«««< HEAD ======= «««< HEAD ======= »»»> 0174b6137125f5376dad55ea9086b227dee81509 »»»> d3d645932d600bcf137f8e6ed68adb0588dcaa14

Pacman Capture the Flag
Assignment 4 Report

GROUP 03
Shekhar Devm Upadhyay Aiman Shenawa
20010531 20001021
sdup@kth.se 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. Then, it goes on to present 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

With the continious development of society and technology, the need for autonomous agents is increasing. It is becoming more and more important to develop agents that can operate in dynamic and complex environments autonomously. In this report we will explore the problem of developing an agent that can play the game of Pacman capture the flag. This is a project developed by UC Berkely [UC 15], 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. Multiple aspects of artificial intelligence will be explored in this report, such as minimax game tree search, Monte Carlo Tree Search (MCTS), Expectimax, and decision-tree-based approaches. The final solution included the use of Minimax and alpha-beta pruning.

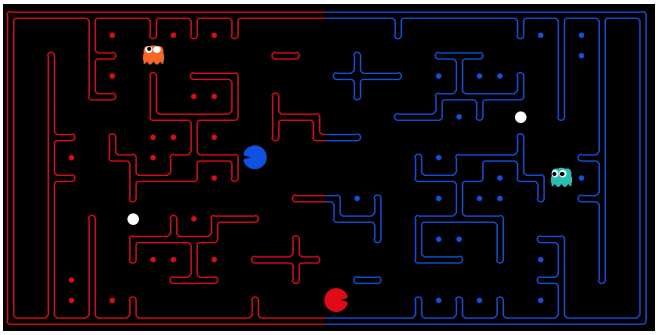

Figure 1: Illustration of environment

Figure 1 shows the game environment. The maze structure can be changes by specifying random seeds. The blue and red dots represent the food pellets, the blue and red lines represent the walls. As seen in the figure above there are both Pacman and ghosts. The agent is a ghost when defending its own side. When crossing over to the enemy side to collect pellets it becomes a pacman. An agent can go into enemy territory and collect pellets, but the score will only increase when the agent returns back to its own side. In the figure there are also white pellets, those are power pellets, when an agent collects a power pellet it can eat the enemy agents for a period of 40 moves. During that period the enemy ghost agents are regarded as scared. When an agent is eaten it respawns at its own side, if the agent was scared before getting eaten it will no longer be scared after respawning. The game ends when all but two food pellets have been eaten, or when the time limit of 1200 moves has been reached. The team with the highest score wins the game. There is a limit on calculation time, 15 seconds for initialization and 1 second for each move. If the agent exceeds the time limit three times during a game the team will forfeit the game.

The information each agent has access to is the following:

- The friendly agent's positions
- The total score
- The position of all the food pellets on the map, this is updated every time an agent moves

- The position of all the power pellets on the map, this is updated every time an agent moves
- The position of an enemy agent if it is within Manhattan distance of 5 from us or a friendly agent.

1.1 Task Description

This report will discuss the problem of developing an agent that can play the game of Pacman capture the flag. The aim is to develop two agents which are able to defend their side of the map and also collect food pellets from the enemy side. The agents will be able to move in four directions, up, down, left and right. It is also possible to stay in the same position and not move, although this is only sometimes strategically beneficial.

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

1.2 Contribution

In this report we will impement Minimax and alpha-beta pruning into the UC Berkeley Pacman Capture the Flag project. We will also explore the use of Monte Carlo Tree Search (MCTS), Expectimax, and decision-tree-based approaches. We will also compare the results of our solution with the results of other methods used for the same problem.

An essential aspect of Pacman capture the flag is the ability to cooperate as a team and communicate with the other agent. Potential real world applications of this project are many, as the ability of intelligent agents to make strategic decisions in a dynamic environment has many applications. Applications include multi-robot system coordination, autonomous vehicles, and unmanned aerial vehicles.

1.3 Outline

This report is organized as follows. First we introduce the problem and its importance in Section 1. Then we discuss and highlight the related work and required backround knowledge in order to understand the proposed solution in Section 2. This is followed by a detailed description of the proposed solution in Section 3. Next, we present the experimental setup and the results in Section 4. We summarize the results and conclude the report in Section 5. Lastly, we present improvements that can be made to the suggested solution in Section 6.

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 [NM44] (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 [CHH02].

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. The utility of a state with uncertain actions is the weighted average of the utilities of the possible outcomes, where the weights are the probabilities of the outcomes. For instance, one could assume that the opponent moves randomly, and compute the expected utility of each action. Building on this, expectimax tries to maximize the expected utility of the maximizing player. In Pac-Man CTF, Expectimax can be employed to account for uncertain states and not knowing what moves the enemy could make, enabling agents to make rational decisions under uncertainty [Riv87].

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a sampling-based search algorithm that has demonstrated remarkable success in various game-playing domains. In particular, the approach gained significant popularity due to its success in various board games, such as Go, Chess, and Shogi [GS11]. MCTS combines random simulations and tree search to make informed decisions. Unlike traditional search algorithms like Minimax, MCTS does not rely on an explicit evaluation function or an exhaustive exploration of the game tree.

In MCTS, the algorithm builds a search tree incrementally by performing a series of simulations. Each simulation consists of four main steps: selection, expansion, simulation, and backpropagation. During the selection step, the algorithm traverses the tree from the root node to a leaf node, employing a policy to decide which child node to explore next. The expansion step adds one or more child nodes to the tree. The simulation step plays out the game from the expanded node using random moves until a terminal state is reached. Finally, during the backpropagation step, the results of the simulated game are propagated back up the tree, updating the statistics of the visited nodes. This approach has been applied to Pac-Man CTF by Group 4, where agents simulate multiple playouts with heuristics to estimate the value of different actions and make informed decisions.

2.4 Decision Tree Based Approaches

Decision trees provide a structured representation of decision-making processes, offering interpretable and rule-based frameworks for agent behavior. These techniques have been extensively explored in various domains, including classification, regression, and data mining. In recent years, researchers have begun to investigate the potential of decision trees in the context of turn-based games, recognizing their ability to capture complex game dynamics and facilitate agent decision-making [HTF09].

One of the primary advantages of decision-tree-based approaches is their interpretability. The resulting decision rules are often intuitive and understandable, allowing humans to gain insights into the decision-making process. This interpretability is particularly valuable in turn-based games, where strategic decision-making and planning play a crucial role. By examining the decision tree, players and developers can gain a clear understanding of the agent's behavior and identify areas for improvement or adjustment.

In the context of Pacman CTF, we have seen Group 2 use such an approach to great success, establishing the effectiveness of decision trees in this domain.

3 Method

In this section, we will discuss the solution proposed by our team. The solution is based on the Minimax algorithm, with some modifications to account for the limitations of the UC Berkeley Pacman Capture the Flag project. We will also discuss the motivations behind our heuristic functions.

3.1 Modified Minimax

Typical Minimax approaches require that the entire game state be observable to us. Firstly, the UC Berkeley challenge does not provide us with the entire game state - we can only see the positions of our team members, and nearby enemies. Secondly, even if we knew the positions of all the enemies, if we consider the movements of both agents of our team as a single action, then the branching factor of the game tree is too large to be explored in a reasonable amount of time. So, we decided to modify the typical Minimax approach as follows:

- At each time step, we consider the possible actions of the agent we are controlling (as the maximizing player). If there is an enemy agent visible to us, we consider the possible actions of the enemy agent as well, by passing control to the enemy as a minimizing player. If not, we continue to consider the possible actions of our agent (by exploring further based on moves that we can make in the next time step as the maximizing player).
- We use a heuristic to evaluate the utility of a game state, instead of evaluating the utility of a terminal state. We compute heuristics until a certain depth of the game tree, and use iteratively deepening search to explore the game tree further if time permits.
- We use alpha-beta pruning to reduce the number of nodes that we need to explore in the minimax exploration of the game tree. To further speed up the search, we use move ordering based on heuristics to explore the most promising nodes first, and implement repeated states checking to avoid computing values of the same game state multiple times when working with deeper game trees.

3.2 Heuristics

We started off by having separate classes for offensive and defensive agents, each with a different heuristic. Later, we merged the two classes into a single class, and used a attribute called `mode` to determine if the behavior should be defensive or offensive, based on current agent position, position of our teammate, positions of any visible enemies, and other relevant factors affecting the game state.

3.3 Offensive Heuristics

When in "offensive" mode, our agent tries to cross over to the other side of the map as quickly as possible, then actively looks for food pellets to capture. We prioritize closer food (without disregarding the rest), de-value food inside dead-end paths (paths that can be choked off at a single point by the enemy) pick up power pills when possible and when enemy agents are closeby. As the food in our pocket increases, we have an increasing tendency to return to our side of the map to deposit the food. We also try to avoid enemy agents when possible and try to eat them when they are scared. Being chased by an unscared enemy also "turns off" the tendency to chase food, and we try to run away from the enemy instead. We also try to avoid getting trapped in dead-end paths, and try to avoid getting cornered by enemy agents. After depositing the food, we start this process again.

We noticed that the offensive agents sometimes got stuck in local extrema of the value function. For example, sometimes the agent sat next to a food particle and didn't eat it, until the enemy agent came closer and pushed it into scramble mode. To avoid this, we added an increasing

tendency to return to our own side for each time step in which the chosen action was to stay in the same place. This helped the agent get out of local extrema in a couple of moves, in most cases.

3.4 Defensive Heuristics

For the "defense" mode of the agent, we incentivize minimizing the sum of distances to all the food we are defending. Since food near the center line is easier to take by the enemy, we give more weight to being close to those food particles. We keep track of the food that is being defended by our agent, and if some of it disappears, we know that an enemy agent has eaten it. We try to chase the enemy agent based on direct sight, knowledge from our teammate, and disappearing food. If we are close enough to the enemy agent, we move to trap and eat it we are not scared. If we are scared, we try to run away from the enemy agent. In no circumstance does the agent go over to enemy territory when in defense mode.

The exact details of our implementation can be found in our Github Repository and submitted final code.

4 Experiments and Results

First, we will describe the experimental setup for the Pacman CTF game, and then we will present the results of our experiments. We will also discuss the merits and shortcomings of our solution, and compare it with the solutions of other groups.

4.1 Experimental Setup

We ran our experiments on the UC Berkeley Pacman Capture the Flag project. We used the provided baseline agents as a starting point, and modified them to implement our solution. We used the provided `capture.py` script to run the experiments, modifying only the `myTeam` file to use our agents. We used the provided `capture_agent.py` file as a starting point for our agents, and modified it to implement our solution as described in Section 3.

All teams agreed on specific map sizes, seeds, and number of agents per team for the pre-final and final tournaments. We also agreed on a maximum compute time for each move, to ensure that the games would finish in a reasonable amount of time. One of the drawbacks of this approach is that different machines may be able to process different amounts of the game tree in the given time, and hence the results may not be completely fair. However, we believe that this is a reasonable trade-off, since it is not possible to ensure that all machines are equally powerful. We did not explicitly penalize teams for exceeding the time limit, relying on the honor system instead. There was a similar limit on the amount of time that a team could take upon initialization, to ensure that the games would start in a reasonable amount of time. We ran into situations where some teams still took a much longer time to initialize, leading to delays in the start of the game. We have not explicitly penalized teams for this, but we believe that this is something that should be taken into account in future tournaments.

4.1.1 Experimental setup for measurement of performance

In order to evaluate the performance of the suggested solution, we will compare our approach with five other groups. This is done by putting the algorithms against each other in the simulation. Two different evaluations will be done, one called pre-finals, in which the groups give their initial solutions, and one called finals, in which after having time to re-evaluate their solutions, the groups give their final solutions. The results of both evaluations are presented in the tables below. Results are measured in terms of wins, ties, losses. The score is calculated as the following: 3 points for a win, 1 point for a tie, and 0 points for a loss. The total score is the sum of the points for each group.

4.2 Results

In this section, we summarize the results of our experiments. First we present the results of the two evaluations in the competition, then we present the approach of the winning group.

Group	**Wins**	**Ties**	**Losses**	**Total Score**
4	8	0	0	24
2	5	1	0	16
6	5	1	3	16
3	3	5	2	14
1	0	7	3	3

Table 1: Results of the pre-finals evaluation.

Group	**Wins**	**Ties**	**Losses**	**Total Score**
4	13	1	1	40
2	9	2	4	29
6	7	2	6	23
3	4	4	6	16
1	0	13	2	2

Table 2: Results of the finals evaluation.

As agreed upon between the groups, we ran the final games on maps of size 32x32, with 2 agents on each team, with varying seeds determining the map layout. We ran one game against each group for each seed, and recorded the results in Table 2. In contrast, the pre-final tournament was run on maps of size 16x16. The results of the pre-final tournament are shown in Table 1. In both cases, to compute the final scores, we gave 3 points for a win, 1 point for a tie, and 0 points for a loss.

During the pre-final, our agents were reasonably good, but we still had some bugs that caused them to perform poorly in some situations. For example, sometimes our offensive agent thought it was better to sit next to a food pellet than to eat it. Sometimes, our defensive agent kept the enemy in sight without moving in for an obvious kill. At this point, our agents still had fixed roles (one attack and one defense), and did not switch roles based on the state of the game.

Seed	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6
31	win	loss	NA	tie	loss	loss
89	win	win	NA	tie	loss	win
489	tie	tie	NA	loss	win	loss

Table 3: Performance of our agents against other groups in the final tournament

Between the pre-final and final tournaments, we fixed the bugs in our agents, and also implemented the role switching mechanism described in Section 3.2. This allowed our agents to switch roles based on the state of the game, and also to switch roles if one of the agents was killed. This improved the performance of our agents significantly in our testing. However, we overlooked something quite important - the size of the map on which we tested our "Fluid" Agents was much smaller than the size of the map on which the final tournament was run. This meant that our agents were not able to switch roles as seamlessly, often leaving gaps in our defense. This was compounded by the fact that our agents were not able to see the entire map, and hence could not plan their actions accordingly. This led to our agents performing poorly in the final tournament. We believe that if we had tested our agents on larger maps and spend some more time tuning the parameters, we would have been able to perform much better in the final tournament.

Observing the results presented in Table 3, we can see that our defence worked well against the best scoring groups. What we lacked in our solution was a good performing offence that could score against the other groups' defences as well as a robust role switching mechanism that is independent of map sizes.

From Tables 1 and 2, it is clear that the overall winner of the tournament was Group 4, who had the highest scores on both the pre-final and final tournaments. There was an issue where they took about 4 minutes on most teams' machines to initialize their agents instead of the agreed 15 seconds, and could thus be disqualified, but we agreed to let them participate in the tournament anyway. Even on their own computers, their code took about 2 minutes to initialize; the reason was that they had tuned their algorithm to use about 15 seconds for a smaller (16×16) map, but a larger map meant that their algorithm took much longer to initialize.

Their approach relies heavily on a Monte-Carlo Tree Search, coupled with a heuristic function to evaluate the value of a game state instead of using rollouts to the end of the game. They also had a very good estimator that allowed them to "guess" where the enemy agents could be, and act accordingly. This allowed them to perform very well in the tournament.

Group 2 came in second place, with both the offensive and defensive behaviors being based on a simple Decision Tree based approach. They had one defensive agent and one offensive agent, with no role switching mechanism. They also estimated the current position of the enemy agents, based on an HMM model. This gave them good intelligence on the enemy agents, and allowed them to perform well in the tournament.

Group 6 also had an interesting approach - their defensive behavior was based on a simple Decision Tree based approach, while the offensive behavior used a minimax implementation quite like our own. They had a much better role-switching mechanism than ours: when they had a lead of more than 10 points, both their agents would switch to defense. If their defense was scared, they would switch to offense and focus on eating food pellets, since they could anyways

not defend their own food pellets at that point. Similarly, if both enemy agents were scared, both their agents would switch to offense and try to eat the enemy food pellets. This allowed them to perform quite well in the tournament, and secured them the third place.

Group 1 had an approach based on Reinforcement Learning. Though it was expected that such approaches could be competitive, their performance in the tournament was not as good as expected. This could be because there was no way to train their agents on the actual tournament map, and they had to train their agents on a smaller map. There was also the problem of the agents not being able to see the entire map, which led to hurdles in the training pipeline. They concluded that RL was not a good approach for this problem.

5 Summary and Conclusions

The purpose of this project was to develop an agent capable of playing the game of Pacman capture the flag, utilizing various artificial intelligence techniques. We explored various methods, including minimax game tree search, Monte Carlo Tree Search (MCTS), Expectimax, and decision-tree-based approaches. The final solution included a modified Minimax, alpha-beta pruning. This, in combination with the heuristics, allowed our agents to perform reasonably well. The modified Minimax was implemented by considering the possible actions of the agent we are controlling (as the maximizing player), and when our agent saw an enemy agent, we considered the possible actions of the enemy agent as well (as the minimizing player). Further, we also implemented role switching between the agents, which allowed them to switch roles based on the state of the game, and also to switch roles if one of the agents was killed.

During the pre-final stage, the agents showed good performance, but we still had some bugs that caused them to perform poorly in some situations. A bug that affected our performance greatly was that sometimes our offensive agent thought it was better to sit next to a food pellet than to eat it. This was dealt with by changing the heuristics and adidng a penalty for staying in the same place.

Between the pre-final and final tournaments, we fixed the bugs in our agents, and also implemented the role switching mechanism described in Section 3.2. The role switching mechanism performed well in the pre-final stage maps, however, in the finals the map size was increased and the agents were not able to switch roles as seamlessly, often leaving gaps in our defense.

Despite improvements made to the agents, the agents' performance suffered in the final tournament due to these challenges. It is believed that testings on larger maps and more time spent tuning the parameters would have improved the performance of the agents.

6 Future Work

In this section we will discuss the improvements that can be made to the suggested solution. These improvements are based on the observations made during the development of the solution and during the finals as well as changes the group would have made if more time was available.

During the development of the solution, we observed that the offensive agents occasionally became trapped in local extrema of the value function. For instance, they would sit next to a food particle

without consuming it until an enemy agent approached, forcing them into scramble mode. To avoid this, we added an increasing tendency to return to our own side for each time step in which the chosen action was to stay in the same place. This helped the agent get out of local extrema in a couple of moves, in most cases. However this is not a perfect solution, as we risk getting stuck in a position within a chokehold in the enemy territory. This roots to the heuristic design we have chosen and in order to improve upon this a in depth analysis of the different heuristics is required.

In the finals we observed a peculiar behavior of the offensive agent; the enemy ghost was chasing our agent, but at 5 distances away, our agent moved in a loop following the exact same route multiple times. This suggests that the heuristics should include a time variant factor in order to avoid loops.

In the development of the role switching mechanism, we implemented it analysing the original map sizes from the pre-finals. As a result of this we observe that there were times when our field was unprotected and the enemy agents could easily pass through. This is when an offensive agent is killed and the role switching mechanism is triggered. At that point, the defensive agent becomes the offensive agent and starts advancing into the enemy territory. As a result of this behavior, the now defensive agent is offset from its optimal guarding position, as the agent was just respawned, and the result is our territory being unguarded. The improvement our group suggest for this is to not trigger the switch until the respawned agent has had the time to move towards the middle of our half plane.

References

[CHH02] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[GS11] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go". In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2011.03.007. URL: https://www.sciencedirect.com/science/article/pii/S000437021100052X.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. New York, NY: Springer, 2009. ISBN: 978-0-387-84857-0.

[NM44] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton, NJ: Princeton University Press, 1944.

[Riv87] Ronald L. Rivest. "Game tree searching by min/max approximation". In: *Artificial Intelligence* 34.1 (1987), pp. 77–96. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(87)90004-X. URL: https://www.sciencedirect.com/science/article/pii/000437028790004X.

[UC 15] UC Berkeley. *Contest: Pacman Capture the Flag*. http://ai.berkeley.edu/contest.html. 2015.