimax algorithm to simulate the future moves for itself, its target, and its pursuer. The planner will generate all possible legal moves defined for each agent and recursively evaluate the outcomes using a heuristic evaluation function.

The architecture includes:

- **Agent Planner:** Encapsulates planning logic and decision tree traversal.
- **Minimax Algorithm:** Handles alternating turns and evaluates multi-agent interactions.
- **Heuristic Function:** Scores board states based on distance metrics and capture conditions.
- **Move Generation:** Determines valid 8-connected movements while filtering out obstacles.

[1]Tanvi Mehetre is a student in the Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, United States tanvim@iastate.edu

[2]Tejal Devshetwar is a student in the Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, United States tejal@iastate.edu

B. PlannerAgent Class

The `PlannerAgent` class is the core interface for each agent. It initializes with the following information:

- Agent's ID (TOM, JERRY, or SPIKE)
- Current game state (grid size, positions, walls)
- Target agent (who this agent is pursuing)
- Threat agent (who is pursuing this agent)

On each turn, the planner invokes the `minimax()` method to simulate moves for itself, target and the pursuer.

C. Minimax Algorithm

The function `minimax(position, depth, maximizingPlayer)` is made for recursively evaluating potential future states. It alternates between maximizing the utility of the planning agent and minimizing it that is simulating the opponents moves and trying to aim to make the best counter move that is in the agents favor.

- **Terminal Conditions:**
  - The agent successfully catches its target (receives a positive reward).
  - The agent is caught by its pursuer (receives a negative reward).
  - The search reaches the maximum allowed depth (a heuristic evaluation is performed).
- **Branching:** Each agent can choose among 9 actions: moving to 8 neighboring tiles or staying in the current place. Each valid move is considered in the tree.

The depth of the tree is set to 3. At each depth level, different agents are simulated depending on turn order.

D. Evaluation Function

The evaluation function captures both catching the target and escaping from the pursuer strategies. It is defined as:

$$\text{Score} = \begin{cases} 1000, & \text{if agent catches its target} \\ -1000, & \text{if agent is caught} \\ -d_{target} + 1.5 \cdot d_{threat}, & \text{otherwise} \end{cases} \tag{1}$$

Here, d_{target} is the Manhattan distance to the target, and d_{threat} is the distance from the pursuer. While the agent approaches its target, it still maintains a safe distance from its threat.

E. Move Generation

For each position, we consider:

- Horizontal, vertical, and diagonal steps as the 8 directions for the movement of the agent
- Staying in the current position
- Filter out positions that are outside the grid for a given position and the positions that are blocked by the obstacles

This results in a branching factor $b \leq 9$. Move generation is wrapped in a utility function `get_valid_moves(position)` for a modular function approach.

F. Pathfinding with A* (Fallback)

For agents needing direct pursuit behavior, we include an A* pathfinding function. It computes the shortest path to a target location. The Manhattan distance is the heuristic used by the A* search, and avoids threats by adding movement penalties.

When adversarial search is too costly or unnecessary this will be useful. For example, when agent is far away from other agents.

G. Agent Interactions

The planner is aware of the positions of the other two agents and uses their current positions to simulate the following:

- Its own future move (maximize)
- The response of the target and the threat (minimize)

A situation where multiple actions have equal evaluation can arise and to prevent the deterministic cycles or ties, random selection is included as well.

H. Sample Execution Flow

On each turn:

1) Obtain the current positions of all agents.
2) Generate valid moves for each.
3) Run Minimax to fixed depth, simulating alternating turns.
4) Use evaluation scores to pick the best move for the agent.
5) Return that move to the simulator.

This cycle repeats for the agent assigned. The other planners that we will be tested against will have a different flow of execution.

III. TIME COMPLEXITY

The minimax algorithm has time complexity:

$$O(b^d)$$

where b is the branching factor (up to 9 for 8 directions + staying at same place) and d is the lookahead depth. For depth 3:

$$O(9^3) = 729 \text{ node evaluations}$$

This cost increases exponentially, but remains computationally feasible due to the limited grid size and depth.

IV. CONCLUSIONS

Our pursue-escape planning algorithm successfully applies a game-theoretic minimax approach to adversarial multi-agent planning. A balance between pursuit efficiency and evasion safety is provided using a tunable evaluation function. This structure allows us to flexibly adapt to the multiple agent roles that have been mentioned (Tom, Jerry, Spike).

Overall, our approach showcases the effectiveness of classical adversarial search—specifically, the Minimax algorithm—in multi-agent robotic planning. The use of terminal conditions, evaluation heuristics, and depth-limited search enables strategic and tractable decision-making, allowing agents to pursue targets while avoiding threats in a dynamic environment.

REFERENCES

"Minimax Algorithm in Game Theory: Set 1 (Introduction)." GeeksforGeeks, GeeksforGeeks, 13 June 2022, www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/.

"A* Search Algorithm." GeeksforGeeks, GeeksforGeeks, 30 July 2024, www.geeksforgeeks.org/a-search-algorithm/.