MGAIA Assignment 2: Multi Agent Pacman Capture the Flag

Federico Cucinotta ( ), Kevin Bretz ( ), Laurens Grote ( )

March 30, 2025

Contents

Abstract

This paper introduces our approach to the Multi-Agent Pacman Capture the Flag assignment, where the objective is to develop intelligent agents capable of outperforming opponents in a strategic game of Pacman. We explore two distinct agent designs: a heuristic agent, and a MCTS (Monte Carlo Tree Search) agent both leveraging perfect information. The heuristic agent aims to maximize its score by following smart strategies such as collecting food, returning to score, defending its territory, and evading or hunting enemy agents. In contrast, the MCTS agent utilizes a tree-based search to explore potential game states, balancing exploration and exploitation to discover optimal actions based on simulated outcomes. This paper talks the concepts, implementation of the heuristic agent, along with the MCTS design, assumptions, selected hyperparameters, and a description of the UCB1 formula for enhancement. Finally, it concludes with a comparison of the agent's strengths and weaknesses, as well as possibilities for improvement.

1 Heuristic Agent

The first task is to implement a heuristic agent, which evaluates all possible immediate actions and makes a decision based on the calculated values of these actions. This is all performed with the assumption that we have "perfect information", meaning that the agents know everything about the environment, including the position of enemies and the structure of the map.

1.1 Concept

The heuristic agent is based on the baseline implementation provided by [1]. For every possible action a value gets defined, and the agent takes the action with the highest value. At the very least, the agent should perform following behaviors:

- Collect food on the enemy half.
- Go back to its own half with collected food to score.
- Defend its own half by eating enemy Pacmen.
- Run away from enemies as Pacman in the enemy half.
- Run away from enemy Pacmen when frightened.
- Hunt down frightened enemies.

To achieve this goal, a distinction between *features* and *weights* is defined. *Features* are properties of an agent in the environment that are specific to the current game state (must be re-evaluated after each step), for example the distance to the closest food or the amount of food that the agent is currently carrying. By optimally defining our features and multiplying them with a pre-defined set of parametric *weights* it is possible to maximize the quality of the chosen action.

Our goal is to develop an agent that utilizes clever strategies implemented purely through the use of heuristics. However, a heuristic agent only takes into consideration the value for

each action immediately following the current game state. Our expectation is that, if we can build a competitively performing heuristic agent, we should be able to apply those heuristics to an MCTS agent that can determine values for actions many steps in the future and therefore achieve even greater results.

1.2 Implementation

The main variation of our implementation from the provided baseline agent is that we do not define dedicated subclasses for our heuristic base agent. Although we do still define certain "modes" for our agents (Offense, Defense, and Run), the mode of an agent is re-evaluated by the game state at every step, rather than being a fixed agent class. The reason for this is that the behavior of the agents is more dynamic, allowing them to switch between attack and defense based on the current state of the game. This approach opens up more possibilities to implement more complex strategies, instead of both agents performing the same actions by being locked into a single static mode for the whole game.

Another adaptation we made to the baseline was including dynamic weights. In most implementations, the weights are defined as static. However, this turned out to be far less practical than expected, as the features and weights need to be well thought out and defined, and the effect of their interaction becomes less and less intuitive the more heuristics we want to use. We found that using some weights that are dependent on the game state makes the overall logic easier to follow and work with.

The heuristics were chosen based on the win-strategy we came up with: Our agent stays in attack mode until we are in the lead, then fully focuses on defense. While in defense mode, it tries to stop the enemy team from gathering any food by protecting the food pellet that is closest to the enemy. Our agent returns back to it's half after every single food it collects. This is done to try and get a lead as fast as possible, as even a lead of just 1 point will win the game if we can stop the enemy from getting any points. When both agents are in defense mode, they identify which food the enemies are closest to and go stand at that position. In this way, our agents prematurely move in the direction of the food that the enemies are going for, and when they enter our half we are in a far better position to stop them from reaching our food.

2 MCTS

2.1 Vanilla MCTS Agent Implementation

After implementing the heuristics agent, we turned to the MCTS implementation. For this, the `MCTSNode` class is the building block of the MCTS tree. Each node stores the game state, parent node, action taken to reach the node, child nodes, visit count, and win count. The `select_child` method selects the most promising child node using the UCB1 formula, balancing exploration and exploitation [2]. This method is called inside the `choose_action()` method. The `add_child()` method creates a new child node by applying an untried action to the current node's game state. The update method then increments the visit count and updates the win count based on the simulation result.

The `chooseAction()` method is what guides the MCTS process. It initializes the root node with the current game state and then iteratively performs the four MCTS steps: selection, expansion, simulation, and backpropagation [3]. During selection, it traverses the tree from the root node, choosing the best child node at each step until it reaches a leaf node (a node with untried actions). During expansion, it creates a new child node by applying a random untried action to the leaf node's game state.

During the simulation phase, we tried two different approaches: the basic version performs a rollout from the new child node's game state until a terminal state is reached by selecting actions randomly. This means that it does not make use of a rollout policy. The second, slightly more advanced version, instead adds heuristics during the simulation phase, to better mimic how two human players would play and think about the game. The heuristics are based on the domain knowledge of the game and it helps to guide the simulation. During backpropagation, it updates the visit counts and win values of all nodes along the path from the root node to the new child node. After a certain number of iterations or a time limit is reached, the agent selects the action associated with the child of the root node that has the highest visit count.

2.2 Assumptions in Vanilla MCTS

- **Discrete Action Space:** Vanilla MCTS assumes a discrete set of actions available in each state. This holds true in Pacman, where actions are typically represented as directions (North, South, East, West, Stop).
- **Deterministic Transitions:** The standard MCTS algorithm assumes that taking an action in a given state leads to a single, predictable next state. This is true for the Pacman game, where half-states are not allowed.
- **Reward at the End of Episode:** Vanilla MCTS often assumes that rewards are only received at the end of a game or episode (terminal state) [4].
- **Perfect Information:** The algorithm assumes that the agent has access to the complete state of the game.

2.3 MCTS Hyperparameters

Our MCTS implementation uses three main hyperparameters that directly affect different aspects of the search process, namely, exploration weight, simulation time, and the max depth of the simulation. The exploration_weight (set to a default value of 0.71) is used in the UCB1 formula to balance exploration versus exploitation during node selection. A higher value encourages more exploration of less-visited nodes, while a lower value favors exploiting nodes that have already shown good results. Given our unique time-constraint of one second we chose to make it quite exploitative (<0.5) such that it would not try too many paths that are clearly not optimal.

Second, the simulation_time (0.9 seconds) determines how long the MCTS algorithm can run before it must return an action. Increasing this value allows for a higher number of unique simulations (breadth of the tree) as well as deeper simulations (height of the tree). Ultimately,

more time will lead to better decisions but requires more computational resources, while decreasing it leads to faster but potentially less optimal decisions.

Lastly, the `max_depth` limits how far each simulation can look ahead in the game tree. When set to a low value like 5 or 10, this parameter prevents simulations from running too long and allows the algorithm to complete more simulations within the given time limit. A larger `max_depth` allows the algorithm to see further into the future and potentially make better long-term decisions, but at the cost of longer simulation times and thus fewer total simulations. Conversely, a smaller `max_depth` means more simulations can be completed, giving better statistical confidence in the results of shallower searches.

2.3.1 Vanilla MCTS Enhancements

The UCB1 (Upper Confidence Bound 1) formula is an exploration vs. exploitation strategy that aims to improve upon the vanilla MCTS algorithm. It is used in the tree policy to guide the selection phase of MCTS. UCB1 balances exploration and exploitation by selecting actions based on a combination of their average reward and an exploration bonus that diminishes as the action is sampled more frequently. This encourages the algorithm to explore less-visited nodes. In contrast to vanilla MCTS, it provides an expected cumulative regret bound against the best single-arm strategy for a fixed horizon. By using UCB1, the MCTS improves its efficiency by guiding the search towards more promising areas of the game tree and avoiding premature convergence on sub-optimal actions [5].

3 Results and Discussions

The heuristic-based agent (`baselineUpgrade.py`) performed really well during our testing. We believe that this is because in a heuristic-based agent, domain knowledge translates directly into behavior through feature weights, allowing for intuitive mappings. For example, it is possible to set a negative value of, say, `-10` to 'invaderDistance' to prioritize defensive behaviors. This creates an immediate feedback loop where changes to heuristics show direct effects in agent behavior.

However, even though it is harder to implement, MCTS offers greater long-term performance potential as it is a smarter and more advanced algorithm. It naturally balances exploration of new strategies with exploitation of known good moves through the UCB1 formula with its exploration weight parameter. Unlike reactive heuristics, MCTS explicitly models possible future states, allowing it to discover multi-step strategies that can outsmart a purely reactive agent. With sufficient simulation time, MCTS can adapt to novel situations by exploring many potential futures rather than relying on pre-defined patterns, and its performance typically scales with increased computation time whereas heuristic approaches plateau once their evaluation function is fixed [6].

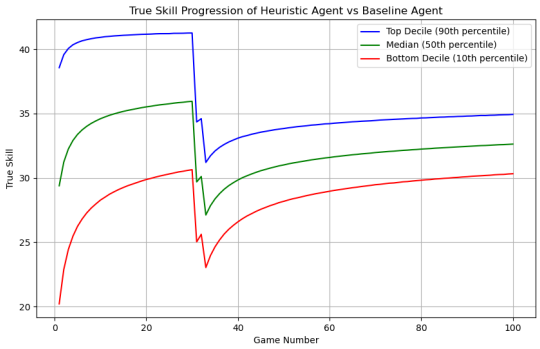

Figure 1: True Skill progression of Heuristic Agent vs Baseline Agent

3.1 TrueSkill

To get a common comparator base *TrueSkill* is used. *TrueSkill* was originally developed by Microsoft to compare *skill* of different players. It is a more advanced version of Elo, a scoring mechanism that the mathematician Arpad Elo invented to compare different chess players. The basic difference is that TrueSkill can also handles more than two players in a game. It computes two values for each player [7]:

- Skill (μ): skill level of the player.
- Confidence (σ): indicates how sure the system is about the players skill. In practice this corresponds to the standard deviation.

To determine the player's skill, a series of games are observed. The output of the games are used to update the player's score. The more often a player is winning, the higher the skill level gets and also the confidence. To implement this we made a custom script (`rating.py`) using the `trueskill` library the python provides. The plots are then made using `plotlib`. TrueSkill also tries to take random breakouts into account and represent them in confidence.

3.2 Comparisons

3.2.1 Heuristic Agent vs Baseline Agent

Our heuristic agent strongly outperforms the baseline. Out of 100 games, it won 98 times. This dominance is reflected in the true skill progression seen in figure 1, which reaches a value of 33 after 100 games.

3.2.2 Heuristic Agent Mirror Matchup

When playing against itself, the playing field is a lot more even, which is portrayed in the graph in figure 2. After 100 games, the red agent won a total of 35 games, while the blue agent won 58 games, and 7 games ended in a draw. This bias towards the blue team can be explained through the random designation of a starting team. For the majority of games, the first team to obtain a lead wins, as the defensive heuristics of this agent are superior to its offense, so the

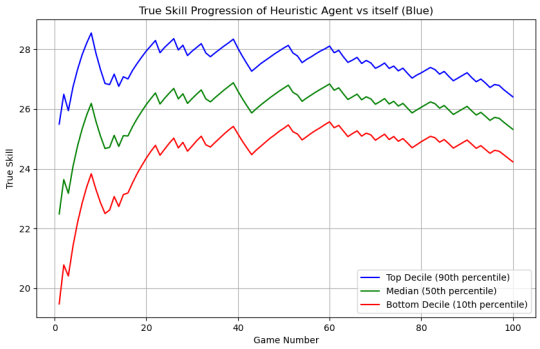

Figure 2: True Skill progression of Heuristic Agent against itself

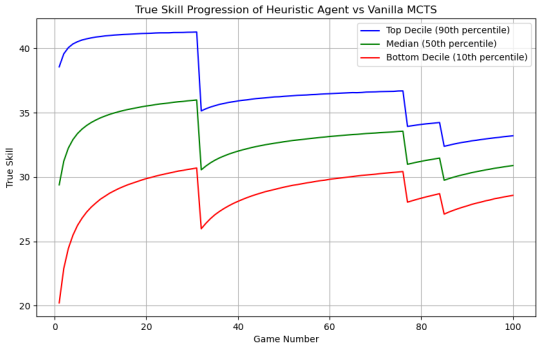

Figure 3: True Skill progression of Heuristic Agent vs Vanilla MCTS

starting team usually has an advantage. If this is the case, further increasing the amount of games should average out this bias. At the end of the 100 games, the blue team achieves a true skill rating of 25.32, while the red team achieves one of 24.68. This further proves the evenness of this agent.

3.2.3 Heuristic Agent vs Vanilla MCTS

Matched up against an agent that uses a vanilla MCTS approach, our heuristic agent once again strongly outperforms the opposition. Out of 100 games, the heuristic agent wins a total of 97 games, while only losing twice and drawing once. The true skill progression can be seen in figure 3. After completing all games, our agent has a true skill score of 31.

3.3 Challenges and Insights

As can be seen from the results, it was much easier for us to create a strong heuristic agent compared an MCTS one. We struggled to find the causes of this, as we knew that an MCTS implementation has clear advantages over a heuristic-based one. This led to us gaining an important insight about the key complementary relationship between the simulation and evaluation functions.

The two functions must use the same heuristics to effectively guide the agent, otherwise they work against each other. Because of the 1 second time constraint, it is not possible to simulate deep into future states. Therefore the terminal evaluation function must meaningfully evaluate partially-completed plans and recognize progress toward goals rather than final outcomes. Accordingly, the rollout policy and must optimize for the same objectives. If this is not done, the simulation policy would get low scores even if it ends up in good states. For example, if the rollout policy pursued food collection as its top priority, while our evaluation instead rewarded capsules as its top priority, there would be a mismatch in objectives and the algorithm would receive contradictory signals.

The roll out policy guides the agent through plausible action sequences while the terminal evaluation assigns value to the resulting state of the simulation. In our implementation, both of these functions make use of domain knowledge to generate better results. The rollout policy goal is to repeatedly perform the best moves till the end of a simulation and try to end up in the best possible state. The evaluation function then evaluates this state with a normalized reward, and couples it with the action currently being added as a node in the MCTS tree.

4 Future Work

4.1 Unfinished heuristic ideas

Due to time constraints, we were not able to fully implement all the logic functions into our heuristic agent that we would have liked to. However, we have come up with a few theoretical heuristics that would improve performance if implemented correctly, and will briefly list these for potential future adaptations.

1. **Improved routing when in offense mode:** The current agent's routing when in offense mode does not do a great job at avoiding enemies. Although an attempt has been made at considering the enemies positions when invading, it does not always choose the best path to it's destination, often walking into enemies. Additionally, the routing needs to be improved for returning to our half after we collect food. Right now it chooses the fastest path without considering enemy positions.

2. **More intelligent defense:** The defensive logic of our heuristic agent is more effective than the offense, at least in a mirror match-up. This is proven by observing that the team that gets the lead at the start of a game usually ends up winning. Comebacks do happen occasionally, but the team that gets the advantage at the start of the game tends to remain in the lead for the entire game. Nevertheless, one major defensive improvement that we had started attempting to implement and that can still be found in the code but does not currently work as intended would be to focus each agent on a different enemy when both of our agents are in defense mode. Ideally, they would be able to switch which enemy they are focusing on based on the current game state, allowing for a very dynamic defense.

3. **Better strategies:** Another idea that can still be found in our code but does not function correctly is for one agent to wait for it's teammate before an attack. The first agent should

not start it's attack until it's teammate is in close proximity on the horizontal axis, but ideally also at least a certain distance away on the vertical axis. Strategically, this would play out like a synchronized attack coming from different sides of the center, making it considerably harder to defend.

4. **Run mode:** When attacking in the enemy half, if an enemy comes within a certain distance from our agent it enters a "Run" mode. In this mode, it no longer cares about food and solely focuses on getting away from it's pursuer. When this happens, and if we are not currently winning, the agent's teammate should be switched into attack mode, basically using the escaping agent as a distraction to hopefully be able to infiltrate the enemy half and collect more food. Ideally, the escaping agent would safely find it's way back into it's own half and go into defense mode. When the new attacking agent encounters an enemy, it too gets set to run mode, essentially cycling through multiple phases of attacking, running, and defending for each agent, until hopefully a lead is established and both agents fully commit to defense.

5. **Capsule logic:** Capsules are currently being completely ignored by our agent. An improved version would implement heuristics to decide wether to attempt to go for the capsule, as well as new behavioral logic for when a capsule is consumed. If the maze distance to the capsule from our agent is shorter than the maze distance to the capsule from all enemies, our agent should fully commit to capturing the capsule. After this, it should start a timer and collect as much food as possible while still returning to its side right as the timer runs out. This is essentially a series of actions that guarantees a certain amount of food will be obtained if the game state fulfills this single, previously mentioned condition. Analogous to this, the agent also needs defensive logic for making sure that this condition is never reached by the enemies agents, as well as logic for in case the enemies do somehow reach our capsule.

4.2 Further MCTS improvements

4.2.1 Parallelization

Parallelization can speed up the process or lead to better results in the same time. The execution of the simulations are independent from each other. Especially in vanilla MCTS the actions are taken randomly, and this process can also be separated into threads. Splitting the task in separate threads makes use of multi-threading, which is supported by almost every hardware. Once a trajectory was executed and a reward calculated, the results can be propagated backward by a central coordinating controller. [3]

4.2.2 Neural Networks

Neural networks can help develop a better policy. They can support the selection mechanism as well as the roll-out by providing better action corresponding to states, which leads to better results. Neural networks were explicitly excluded for this assignment, so this topic is not investigated further.

5 Final Thoughts

This project owns two distinct approaches to developing intelligent agents for the Multi-Agent Pacman Capture the Flag game: a heuristic-based agent and a Monte Carlo Tree Search (MCTS) agent. The heuristic agent performed well in quick decision-making scenarios, providing domain knowledge and predefined strategies. However, it only takes into account the best action in a given state, and not states that are further away. The MCTS agent showed potential for long-term strategic planning by simulating future game states, but faced challenges due to computational constraints and the need to synchronize the simulation and evaluation functions. Both approaches have their strengths. Heuristic agents can be effective for quick responses but struggle with more complex situations, while MCTS offers flexibility and depth, though at a higher computational cost. Future improvements could include both methods into one agent to combine the strengths of fast decision-making with strategic foresight.

List of Acronyms

MCTS Monte Carlo Tree Search 1

UCB Upper Confidence Boundary

References

[1] C. Shelton, "cshelton/pacman-ctf," Mar. 2025, original-date: 2020-06-23T19:57:58Z. [Online]. Available: https://github.com/cshelton/pacman-ctf

[2] "MCTS.ai — what is mcts?" https://mcts.ai/about/index.html.

[3] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte Carlo Tree Search: A review of recent modifications and applications," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 2497–2562, Mar. 2023, https://doi.org/10.1007/s10462-022-10228-y.

[4] T. Miller, "Github — monte-carlo tree search," https://gibberblot.github.io/rl-notes/single-agent/mcts.html.

[5] D. Jayakody, "Monte carlo tree search - a quick introduction (with code)," https://dilithjay.com/blog/monte-carlo-tree-search.

[6] H. Baier and M. H. M. Winands, "Time management for monte carlo tree search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 301–314, 2016.

[7] "TrueSkill — trueskill 0.4.5 documentation," https://trueskill.org/.