Pacman Capture the Flag
Assignment 4 Report

GROUP 03

Shekhar Devm Upadhyay
20010531
sdup@kth.se

Aiman Shenawa
001021
ashenawa@kth.se

May 2023

Abstract

This report presents a study on the development of AI agents for controlling a team of two in a competitive Pac-Man Capture the Flag (CTF) scenario. The objective of the agents is to maximize their score by efficiently collecting food from the enemy turf within a limited number of moves. This report studies various approaches used in the field of artificial intelligence, including minimax game tree search, modifications like alpha-beta pruning and move ordering, Monte Carlo Tree Search (MCTS), Expectimax, and decision-tree-based approaches, presents the final solution employed by the authors, and does a comparative analysis of the approaches utilized by all the teams in the competition. By analyzing and comparing these strategies, we aim to identify effective techniques for AI-controlled agents in Pac-Man CTF.

1 Introduction

Pacman capture the flag is a project developed by UC Berkely, it is a challenge that combines classic video game mechanics with artificial intelligence techniques. The goal of the project is to develop an agent that can play the game of Pacman capture the flag. The game is played on a grid, where the agent can move in four directions, up, down, left and right. The grid is split into two halves, in each half there is a team of Pacman agents, the goal of the game is to capture the food pellets from the other team's side and bring them back to your side while also preventing the other team from doing the same.

1.1 Task Description

This report will discuss ....

These problems and their solutions are addressed in further detail in the following section 2.

1.2 Contribution

1.3 Outline

2 Related Work

The success of AI agents in Pac-Man CTF heavily relies on the choice and implementation of appropriate strategies. This section provides an overview of the relevant research in this domain.

2.1 Minimax Game Tree Search

The problem of finding the optimal move in a game can be formulated as a search problem. The search space is a tree, where each node represents a state of the game, and each edge represents a possible move. The root node represents the current state of the game, and the leaf nodes represent the terminal states of the game. The goal is to find the optimal path from the root node to a leaf node.

The minimax algorithm (shown in Algorithm 1) is a recursive algorithm that computes the optimal move for a player in a two-player game. The algorithm assumes that the opponent plays optimally, and it tries to minimize the maximum loss that can be incurred by the player. It is a classic approach in Game Theory, aiming to find the optimal move in a two-player, zero-sum game. By constructing a game tree and evaluating the utility of different game states, the algorithm enables agents to make informed decisions.

Sometimes, however, it is not feasible to search the entire game tree, due to the large number of possible moves. To address this issue, the algorithm can be modified to limit the depth of the search tree. The algorithm can also be modified to incorporate heuristics that evaluate the utility of non-terminal game states. Variations of minimax, such as alpha-beta pruning, move ordering based on heuristics, iterative deepening search (IDS), and Repeated States Checking (RSC), have been proposed to improve its efficiency and effectiveness [1][2][3].

Algorithm 1 Minimax Algorithm

```
1: procedure MINIMAX(node, depth, maximizingPlayer)
2:     if depth = 0 or node is a terminal node then
3:         return the heuristic value of node
4:     end if
5:     if maximizingPlayer then
6:         value ← −∞
7:         for each child of node do
8:             value ← max(value, Minimax(child, depth − 1, False))
9:         end for
10:        return value
11:    else
12:        value ← ∞
13:        for each child of node do
14:            value ← min(value, Minimax(child, depth − 1, True))
15:        end for
16:        return value
17:    end if
18: end procedure
```

2.2 Expectimax

Expectimax is a variant of the minimax algorithm that considers uncertain outcomes in games. It is commonly used in domains with probabilistic elements. In Pac-Man CTF, Expectimax can be employed to account for the ghost movement and uncertain states, enabling agents to make rational decisions under uncertainty [6].

2.3 Monte Carlo Tree Search

MCTS is a sampling-based search algorithm that has demonstrated remarkable success in various game-playing domains. By combining tree exploration with random rollouts, MCTS performs effective exploration and exploitation of the search space. This approach has been applied to Pac-Man CTF, where agents simulate multiple playouts to estimate the value of different actions and make informed decisions [4][5].

2.4 Decision Tree Based Approaches

Decision trees offer a structured representation of decision-making processes. By learning from historical data and constructing decision rules, decision-tree-based approaches provide an interpretable framework for agent behavior. These techniques have been explored in Pac-Man CTF, where agents use decision trees to guide their actions based on various game state features [7][8].

2.5 Hybrid Approaches

Several studies have proposed hybrid strategies by combining different AI techniques in Pac-Man CTF. For example, integrating MCTS with minimax or Expectimax algorithms has shown promise in improving agent performance [9][10]. Such hybrid approaches leverage the strengths of different methods to create robust and adaptable AI agents.

3 Method

In this section, we will discuss ...

4 Experiments and Results

First, we will describe the experimental setup for the two problems, and then we will present the results for each of the problems separately.

4.1 Experimental Setup

The experimental environment was largely dependent on the problems. Both problems shared a a similar map and obstacle structure, the main difference being the agent objectives, environment sizes, and the fact that we didn't have any obstacles in the soccer environment (except, of course, for the boundaries of the field). For each of the problems, we were to compute and pass a set of inputs to the agents (different based on whether the agent was a car or a drone). The agents moved based on the provided input, and the updated positions and velocities were used in the next timestep to compute the next set of inputs. The agents were also provided with a set of parameters that decided how the different forces were weighted. The parameters were tuned by hand separately for each type of agent, and were not changed during the course of the simulation.

All the solutions for Collision Avoidance were repeatedly tested in the Unity game environment on three terrains: an open field, an intersection, and a random map. The open field was a simple map with no obstacles. The intersection was a map with a crossroad and a few obstacles. The random map was a map with randomly generated obstacles. There were two possible initializations for the positions of the agents: first, in a circular formation, and second, in a random formation. The solutions were tested in the Unity game environment with 50 agents. Based on performance, the solutions were improved and adjusted in an iterative manner.

4.2 Results

In this section, we summarize the results of our experiments. We first present the results of the Collision Avoidance problem, and then present the results of the Soccer problem. We then go on to discuss the merits and shortcomings of the best solutions from all the groups.

5 Summary and Conclusions

A Appendix