Team

Anca-Mihaela Matei:	s4004507
Giulia Rivetti:	s4026543
Evan Meltz:	s3817911
Kacper Kadziolka:	s4115945

Pac-Man: Learn to Play or Play to Learn? Monte Carlo Tree Search and A* Search in Action

Modern Game AI Algorithms - Assignment 2

1 Introduction

This report presents the implications of using two different algorithms - Monte Carlo Tree Search (MCTS) and the A* search algorithm - within the domain of game-based artificial intelligence (AI). It elucidates how these two approaches significantly enhance the decision-making paradigm of the AI agents and showcase in parallel their effectiveness compared to other algorithms from literature.
Among the games in which this technique was employed, Pac-Man stands out as a classic maze chase video game, challenging players to navigate complex environments while being pursued by ghosts. The goal of the game is a simple one: to consume all the pellets placed in the opponent's territory, all while evading being eaten by ghosts and striving to accomplish this task before the opponent can achieve the same goal or before the time budget is terminated.
This report is divided into four main sections and conclusions. Firstly, in section 2 we conduct a comprehensive literature research aimed at highlighting a variety of techniques employed to tackle the Pac-Man game. This examination extends beyond the methods we experimented with, offering a broader perspective on the number of strategies applied in navigating the Pac-Man's challenges. Then, in the section 3 an in-depth analysis is done into the game itself along with its environment and agents, as well as the assumptions and simplifications made. Subsequently, sections 4 and 5 are dedicated to explain the two main algorithms we investigated: MCTS and A* search algorithm. To ensure a balanced analysis, we begin with a foundational overview of each algorithm's theory, followed by an examination of its integration into the game's strategy. This involves the creation of two distinct agents: one focusing on offensive manoeuvres and the other on defensive tactics. The report culminates in a conclusion section (6) that summarizes our findings and insights.

2 Literature Research

The literature on AI in games, particularly using MCTS, is both rich and diverse. MCTS' application to arcade-style games like Pac-Man presents unique challenges as well as distinctive opportunities for innovation. In this section, we will present a few of the key contributions and findings from existing research on MCTS and its application to game AI. Moreover, in order to present a trustworthy analysis, we will also delve into the overall details of some other algorithms used for implementing the Pac-Man agent.

The research area ventures into a more complex environment; in [5] the authors introduced several innovations to adapt the general MCTS in MS Pac-Man, detailing four key enhancements to traditional MCTS approaches: implementing a variable-depth search tree, tailoring simulation strategies for both Pac-Man and the ghosts, incorporating long-term goals into the scoring function, and reusing the search tree across multiple moves with a decay factor. This research demonstrates the feasibility and enhanced efficiency of MCTS in navigating the dynamic and unpredictable environment of MS Pac-Man, offering valuable insights for applying AI in real-time game settings. These innovations could indeed prove highly promising for our game: "Pac-Man Capture the flag", enhancing strategic depth and adaptability in complex environments.
Another approach applied in MS Pac-Man has been presented in [3]. The study showcases a technique to improve the agent's survival by employing the UCT (Upper Confidence bounds applied to trees) algorithm to evade ghost "pincer" moves. The idea behind UCT is: it conducts a multitude of searches among possible moves, weighting the mean reward from random simulations against the search frequency of each node. The comparative performance analysis shows that this approach significantly enhances survival rates over existing methods, demonstrating its effectiveness.
An overview of various AI algorithms implemented for Pac-Man is described in [4]. The authors implemented a suite of AI search algorithms, including Depth-First Search, Breadth-First Search, A* Search, and Uniform Cost Search. Beyond individual agent strategies, the study also delved into multi-agent algorithms like the Reflex Agent, Minimax Agent, and Alpha-Beta Agent, which are designed to adapt to environmental conditions and evade ghosts effectively. In addition to the MCTS agent, we developed a complementary approach utilizing the heuristic A* search algorithm. A* is a variant of best-first search algorithms frequently employed in path-finding problems due to its efficiency and optimality properties [8].
Moreover, we investigated other strategies that provide another view of the problem. In [6] the Deep Q Network (DQN) Reinforcement Learning algorithm was employed. The key idea was training the agents on game's raw pixel images and using action lists as learning data. The results of the experiments underscored the agents' successful learning and real-time game participation, highlighting their ability to adapt swiftly to varying opponents and maps thanks to knowledge gained from previous training sessions.

3 Analysis of Pac-Man Capture the Flag

Before considering a solution, an in-depth look into the game and the environment in which it is played was conducted. This game is a variant of the Pac-Man game but modified to be a version that involves two teams competing, each consisting of two agents (which control both Pac-Man and the ghosts) on opposite sides of the environment.

- **Objectives:** In order for a team to win, its Pac-Man must eat more pellets than the opposing team and attempt to prevent them from eating the pellets on their own side.
- **Environment:** The map in which the game is played is a maze-like environment with walls to restrict movement, power pellets which power up an agent by turning enemies into a scared state, as well as the flags (food) that need to be captured.

- **Agents:** The agents begin as the ghost characters on their own side of the map. They have the ability to move around the map, and when passing over to the enemy's side, they become Pac-Man who has the ability to eat the pellets as well as the power-ups. However, the agent will be killed and returned to its original side if it is attacked by an enemy agent in a ghost state while it is in a Pac-Man state. Lastly, when an agent in the Pac-Man state eats a power-up, all enemy agents currently in a ghost state will go into a scared state, in which the Pac-Man agent can kill them, forcing them to respawn from the start.

3.1 Assumptions and Simplifications

To effectively apply the MCTS algorithm to "Pac-Man Capture" the Flag, we must make several assumptions and simplifications:

- **Assumptions:**
  1. Agents have perfect knowledge of the game state, including the exact locations of all agents, walls, pellets, and power-ups at any given time.
  2. The decision-making process does not account for the dynamic nature of opponents' strategies, assuming that they follow predictable patterns.
- **Simplifications:**
  1. The game's state space is reduced by focusing on critical elements such as the positions of agents, the number of remaining pellets, and the status of power-ups. This excludes less impactful variables like the specific paths taken by agents.
  2. Real-time decision-making constraints are managed by limiting the depth of the MCTS search tree and using heuristics to evaluate game states beyond the depth limit.

By adopting these assumptions and simplifications, we aim to strike a balance between computational feasibility and the preservation of strategic depth in applying MCTS to the game.

4 Monte-Carlo Tree Search Algorithm

4.1 Theory

MCTS is an algorithm used for making decisions within tree-structured combinatorial paces. Within these trees, the nodes represent problem states or configurations, while the edges signify the actions or transitions between one state to the next [7].
The principle of this iterative algorithm is that it explores the space state of the environment to accumulate statistical data on the available decisions within those specific states. The goal is to accumulate rewards by following a series of actions that lead to a sequence of decisions. One important aspect is represented by the way in which the future state is determined. The next state in the environment is decided solely by the current state and the action state.
The iterative process consists of 4 phases which are illustrated in Figure 1 and detailed in the following lines:

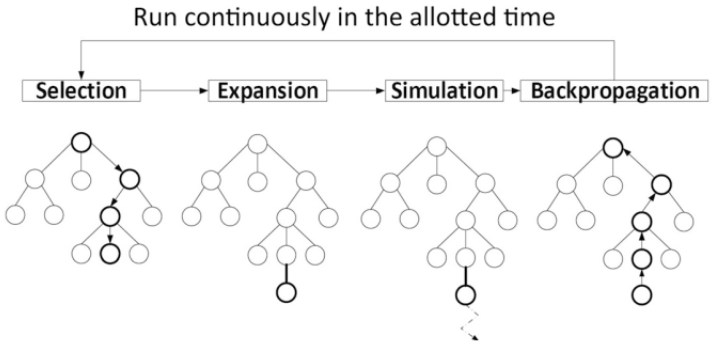

Figure 1: The four phases of Monte Carlo Tree Search Algorithm [7]

1. **Selection:** The first phase of the process involves navigating the tree from the root to a leaf node by selecting child nodes at each level. This phase is guided by a strategy that balances exploration of less visited nodes with exploitation of nodes known to be promising. This strategy is often referred to as the selection policy or as tree policy [7].

2. **Expansion:** Upon reaching a leaf node that hasn't been fully expanded (i.e., the depth of the tree has been reached), the expansion stage adds one or more child nodes to the tree to explore new states. This step represents the exploration of a new action that has not yet been simulated for the current state.

3. **Simulation:** In the third phase, the algorithm simulates a random play from newly added node(s) to the end of the game. The aim of this stage, also known as the "Monte Carlo" stage in [7], [9] or "Play-out" phase in [9], is to generate an outcome based on random moves, which helps to estimate the value of taking the action represented by the node added during the expansion phase.

4. **Backpropagation:** After the simulation, the results are propagated back up the tree. This means updating the nodes along the path from the newly expanded node back to the root with the results of the simulation.

These four stages work in a loop, with each iteration providing more information to refine the tree's representation of possible outcomes, leading to more informed decision-making as the simulation progresses.
Further in our report, we will detail the integration of MCTS theory into our two agents' strategic framework for playing "Pac-Man Capture the Flag".

4.2 Agent Implementation

4.2.1 Offensive Agent

The implementation of the offensive agent within the Pac-Man game is guided by the strategic application of MCTS algorithm to maximize the agent's performance in consuming pellets and evading ghosts. The offensive strategy is also adapted to the dynamic and unpredictable nature of the Pac-Man environment. This section delves into the specific modifications and considerations undertaken to tailor MCTS for an effective offensive agent.

Node Implementation
The core of the MCTS implementation is encapsulated in the *MCTSNode* class, which represents the state of the game at any point in time, including the agent's position, possible actions, and the outcomes of those actions. Each node in the MCTS tree is associated with a game state, an action that led to that state, the node's parent, and any resulting child nodes. The expansion of the nodes is realized based on the methodology detailed in the previous section.
The simulation assesses the agent's progress towards its objective, such as consuming pellets and reaching the border while avoiding enemy ghosts. The action's evaluation prioritize those that lead to closer proximity to the border, as escaping the border is a crucial strategy for the offensive agent. The reward system is designed to penalize lack of progress, encouraging the agent to make strategic moves that advance its position.
Results from the simulation are propagated back at each node, updating the number of visits and win statistics, based on the simulation outcomes. This information is then used to select the next best action to take.
The selection process incorporates the UCT formula [3], balancing between exploitation of known successful paths and exploration of new actions. The amount of exploration is set by modifying the parameter c in equation 1.

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{2 \times \ln N_i}{n_i}} \tag{1}$$

where:
w_i is the number of wins after the i^{th} move,
n_i is the number of simulations for the node after the i^{th} move,
N_i is the number of simulations for the parent node,
c is the exploration parameter, which controls the degree of exploration.

In our experiments, we varied this parameter to identify the optimal one for our game environment. Tests were run with c set to 0 (no exploration), 1, 5, and 10. The experiments demonstrated that a c value of 1 provided the best balance, leading to superior performance of the offensive agent, by wining the game in the time budget. Moreover, the game was won also by the agents which had the exploration parameter set to the other mentioned values. However, in each of the cases the agent won by having more points than its opponent when the time was up, not by returning on its terrain with 18 points. One other noteworthy aspect is that when we used only exploitation in the selection process the agent got stuck in its territory by the opponent's ghost, making impossible for it to advance. Therefore, choosing $c = 1$ seemed the best solution for this task.

Agent Implementation
The class designated for the agent implementation leverages the game's built-in functionalities while introducing the decision-making mechanism based on MCTS.
Upon initialization the agent determines the border positions of its safe zone, a zone that is aimed to be reached after collecting all the 18 pellets necessary for wining the game. The border is determined by identifying the midline of the game map and adjusting it based on the agent's team, focusing its efforts on invading the enemy's side and retreating safely when necessary.
Moreover, an algorithm was implemented to make the agent aware of the possible threats. In this way, the agent can be constantly shrewd, identifying enemy ghosts within a specific radius and adjusting its actions to evade capture.

The core of the agent's offensive strategy lies in its decision-making process, encapsulated in the "chooseAction" method. This method decides whether to employ the MCTS algorithm or a simpler evaluation function based on the game's state, such as the number of remaining food pellets, the agent's current payload, and the proximity of enemy ghosts. When conditions favor a more calculated approach (i.e., when the agent is Pac-Man, carrying significant food or facing nearby threats) the agent initializes an MCTS tree with the current game state as the root. The agent then uses the MCTS algorithm to iteratively explore and evaluate possible moves, eventually selecting the best action based on the outcomes of these simulations. On the other hand, in the scenarios where the agent is not the Pac-Man, it restores to a more simplified evaluation mechanism, maximizing pellets collection or minimizing danger from enemy ghosts.
The overall agent integration algorithm embodies a blend of strategic foresight and reactive adaptability, by employing MCTS. The agent is able to navigate a complex dynamic environment with notable efficiency, achieving a balance between exploiting known successful strategies and exploring new possibilities. Through this tuned trade-off, the agent showcased its ability to make informed decisions that optimize both immediate gains and potential future opportunities.

4.2.2 Defensive Agent

For the implementation of this agent, the focus was on ensuring that it protects its territory from invaders by staying on its own side patrolling the border, and by intercepting invaders as soon as they cross over the border. The defensive strategy of the agent is straightforward and directly programmed, focusing on real-time analysis of the game state to make strategic decisions.
The defensive agent analyzes the game state in order to identify enemy positions, calculate distances, and then determine the most strategic actions so that it can intercept or block any enemy invaders. This process allows the agent to be both reactive and strategic, adjusting its actions based on the current situation. The agent employs direct logic and condition checks for decision-making. It switches between patrolling the border to maintain a robust defense and moving to intercept invaders based on their proximity and the likelihood to cross over and invade.
This balance of immediate reaction to threats and strategic positioning for defense showcases the agent's adaptability to various game states. The simplicity of its strategy, focused on essential defensive tasks, allows for effective territory protection against opponents.

5 A* Search Algorithm

The second agent that we have implemented is based on heuristic search, more specifically on the A* algorithm, which we have decided to use because of its efficiency and since it guarantees to find the optimal path when used with appropriate heuristics [2]. The A* search algorithm is one of the best and most popular techniques used in path-finding and graph traversals; it is a heuristic searching method that aims to find the least-cost path from a given initial node to the goal. In particular, the search executed by the algorithm is based on the following formula:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost of the path to node n and $h(n)$ represents the heuristic estimate of the cost of getting to a goal node from n [1]. This means that the algorithm uses search heuristic as well as the cost to reach the node. To have a better understanding of how this method works and how we have implemented it, we have provided the pseudo-code (see Algorithm 1).

Algorithm 1 A* Search Algorithm Pseudocode

```
Initialize open list, closed list

Place the starting node into the open list
while open list is not empty do
    From the open list, select node n that has the smallest f value ▷ f(n) = g(n) + h(n)
    if n is goal node then
        stop search
    else
        expand node n and generate its successors
        place n into the closed list
        for each successor n' do
            if n' not in open list and n' not in closed list then
                compute evaluation function f(n')
                place n' into the open list
            end if
        end for
    end if
end while
```

5.1 Agent Implementation

5.1.1 Offensive Agent

The implementation of the offensive agent is guided by the A* search algorithm which uses a heuristic function in order to determine what is the best next action. Now, let us analyze how this is done in the case of the offensive agent.
Since the offensive agent can be either a ghost or a Pac-Man depending on which side of the field it is, we have handled these two possible states separately.

Offensive agent as ghost When the offensive agent is a ghost, its main purpose is to go on the opponent's side in order to be able to eat food. This is done by employing the A* algorithm to evaluate successor states, prioritizing actions that minimize the distance to the nearest food pellet. In particular, the algorithm first initialized a priority queue with all the positions that need to be explored. Then it iterates over a loop until the priority queue is empty, and at each iteration it pops the position with the lowest cost (priority) from the queue, which corresponds to the position which is closest to a food pellet. If this position contains food, it means that we have found the goal state and we stop the search. Otherwise, we search among all the successors of this position, namely we look for all the next actions that can be taken from that position and we always look for the one with lowest cost. By following this approach iteratively, the A* search identifies the best path to follow, which corresponds to the one leading to the closets food pellet.

Offensive agent as Pac-Man As already explained, the offensive agent becomes a Pac-Man when it enters the opponent's side of the field. At this point the Pac-Man's behavior is determined based on several factors.
First of all, we have decided to make the Pac-Man retreat to our side of the field as soon as it has eaten a food pellet. Since points are received only when the Pac-Man is able to get to our side of the field, we have deemed as a good strategy trying to collect a single food pellet at a time. Aiming for more food pellets before retreating could potentially result in the Pac-Man being captured and hence the gain of no points.
If instead the Pac-Man has not eaten yet any food pellet when it enters the enemy's area, the agent first makes sure that there is no enemy nearby. This is done by retrieving the positions of the enemy's ghosts which are within 4 steps from our agent and selecting the best action which is closest to a food pellet but which is at least two steps away from the closest opponent's ghost. Moreover, if several good actions are found following this procedure, the algorithm prioritizes the action that does not lead to a dead end, in order to minimize the chance of our agent being caught by the enemy's ghost.
Finally, if the Pac-Man hasn't eaten any food pellet since it has entered the opponen's field and no opponent's ghost is detected nearby, the offensive agent purpose becomes again to find the shortest path that leads to eating the closets food pellet. The action selection procedure in this case follows the same A* search approach employed for the offensive ghost as explained in paragraph 5.1.1.

We have observed that the A* search is an efficient algorithm to guide our offensive agent, however it is not always able to find the optimal action. Therefore, we have also handled the case where the search ends without finding any good action or even when the returned best action is to stop (since the game has a limited amount of time, we do not consider 'Stop' as a good action. In these two cases, the algorithm retrieves the list of all the legal actions, which are all evaluated based on the following criteria: if the offensive agent is currently Pac-Man, actions are evaluated based on their distance to the closets food pellet and the algorithm prioritizes the states leading to eat a food pellet or becoming closer to it; instead, if the offensive agent is a ghost, we prioritize states with fewer food pellets remaining as well as states which are closer to food pellets. After the evaluation part, which takes into account the aforementioned features and their respective weights, we retrieve the best actions, and then the actual action that will be executed is selected randomly among those.
To sum up, the offensive agent's behavior is designed to adapt to various game scenarios and to fulfill its objectives by employing a heuristic approach. We have developed strategies that prioritize actions based on the proximity to food pellets and the avoidance of opponent threats; moreover, our agent demonstrates adaptability by retreating after consuming a single food pellet to mitigate the risk of capture and maximize point accumulation. Additionally, we have implemented a fallback mechanism to handle the case where the A* algorithm fails to find a good solution.

5.1.2 Defensive Agent

The defensive agent integrated into our secondary approach, employing heuristic techniques, closely resembles the defensive agent utilized in the MCTS approach. We were pleased with the efficacy and strategic acumen demonstrated by this agent within the framework of MCTS. It diligently patrols the perimeter of its designated region on the board, actively scanning for potential intruders to intercept.

The primary distinction between the preceding method and the current one lies in its implementation strategy. Rather than directly computing the shortest Maze Distance, we integrate the A* search algorithm enhanced by a heuristic function utilizing Manhattan distances:

$$\text{Manhattan distance} = |x_1 - x_2| + |y_1 - y_2|$$

Consequently, multiple paths to the goal state are systematically assessed, considering their associated costs. Ultimately, the A* search algorithm yields the most optimal and efficient path for the agent to traverse. It is imperative to note that within the sets of actions evaluated by the searching algorithm, priority is exclusively given to defensive actions.

6 Conclusion

In this report, we explored the domain of game-based artificial intelligence, particularly focusing on the game "Pac-Man Capture the Flag". Our investigation was based on utilizing Monte Carlo Tree Search algorithm and the A* search algorithm to yield the best possible performance within the in-game competition.

Our analysis of the task was preceded by a comprehensive literature review, wherein we discussed various techniques previously employed. Notably, we focused extensively on exploring MCTS enhancements proposed by various studies. Additionally, we investigated the application of the A* search algorithm, among others, providing us with a broader perspective on AI algorithms in game environments. The analysis laid a solid groundwork for understanding the game's objectives, enabling us to subsequently formulate assumptions and simplifications to effectively apply MCTS and A* search to the game.

The Monte Carlo Tree Search algorithm, when integrated into offensive and defensive agents, showcased remarkable adaptability and strategic foresight. Both agents leveraging MCTS showed efficiency: the offensive agent excelled in consuming enemy food while evading enemy ghosts and securing victories within the time limit. On the other hand, the defensive agent efficiently patrolled its territory, intercepting invaders with optimal strategies.

In contrast, the A* search algorithm provided a reliable framework for strategic pathfinding, enabling either the offensive or defensive agents to efficiently play against their opponents. By prioritizing particular actions and computing optimal paths to them, the A* algorithm demonstrated its effectiveness in maintaining food consumption and territorial integrity.

In conclusion, both MCTS and A* search algorithms offer valuable contributions to game-based AI, each with its unique strengths and applications. By leveraging these algorithms, AI agents can navigate complex game environments, make informed decisions, and achieve strategic objectives effectively.

References

[1] Heuristic search: A* search. https://ecealptekin.medium.com/heuristic-search-a-search-1e3e41d1802. 22-02-2021.

[2] Informed search algorithms. https://www.javatpoint.com/ai-informed-search-algorithms.

[3] Nozomu Ikehata and Takeshi Ito. Monte-carlo tree search in ms. pac-man. pages 39 – 46, 10 2011.

[4] Swetha Pendem and Sowjanya A. Mary. AI Algorithms for Pacman. *International Journal of Creative Research Thoughts (IJCRT)*, 10:a90–a94, 2022.

[5] Tom Pepels, Mark H. M. Winands, and Marc Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

[6] Rosalina Rosalina, Axel Sengkey, Genta Sahuri, and Rila Mandala. Generating intelligent agent behaviors in multi-agent game ai using deep reinforcement learning algorithm. *International Journal of Advances in Applied Sciences*, 12:396, 12 2023.

[7] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: A review of recent modifications and applications. *CoRR*, abs/2103.04931, 2021.

[8] Wikipedia contributors. A* search algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1214713509, 2024. [Online; accessed 23-March-2024].

[9] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 25–36, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.