Project 3: Game Tree Search

Luke Runnels

March 28, 2023

1 Introduction

1.1 Game Tree Search

Two player, turn based games are some of the most studied problems in the field of Artificial Intelligence. These game domains are often classified as having ”perfect information” and ”zero-[summation]”, which, according to Russell et al, means that the game domain is full observable and what is good for one player is bad for the other. [2] To fully exploit these properties, an algorithmic technique known as a game tree is imposed over both players for deciding the next action for each player to take.

A traditional approach for implementing game trees involves directly evaluating the game space as utility for deciding the actions for both players. Direct game evaluation, combined with principles of ”perfect information” and ”zero-summation”, is implemented as *minimax* game tree search. Often, the game search space is too large for practical applications, so an optimization for minimax called *α-β pruning* is applied to minimax game tree search.

Unfortunately, it is often difficult apply a direct evaluation of the game space to decide actions for both players in a game. These limitations often stem from the fact that it can be difficult to define strong indicators for players in a game. So, unlike using direct evaluation of a game space, multiple simulations of a game can be applied to try deciding a good action for the players. A common implementation of this idea is known as *Monte Carlo* game tree search.

2 Problem domain used

2.1 Discrete Soccer Game

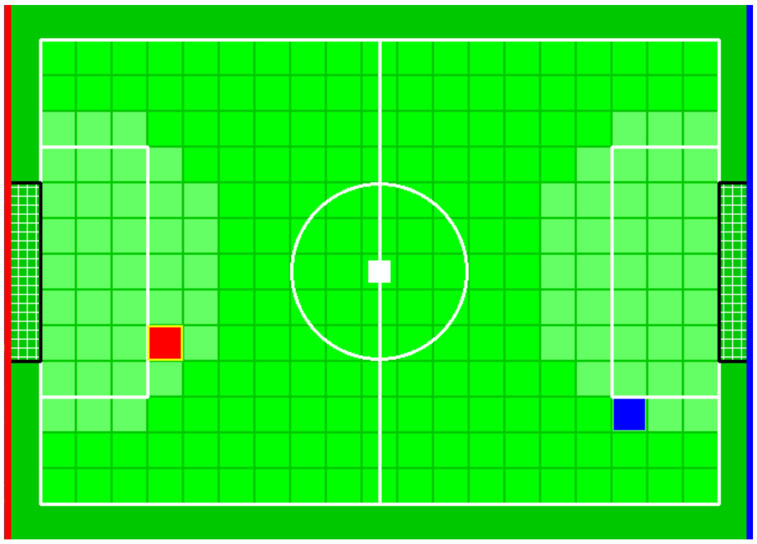

Figure 1: Initial state of the two player, turn-based soccer game

The problem domain was originally written by TU MASTERs group as a domain test for game trees in a turn-based manner [1]. It consists of two agents in a 12x20 field. Each player is placed randomly on either sides of the field, with the soccer ball placed in the middle of the field. Each corresponding player's goal is placed on the opposite side of the field. Both players have two internal states: It has the soccer ball, or it does not have the soccer ball.

If the player does not have the soccer ball, it can move to all eight adjacent positions from its current position. If the player does have the soccer ball, it can move to the adjacent positions in addition to kicking the ball itself. A player can get possession of the soccer ball in two ways. The first way is to move into the initial soccer ball's position in the middle of the field. The second way is to have the other player kick the ball while it is outside of its goal region.

The goal of both players, just like in a regular soccer match, is to kick the soccer ball into its goal on the opposite side of the field. Once the soccer ball has been kicked into either goal, the game will end. However, the game will also end if the current state of the game, which consists of important details such as both player's position, their goal's position, and which player has the ball, is repeated.

3 Algorithm Explanations

3.1 Minimax Tree Search

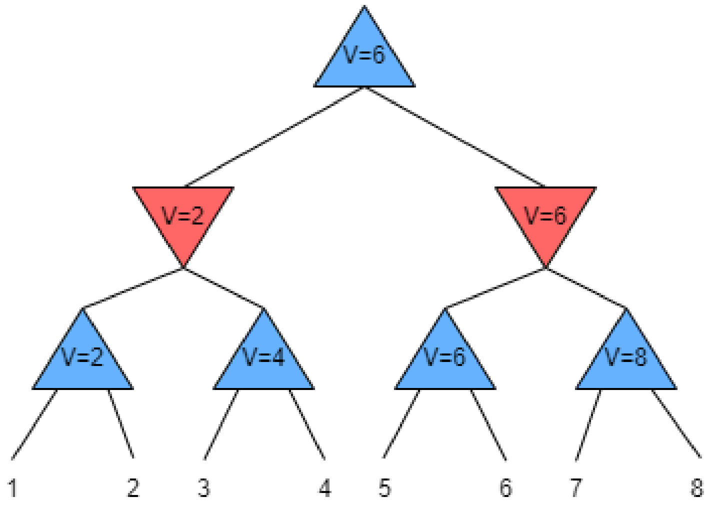

Figure 2: A example of a two depth search minimax game tree

To fully exploit the nature of "perfect information" and "zero-summation" in a two-player game, both players are given a chance to maximize their estimate utility from the state space while also considering the possible actions that may be taken by the other player.

This idea, as described in section 5.2 by Russell et al, can be expressed as the equation [2]

$$Minimax(state) = \begin{cases} Utility(state) & \text{if is-terminal(state)} \\ \max_{a \in actions(state)} Minimax(result(state, a)) & \text{if current\_player's turn} \\ \min_{a \in actions(state)} Minimax(result(state, a)) & \text{if other\_player's turn} \end{cases} \tag{1}$$

To help illustrate this algorithm, figure 2 shows an artificial situation of hierarchical utility values that were calculated using (1). The blue triangles can be thought of as states in the search space where it is the current player's turn, while the red triangles can be thought of as states in the search space where it is the other player's turn. These can be referred to as the maximum layer and the minimum layer in the game tree. At the bottom of the tree, the leaf nodes are interpreted as terminal states with the associated raw utility.

For the maximum layer, the maximum utilities from the previous layer is propagated up the tree. It can be illustrated from the leaf node layer, where 2, 4, 6, and 8 are the maximum utility values. In addition, at the root of the tree, since the red triangles obtained 2 and 6, 6 is propagated to the root of the tree.

Similarly to the maximum layer, for the minimum layer in the game tree, the minimum utilities from the previous layer is propagated up the tree. This is clearly illisturated in figure 2 from 2 and 6 being propagated up the tree.

Given the calculated utilities in figure 2, it is clear that the current player will choose the action that leads down the right subtree.

3.2 α-β Pruning Tree Search

According to Russell et al. in section 5.2.1, it is common for the time complexity of constructing the minimax game tree as $O(b^m)$, where b is the branching factor and m is the maximum depth of the tree [2]. Therefore, it is often impractical to fully expand every state in the game domain.

Fortunately, it is not often necessary to entirely expand the state space. Two parameters can considered along with the minimax algorithm. These two parameters, known as α and β, will bound the utility values that have been observed so far. The parameter α can be thought of as tracking the highest utility choice in the maximum layer, while the parameter β can be thought of as tracking the lowest utility choice in the minimum layer.

To actually prune the search tree, α is compared against the actions taken in the minimum layer, while β is compared against the actions taken in the maximum layer. For both the maximum layer and minimum layer respectively, if $v \geq \beta$ or $v \leq \alpha$ is the case at any time where v is the current utility from acting on the current action, then the game tree will not consider the remaining actions, since there is not a better option.

By implementing this approach into the minimax algorithm, the time complexity can be approximately reduced down to $O(b^{m/2})$, which according to Russell et al, effectively reduces the branching factor of the game to $\sqrt{b}$ [2].

3.3 Monte Carlo Tree Search

Even though α-β pruning can reduce the branching factor of minimax search to $\sqrt{b}$, the effectiveness of minimax game tree search with α-β pruning is fundamentally dependent on designing a good evaluation for the game's state space. It can be a difficult task for many game domains. Thus, an alternative approach to the minimax tree search is Monte Carlo tree search. Instead of using a domain dependent evaluation function as direct influence which actions to take, Monte Carlo tree search will utilize an average utility of all of its actions that have been collected from a set of game simulations.

The algorithm for processing Monte Carlo tree search is divided into four phases: selection, expansion, simulation, and propagation. The algorithm can be represented as a tree structure, where each node contains a unique state in the game with its associated total estimated utility and total number of simulations that contains that particular state.

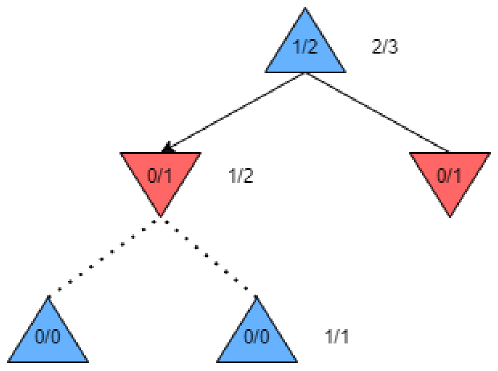

Figure 3: A example of Monte Carlo search tree with three simulations

3.3.1 Selection Phase

At every potential state in the game with the total utility and simulations, there is a problem with deciding which action to act upon. Although there are numerous implementations, one of the most studied implementations is the upper confidence bound applied to trees. A general formula for this selection policy is shown in the following equation.

$$UCB(n) = \frac{Total\_Utility(n)}{Total\_Playouts(n)} + \sqrt{2} \times \sqrt{\frac{log(Total\_Playouts(Parent(n)))}{Total\_Playouts(n)}} \qquad (2)$$

where $Total\_Utility(n)$ represents the total average utility at a particular game state and $Total\_Playouts(n)$ is the number of simulations contained in the state.

The first term in this formula can be interpreted as an exploitation factor for what is known in the game space so far, while the second term can be interpreted as an exploration factor for exploring the state search in earlier simulations. It should be noted that in the exploration term, $\sqrt{2}$ is a tunable constant C. Although any constant is sufficient, Russell et al. makes the argument that, in theory, C should be set to $\sqrt{2}$ [2].

In the example tree shown in figure 3, the current agent selected the left subtree, which is the node that maximized (2) with the current data. The average utility is shown in the numerator of every node, while the number of simulations is shown in the denominator of every node. This rationale that attempts maximize (2) will be applied to every node in the search tree until a leaf node is reached.

3.3.2 Expansion Phase

Once a leaf node is reached, the algorithm will need to consider possible actions to act on in the future. For every possible action from the current state, child nodes will be generated from the new states that are calculated from the acting on the possible actions and appended to the tree. Then, a random child node will be selected to perform a simulation.

A simple illustration in figure 3 shows two child nodes that were generated from the previous state on the left red triangle. Both nodes were initialized with no total utility or simulations. Through random choice, the right-side node was selected for running a simulation.

3.3.3 Simulation Phase

From the selected child node, a simulation of the game is executed from the child node's state. The simulation will repeat until a terminal state is reached. Then, the estimated utility will have calculated for propagation back through the search tree.

In figure 3, the simulation ended with a terminal state in which the starting player at the root of the tree won. This is reflected in the new utility/simulation pair to the right of the current node, which is updated to 1/1.

3.3.4 Propagation Phase

In the last phase, the results from the simulation are propagated up the tree. In the path nodes from the current leaf node to the root of the tree, the average utility and number of simulations will be incremented. It should be noted that different levels of the tree, just like in the minimax game tree, represent turns by both of the players. Therefore, the utility itself is only updated if the corresponding current player in the node to be updated is also the current player in the leaf node that was just simulated.

This phase is clearly illustrated in figure 3. It is assumed that the current player won the game in the last simulation. Thus, the root of the tree and the current leaf node's utility are updated to 2 and 1 respectively. However, for all nodes contained in the path from the leaf node to the root node, the number of simulations are updated.

3.3.5 Deciding on an action

Once all of the phases have been repeated a set number of times, the action that the current player had simulated the most is chosen.

4 Soccer Game Evaluation

In theoretical situations, game tree search could be processed until a terminal state is reached. However, this is often impractical as the search space is exponential. Even then, an evaluation for utility is needed whenever the agents reach a terminal state. It is more common to fix the maximum depth of the game tree for minimax searches or the maximum number of simulations for Monte Carlo searches. In both of these situations, a rational evaluation function needs to be designed to handle non-terminal states.

In the discrete soccer game domain, the terminal states are specified whenever the soccer ball is kicked into the current player's goal or the other player's goal. However, with respect to the current player and the soccer ball, there are three cases of non-terminal state that are considered. The first case is when neither player possesses the ball. The second case is when the other player possesses the ball. The third case is when the current player possesses the ball.

For all three cases, *curr_pos* will refer to the current player's coordinates, *other_pos* will refer to the other's player's coordinates, and *ball_pos* will refer to the ball's coordinates. In addition, higher utility evaluations will *preferred* by the agents, as specified by the game's documentation [1].

4.1 Terminal States

For the terminal states in the game, the evaluation function simply returns weighted constants corresponding to when the current player won or the other player won.

$$Utility(state) = \begin{cases} 25 & \text{if current\_player won} \\ 0 & \text{if other\_player won} \end{cases} \tag{3}$$

The numbers 25 and 0 are constants that make the current player prefer winning the most and losing the least in the entire evaluation function.

4.2 Neither player possesses the ball

In this case, the agents should try moving towards the soccer ball's position. So the utility evaluation for this case was calculated as

$$Utility(state) = \frac{1}{dis(curr\_pos, ball\_pos)} \tag{4}$$

The fraction acts to minimize the euclidean distance between *curr_pos* and *other_pos*, which makes both agents prefer the action that brings it closer to the soccer ball.

4.3 The other player possesses the ball

According to the documentation, if a player kicks the ball while it is outside of the the goal region, or when the other player is placed in-between the current player and the goal, then the other player will get possession of the ball.[1]

To exploit this behavior in the game, the current player will try to position itself in-between the other player and the other player's goal. This is expressed in the following equation

$$Utility(state) = \begin{cases} 1 & \text{if dis(mid\_pos, curr\_pos) == 0} \\ \frac{1}{dis(mid\_pos, curr\_pos)} & \text{if dis(mid\_pos, curr\_pos) != 0} \end{cases} \tag{5}$$

where *mid_pos* is the midpoint position between *other_pos* and the other's player's goal.

It should be noted that there are cases where the midpoint position is zero. In these cases, 1 will be returned from the evaluation function to indicate the highest case of preference whenever the other player possesses the ball.

Similarly to the case where neither player possesses the ball, the fraction works to make both agents prefer being as close to the midpoint as possible.

4.4 The current player possesses the ball

In this case, the current player should attempt to move directly towards its goal. However, it needs to consider the placement of the other player in order to avoid passing the ball to it. This behavior was evaluated using the following equation.

$$Utility(state) = \begin{cases} \frac{10}{dis(curr\_pos, curr\_goal)} & \text{if current\_player is closer} \\ \frac{10}{dis(curr\_pos, curr\_goal)} - \frac{dis(curr\_pos, other\_pos)}{10} & \text{if other\_player is closer} \end{cases} \tag{6}$$

It should be noted that the number 10 acts a preference weight, so the current player will prefer possessing the ball as opposed to not possessing the ball. Whenever the other player is closer to *curr_goal*, then the current player will to try to get closer to *curr_goal* while moving away from the other player. Otherwise, the current player will simply try closer to *curr_goal*.

5 Performance & Analysis

The statistics presented below represent the average amount of moves that were taken by the red player and blue player as well as the average runtime in seconds from all of the 25 tests. In addition, the number of times the tests resulted in the red player winning, the blue player winning, and a draw happening are reported.

The game tree search methods described in section 3 were implemented in accordance to their description. The soccer game evaluation function from section 4 is used as a direct utility function for minimax and α-β pruning. However, this function is utilized as a playout policy for Monte Carlo tree search to decide which node to expand from the expansion phase, and the utility preference for the simulation.

5.1 Minimax Game Tree and $\alpha - \beta$ pruning

Statistics						
Depth	Red Moves	Blue Moves	Red Wins	Blue Wins	Draw	Runtime
1	11.12	11.32	2	7	16	25.587
2	6.24	7.04	1	3	21	471.770
3	7.56	8.04	4	1	20	17347.543

Table 1: Statistics collected from 25 tests of the specified depth configuration in minimax alone

The average runtime from the tests do increase exponentially with the depth count, which is expected. However, somewhat surprisingly, the minimax search tree that was limited to the first depth produced the most wins for both agents.

It should be noted that the evaluation function for estimating the utility in the game is rather sophisticated. What is most likely happening is that the sophisticated evaluation is nullifying the relevance of deeper search trees, as both agents are gathering a lot of information from the possible moves of each other. Thus, both agents could not easily exploit the shortcomings of one other, which lead to a draw more often than not.

Statistics						
Depth	Red Moves	Blue Moves	Red Wins	Blue Wins	Draw	Runtime
1	13.4	13.6	3	2	20	7.474
2	10.08	10.32	3	12	10	28.7446125
3	10.2	10.68	8	6	11	215.0245683
4	13.4	13.6	3	0	22	1881.165718
5	13.56	13.72	3	0	22	13227.58784

Table 2: Statistics collected from 25 tests of the specified depth configuration from minimax with $\alpha - \beta$ pruning

Similarly to minimax search alone, minimax search with $\alpha - \beta$ pruning has exponential runtime. However, the runtimes from the $\alpha - \beta$ tests seem to be a factor of 10 lower than the minimax search tests. Since the branching factor of $\alpha - \beta$ is $\sqrt{b}$ compared to b with minimax alone, these results do make sense.

It seems that both minimax alone and minimax with $\alpha - \beta$ pruning produced the most wins at somewhat lower depth configurations. In addition, minimax with $\alpha - \beta$ pruning also shows that the relevance of deeper trees are nullified by the sophisticated evaluation function.

Interestingly, depth 1 performed just as poorly as depth 5. For depth, the search trees most likely could not effectively exploit the game state, despite the specified cases for handling non-terminal states in the evaluation. However, search trees of depth 2 and 3 got the most amount of wins. Given the faulty nature of searches that limited to depth 1 and depth 5, depths 2 and 3 most likely found the best balance between evaluation function's complexity and depth knowledge.

5.2 Monte Carlo Tree Search

Statistics						
Playouts	Red Moves	Blue Moves	Red Wins	Blue Wins	Draw	Runtime
10	6.92	7.28	0	0	25	133.1609636
25	16.32	16.88	4	0	21	479.710226
50	14.72	15.32	0	2	23	844.9317201
75	12.52	13.16	1	0	24	1024.14957
100	13.44	13.88	0	0	25	1466.594685

Table 3: Statistics collected from 25 tests of the specified maximum playout configuration

The average runtimes from all 25 tests show exponential growth. This does make sense, as with minimax game tree search, Monte Carlo game tree search has exponential time complexity.

The influence that the the number playouts has on the number of wins are quite interesting. Similarly to the nature that the depth had with minimax search, the highest and lowest playouts produced the least amount of wins. The monte carlo tree search seems to be balancing the evaluation function's non-terminal cases and the deep knowledge the best with 25 playouts.

Interestingly, as the number of playouts increases from 25, less wins are detected from the tests. This behavior, just like the minimax search, shows that more knowledge gathered from the evaluation function will create agents that cannot effectively exploit each other's flaws, which causes the agents to come to a draw most of the time.

5.3 Comparison of algorithms

Based on the statistics that were reported from the tests, $\alpha - \beta$ pruning obviously resulted in the most wins. The minimax search and $\alpha - \beta$ pruning utilize the evaluation as a direct function for utility. Therefore, both algorithms can effectively assess the situations in the game. However, $\alpha - \beta$ pruning can easily isolate the best cases for the minimax search, which works to easily built more competitive agents that can exploit the other agent's flaws.

Unlike the minimax search, the Monte Carlo tree search used the evaluation function as an playout policy as guidance. Unfortunately, this indirect use of the evaluation function failed to find agents that could exploit the other agent's shortcomings to win. This is clearly illustrated by the fact the Monte Carlo tree search came to a draw far more often than minimax search combined with and without $\alpha - \beta$ pruning.

6 Conclusion

Game tree search algorithms are simple, yet effective strategies for building competitive agents for turn-based games with two players. However, the effectiveness of these algorithms is highly dependent on a good domain-dependent utility function.

In this paper, it was shown that direct utility evaluation on the soccer using minimax tree search was sufficient for create agents that could win the game. Applying $\alpha - \beta$ pruning to the minimax search improved the results in terms of average runtime and the number of wins that the agents could achieve. Conversely, indirect evaluation using Monte Carlo tree search was not sufficient for effectively finding agents that could win the game.

References

[1] TU Masters. Computer science 4253-01. https://github.com/TUmasters/cs4253, 2017, 2023.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearsons, 4 edition, 2021.