Report Assignment 2 DD2380

GROUP 4:3

Vilmer Jonsson Tor Strimbold

2001-06-26 1999-10-06

vilmerj@kth.se torstri@kth.se

May 27, 2024

Abstract

In this report, we present our implementation of an agent able to play the Battlesnake game where two or more agents complete in a last-man-standing format. Implemented in JavaScript using Node.js, our agent utilized the Monte Carlo Tree Search (MCTS) algorithm to enable collaborative and competitive behavior. The results of the project are presented and compared to other implementations for the same problem thereafter we present some extensions to the algorithm which would further improve the performance. This work offers insight into how relatively simple AI can be applied to a zero-sum game and exhibit collaborative and competitive behavior, as well as how the MCTS algorithm can be applied to video-game AI agents.

1 Introduction

The game Snake was first released in 1976 where two players controlled a "snake" which continually grows for every turn, to grow as large as possible before being eliminated by either colliding with the surrounding wall. This project is based on the retro game through the coding challenge provided by `Battlesnake.com`. Several formats are more or less similar to the original game, but the main difference is that snakes now have a *health* property which starts at 100 points and decrements for every in-game turn. When a snake consumes any of the food on the board, the snake grows by one square and their health is restored. This extension is a part of every *battle snake* theme but some of the most popular formats are *standard*, where four snakes battle each other, *duels*, where two snakes face-off, which the similar format to that of the original game, and *royale* where four snakes fight on a board where a "hazardous" zone grows from the edges towards the center, inflicting further reductions to the snakes' health when they pass through the zone.

Thus, the problem underlying the problem is to design an agent that can effectively handle different situations and make decisions regarding short-term versus long-term rewards. It is, for example, not always the best move to eat food to grow or risk an attack against an enemy if it could result in a precarious situation if the attack fails. Furthermore, in this project, teams were tasked with writing code to guide the snake in a 2v2 format which introduces additional parameters to the problem. Through this extension, the snakes' survival chances are increased if they can collaborate to eliminate the opposing team.

1.1 Game & Competition Rules

At the end of the project, a competition was held to determine which of the submitted solutions was the strongest. The matches were run on the standard format which can be summarized as follows:

1. The map consists of an 11 by 11 grid.
2. Two teams compete with two snakes per team.
3. Each snake has a "health" value that starts at 100 but decrements for every in-game turn. If a snake's health is reduced to zero, the snake is eliminated.
4. Every round of food can be spawned on the map, if a snake eats any of the food its length is increased by one and its health is restored to 100.

5. If a snake runs out of bounds or into another snake's body, excluding the head, the snake is eliminated.

6. If two snakes collide head-on, the shortest snake is eliminated. In the case where both snakes are of equal length, both snakes are eliminated.

7. A team wins a match if both of the enemy snakes are eliminated before the own snakes. In the case where both teams' snakes are eliminated simultaneously, the game results in a draw.

Lastly, the competition consisted of two parts, first a round where every team faced off against every other team, and a play-off consisting of the top 8 teams from the previous round.

1.2 Contribution

The work presented in this report is an implementation of a collaborative AI agent that utilizes the Monte Carlo Tree Search (MCTS) algorithm. The algorithm was implemented in Node with JavaScript and was hosted on a server owned by Replit. The presented work aims to further explore the capabilities of the MCTS algorithm and contribute to the field of artificial intelligence. Many modern video games rely on having non-playable characters (NPC) which human players can interact with. These agents need to be able to navigate through and interact with their environment to engage with the player. The algorithms used can however also be employed when running other types of simulations not related to video games. Lastly, the results presented in the report are not only relevant to the techniques implemented in the solution but also compared to the work performed in conjunction with our own. In total 25 teams were tasked with solving the presented problems which resulted in a plethora of different approaches and algorithms at play.

1.3 Outline

The outline of the rest of the report is as follows: section 2 presents the related works concerning our implementation of the problem, the following section 3 gives the reader an overview of our solution and the underlying algorithms, section 4 presents the results of the competition hosted at the end of to protect as well as some online tournaments, our work is lastly discussed and summarized in section 5 and 6 respectively.

2 Related work

[2] introduces the UCB1 algorithm which is a general algorithm for balancing exploration and exploitation. They prove that simple and efficient policies can achieve the optimal logarithmic regret uniformly over time. The algorithm is general and applied to the Multi-Armed Bandit problem in the study. The Multi-Armed Bandit problem is a problem where a gambler is faced with a row of slot machines, each with an unknown probability of providing a reward. The gambler has to decide which slot machine to play to maximize the reward. The UCB algorithm can then be used to choose the slot machine with the highest potential reward.

One of the possible factors that can be improved in the MCTS algorithm is the selection policy. The selection policy is the policy that decides which node to explore next. [3] introduces the *Upper Confidence Bound for Trees (UCT)* algorithm, which is a selection policy that balances exploration and exploitation. The UCT algorithm is based on the Upper Confidence Bound (UCB) algorithm and is very similar to it. The goal of the UCT algorithm is to select the node that has the highest potential for being the best node thus ignoring nodes that have already been explored and found to likely be suboptimal.

Several possible enhancements can be made to an MCTS, and different problems allow for different enhancements. However, some enhancements are general-purpose. In [4] the authors introduce a framework, *Information Capture And ReUse Strategy (ICARUS)* used for evaluating and combining the enhancements.

AlphaGo was the first AI to beat a human professional Go player, the game of Go has a large state space and is considered more complex than for example chess. The AI was developed by DeepMind and used a combination of deep neural networks and MCTS. The reason for using MCTS was that it is a good algorithm for games with a large state space and where the state space is not fully known.

An interesting event in the world of chess programming occurred in late 2017. At the time Stockfish was firmly considered as the strongest chess engine, outperforming human players and engines alike. This view was however questioned when the engine was pitted against DeepMind's project AlphaZero. [1]

DeepMind had in the previous year achieved an impressive feat in a similar board game named Go through their engine AlphaGo Zero, which was able to defeat the current world champion Lee Zedol. What separated AlphaGo Zero from previous game-playing systems was the fact that the program was only given the rules of the game and learned by playing against itself. The

Deep Mind team now aimed to develop a more generic program able to play not only go but also chess and shogi. [5]

3 Proposed method

The AI is implemented using the MCTS algorithm. The algorithm is divided into four main steps: selection, expansion, simulation, and backpropagation. Every turn, the algorithm is called and starts to run iteratively for a certain amount of time.

Every node in the tree represents a state in the game. The root node is the current state of the game, and the children of the root node are the possible moves that can be made from the current state. A move is defined as all snakes in a team moving one step in a certain direction. If a move results in a collision, the move is invalid, and the node is not expanded. Thus, the number of children a node can vary, but is at most nine, since each snake can turn left or right, or go straight, and there are at most two snakes in a team.

3.1 Selection

In every iteration, there is a selection phase where the algorithm selects the best child node to explore. The selection starts from the root and then traverses down the tree, picking the best child at each level, until it reaches a leaf node, a node with no children.

The best child is selected based on the Upper Confidence Bound (UCB) formula. The formula is defined as:

$$UCB = \text{score}_i + C\sqrt{\frac{\ln N}{n_i}} \tag{1}$$

Where score_i is the average score of the node, N is the total times the parent node has been visited, n_i is the number of times the node has been visited, and C is a constant that controls the exploration-exploitation trade-off. The constant C was set to $\sqrt{2}$ in this implementation, but no empirical testing was done to determine the optimal value. If the node has not been visited, the UCB value is set to infinity to ensure that the node is selected at least once if its parent node has been visited.

3.2 Expansion

The expansion phase is where the algorithm creates new child nodes for a leaf node. To create the child nodes, every possible move from the current state is generated. Generating a state is done by copying the current state and applying a move of one of the team's snakes. If the snake has no valid moves, it will be removed in the new state.

3.3 Simulation and Evaluation

When new children have been added to the leaf node, the best child according to equation 1 is selected. To determine the score of the node, the algorithm simulates the game from the selected child node for 12 or 12.5 rounds, depending on whether the friendly or enemy team makes the first move. The purpose of this is to ensure both teams have done the same number of moves when the simulation ends. The simulation is done by selecting a random *valid* move for each snake in the team. A move is considered valid if it does not result in a collision with a wall or another snake. This is done to make the simulation more realistic while still covering many possible states. There were considerations of for example simulating the snakes targeting food, but this was not implemented due to not wanting to bias the simulation and risk limiting the exploration of alternative, potentially better, moves.

The score of the simulation is based on a heuristic function that evaluates the state of the game at the end of the simulation. The heuristic function is described in equation 2.

$$\begin{aligned} \text{score} = \text{accLengthFriendly} - \text{accLengthEnemy} \\ + 10(\text{accHealthFriendly} - \text{accHealthEnemy}) \end{aligned} \quad (2)$$

Where both the values of length and health is accumulated, which means that it is the total length or health of all snakes in a team. The heuristic function is simple and is designed to decide which of the teams have the most beneficial position at the end of the simulation. Accumulated values are used to give a higher score to the team with the most snakes alive. There were other heuristic functions tested, but when played against each other, the heuristic function described in equation 2 performed the best.

An important attribute of the scoring heuristic is that it is symmetric. This means that the same can be used for both the friendly and enemy teams. When back-propagating the score, the score is negated when back-propagating to the parent node, to ensure that the parent node gets the

score of the enemy team. This is done to ensure that the simulation tries to maximize the score of the enemy team, which is the same as minimizing the score of the friendly team.

3.4 Head-To-Head Collisions

In the early stages of testing the algorithm, it was discovered that the snakes would often collide head-to-head and this would thus heavily impact the outcome of the game since the shorter snake would die. The reason why head-to-head collisions are difficult to model is that the snakes move simultaneously in the game, but in the simulation, the snakes move one after another. This can lead to a situation where one snake moves into a position that another snake also can move to, resulting in a head-to-head collision. To handle this, the algorithm is very safe regarding the friendly snakes. The friendly snakes will never move into a position that would risk a head-to-head collision with a longer snake. However, the algorithm assumes that enemy snakes can take such risks, this leads the algorithm to be more aggressive and try to set up for a head-to-head collision when the enemy snake is shorter.

4 Experimental results

This section presents the experimental results collected at the end of the project through the competition mentioned in section 1.3.

4.1 Competition Results

The largest testing of our solution was through the competition hosted on the 19th and 20th of May containing 26 other teams from the course. The competition consisted of two different parts, first a round where every team played a match against every other team, and secondly a play-off between the top 8 teams. During the first part, each team was responsible for hosting twelve games each, thus we hosted games against twelve other teams and the remaining teams hosted games against us. In this manner, every team was paired against each other in the first round. Furthermore, a win was awarded 3 points, a draw 1 point and a loss did not net any points. Consequently, the eight best teams from this round then played a knockout tournament, where matches consisted of a best-of-three format and the loser was eliminated while the winner progressed to the next round.

Table 1 presents our results from the first round, where our solution performed well beating 19 of the teams while losing the remaining 5 matches.

This resulted in a total of 57 points and a shared first place with five other teams, see Table 2 for reference, and in turn a place in the ensuing knockout stage. The results from the play-offs can be viewed in Figure 1, where we are group 3.

	Wins	Draws	Losses	Points
Home	10	0	2	30
Away	9	0	3	27
Total	**19**	**0**	**5**	**57**

Table 1: Results from the first round of the competition.

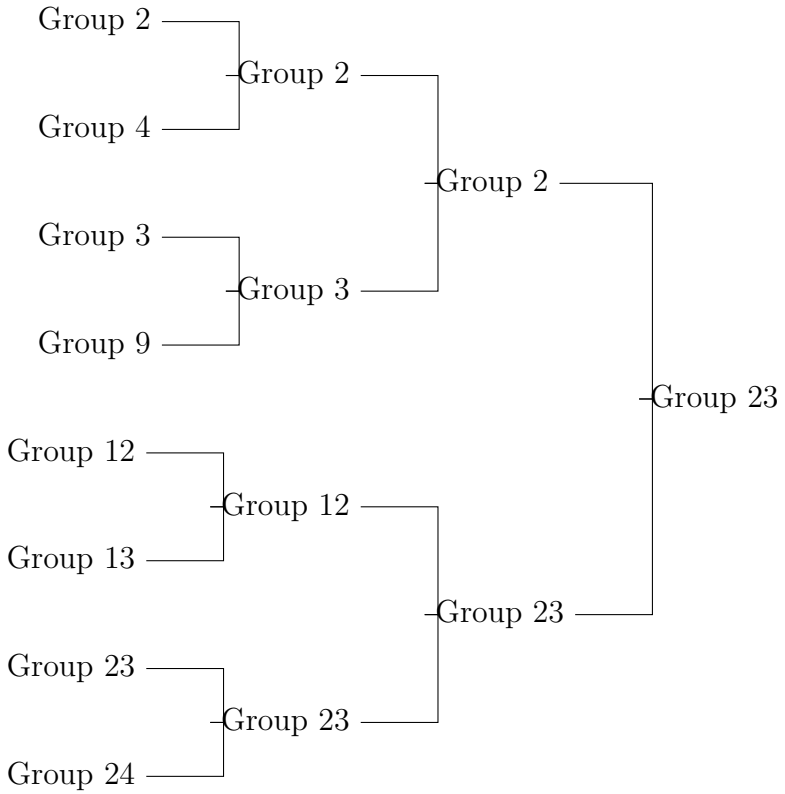

Figure 1: The results from the playoffs.

Our solution differed from most of the top 8 teams where only two other groups employed a tree search algorithm. The first was group 13 which utilized a paranoid Minimax algorithm where each snake saw all the other snakes as opponents, hence the paranoid part, and group 9 used a game tree of depth 2 to determine which move to make. The remaining teams' approaches

were similar to a "rule-based" method where a modified A* algorithm would guide the snake based on different weights and heuristics. One common idea was to use the flood fill algorithm to find dead-ends and reachable grid cells from any position. This can then be used to determine which grid cell a snake should navigate towards and the path to the destination would be determined using A* where different weights, such as proximity to enemy snakes, guide the search.

5 Summary and Conclusions

To summarize, this project implemented an AI capable of playing the Battlesnake game utilizing the MCTS, resulting in a third place in the final competition. Although we are content with the results it is observable that there is room for improvement. The first and perhaps easiest fix that could be made is to fine-tune the different parameters such as the exploration constant in the tree policy. However, these types of improvements can be tedious and not very interesting, thus if we were to redo or extend the project, this part of the implementation would not take priority. Another similar type of improvement would be to make the state generation methods more efficient since these are often the slowest part of the algorithm. With a faster state generation, the tree search would be able to reach lower depths of the game tree, resulting in more qualitative decisions. However, the solution is able to reach a depth of 12-15 consistently, and instead, the quality of the search should perhaps be improved before improving the efficiency.

One way to improve the search has to do with the fact that food is spawned with a certain probability every round in the free squares of the board. Currently, our implementation does not take this into consideration when generating new states during the search. Thus, it might be necessary to include this functionality to reap the benefits of increased depths. Since, if the algorithm reaches a depth of say 30, the current state representation at this depth would be incorrect since no food would have been added to the board, which could lead to misinformed decisions and evaluations being made.

A second aspect in order to improve the search is the evaluation function which as explained earlier was quite simplistic, only calculating the difference in health and length of the two teams. Integrating domain-specific knowledge such as avoiding dead-ends or constricting the space of enemy snakes, could improve the accuracy of the evaluation and in turn the performance of the solution. These types of ideas were present among most of the top-performing teams, which makes us believe that this could indeed lead to an improvement

in performance.

Additional aspects which could be improved are regarding the game tree which is discarded every round only to be rebuilt from scratch. One obvious improvement is to reuse parts of the game tree in the following turns since these have already been explored to a certain extent. Similarly, some states are either identical or very similar, retaining information about one state could then be applied to similar or identical states later in the search which could save computational power and allow the algorithm to increase the depth..

Lastly, AlphaZero showcased an impressive performance against very strong traditional chess engines such as Stockfish, where neural networks were used to guide the selection and evaluation of the game tree, coupled with reinforcement learning to improve from self-play. The time limit of this project did not allow us to include these types of extensions but they describe an interesting improvement potential that would be suitable for further studies.

References

[1] Stockfish - chess engines. `https://www.chess.com/terms/stockfish-chess-engine`. Accessed: 2024-05-27.

[2] Peter Auer, Nicolò Cesa-bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[3] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[4] Edward J. Powley, Peter I. Cowling, and Daniel Whitehouse. Information capture and reuse strategies in monte carlo tree search, with applications to games of hidden information. *Artificial Intelligence*, 217:92–116, 2014.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.