Monte Carlo Tree Search Applied to Plants vs. Zombies

Amy Zhang
CS 238: Decision Making Under Uncertainty
Stanford University

***Abstract*—Plants vs. Zombies poses numerous sources of uncertainty, including stochastic resource constraints and stochastic adversarial behavior. In order to succeed in the game, a player must be able to build up their defenses to handle an exponentially large variety of situations. This paper introduces an application of Monte Carlo Tree Search (MCTS) to address the stochastic and exponential nature of the Plants vs. Zombies game tree. The performance of MCTS is compared against a stochastic policy on both a small game board and a large game board. Analysis of 400 simulated games shows that MCTS greatly out-performs the stochastic policy, particularly as the size of the game board increases.**

I. Introduction

Online planning algorithms, such as Monte Carlo Tree Search, are often used in applications that involve exponentially large state and action spaces. These algorithms compute estimations of the optimal policy while interacting in their environment, hence the name online planning. Through using the reachable state space to generate an action at each step, these algorithms greatly reduce the computational complexity and storage requirements that come with analyzing large or continuous state and action space problems.

MCTS has been proven to be incredibly effective when applied to finite horizon games such as Go or Chess [1]. Plants vs. Zombies is a perfect example of a finite horizon game with a well-defined winning terminal state and loosing terminal state. Furthermore, there are many elements of uncertainty inherent in the game such as which lane a zombie will enter in and whether the player will have enough resources to plant a defensive plant. Given the exponential branching factor of the Plants vs. Zombies game tree, other popular reinforcement learning algorithms, such as Q-learning or Sarsa, are intractable not only from a computation perspective but also from a data storage perspective. All of these properties made MCTS a great candidate for this application.

II. Related Work

When applying MCTS to the area of video games, a common challenge is navigating complex games with "dynamic information and complex victory conditions". An application of MCTS to the complex game of Kriegspiel is studied in [2]. Ciancarini et al compared the performance of traditional MCTS, MCTS which only involved information broadcasted by one of the players and MCTS with a depth of 1. Both of the second approaches were able to significantly reduce the noise of the MCTS simulations. Another complexity, discussed in [3], are games that require the achievement of sub-goals in order to maximize rewards at the end of the game. Depending on the depth of the rollouts, it is possible that MCTS is not able to incorporate a higher level of planning that involves short term sacrifices to achieve sub-goals. Waard et al introduced the concept of an option, which is a policy that allows the player to achieve a specific subgoal. Using MCTS with progressive widening Waard et al were able to successfully apply MCTS to the area of general video game playing. They were able to demonstrate that MCTS could handle planning at a higher level of abstraction through having MCTS choose between options instead of actions at each step.

Another challenge when performing MCTS in game settings is balancing computation and time constraints at each step with making sure that MCTS performs enough rollouts to explore the reachable state space. One solution to this problem, presented in [4], is rapid action value estimation (RAVE) which provides a framework for sharing knowledge between related nodes in the search tree. RAVE does this through using the all-moves-as-first heuristic which defines a score for each possible move regardless of when it is played in the game, thereby allowing repeated actions to be quickly scored during simulations. Another approach, discussed in [5], is to use offline MCTS planning to bootstrap deep-learning online approaches, which tend to perform much more quickly.

III. Model

Fig. 1. Traditional version of Plants vs. Zombies.

The traditional version of Plants vs. Zombies (see fig.1) involves a 5x9 game board, divided into 5 horizontal lanes, where adversarial zombies enter the game on the right side of the board and proceed down the rows towards the home that is one the left side of the board. Should any zombie manage to make it to the leftmost side of the board, the player loses and the game is over. The player's objective is to survive the night by planting defensive plants that have the ability to harm and/or kill zombies in their specific rows. In order to obtain these plants the player must plant sunflowers to generate sun which can then be used to buy plants. Should a zombie run into a plant, it begins eating the plant until either the plant is gone or the zombie is killed. Traditionally, there are 5 types of zombies with special behaviors and skills and there are 7 types of plants with special defense mechanisms, plus sunflowers. This makes for approximately $14^{45} = 3.76 \quad 10^{51}$ different game states. This paper will focus on a simplified version of the traditional game described below.

A. States

This version takes place in either a small 3x3 game board or large 10x10 game board that is stored in a zero-indexed array of length 9 or length 100, respectively. There is only one type of plant and two types of zombies (a strong zombie which takes two hits to kill and a normal zombie which takes one hit to kill). This makes for $5^9 = 1.95 \quad 10^6$ states for the small board and $5^{100} = 7.88 \quad 10^{69}$ states for the large board. The starting state consists of either a strong or a normal zombie at the rightmost side of the board. The night is modeled as 20 time steps (t_{end}) so the terminal state is reached when 20 time steps have been reached and there are no more zombies on the board or when a zombie reaches the left side of the board. Note: even if 20 time steps have been reached, the game is not over until all zombies have been cleared from the board.

B. Actions

To model the resource constraint of collecting sun, I allow the player to plant a plant with probability $p_{plant} = 0.5$ at each time step. The one exception is the first time step when the player is given an advantage and is always able to plant a plant. The player can plant a plant in any cell that is empty in the current state and the chosen cell's index is recorded as the action. If a player is not able to plant a plant or chooses not to, the action is recorded as n where n is the number of cells in the game board.

C. Transition Model

I allow the zombies on the board to take a step forward at each time step with probability $p_{move} = 0.5$ to model the slow nature of the zombies. So long as it is the night, a single zombie can enter from the left side of the board with probability $p_{new} = 0.7$ at each time step. If a zombie runs into a plant, the plant disappears and the zombie moves forward. Once a normal zombie is hit once by a plant it disappears from the state. The first time a strong zombie is hit by a plant it becomes a weakened zombie and is still able to advance leftward until it is hit for a second time and disappears from the state.

D. Reward Model

The terminal state of a zombie reaching the leftmost side of the board is assigned a reward of 200 since this is a the loosing state. The zombies not only win but they eat the brains of the people in the house, which is very bad. The terminal state of reaching 20 time steps and clearing the board of zombies is assigned a reward of 100 since this is the winning state. The humans merely survive to fight another day. Throughout the game if a plant kills a zombie, that state is assigned a reward equal to the number of zombies killed. If a plant gets eaten by a zombie, that state is assigned a reward equal to the negative of the number of plants eaten. This is to model the fact that plants take resources to plant therefore allowing a zombie to eat a plant is detrimental to the game. I chose a discount factor of $= 0.95$ to account for the fact that plants persist throughout the game, unless they are eaten by a zombie, so the same plant can kill multiple zombies in that lane.

IV. Methods

A. Random Policy Simulators

In order to quantify the performance of MCTS in comparison to a baseline, two random policy game simulators were built using the game model outlined above. The first simulator was aimed at simulating games in the small 3x3 board and the second simulator was aimed at simulating games in the large 10x10 board. At each time step of the game, the simulators recorded tuples consisting of the $(state; action; reward; nextstate)$. Any time a player could plant a plant (p_{plant}), these simulators planted a plant randomly in any free cell in the current state.

B. Simulators with Monte Carlo Tree Search

Two simulators were also built to apply MCTS to the small board and large board. Any time a player could plant a plant (p_{plant}), these simulators ran MCTS from the current game state and proceeded with the action returned by MCTS. The details of the MCTS algorithm are described in the following sections.

C. Selection

Given the current state of the game (s), the algorithm greedily explores actions that maximize the upper confidence bound (UCB). In other words, for each iteration of MCTS, the algorithm chooses to explore the action (a) that maximizes the UCB, which is defined as follows:

$$Q(s;a) + c\sqrt{\frac{\log(\sum_a N(s;a))}{N(s;a)}} \tag{1}$$

$N(s;a)$ is defined as the total number of times the algorithm has taken action a from state s. The hyperparamter c scales the exploration bonus (second term). The more times we have taken action a from state s the greater $N(s;a)$ will be and the

smaller this exploration bonus will be. $Q(s;a)$ is the current estimate of the action value function and will be defined in the back propagation step. This step will return an index in the current state in which to plant a plant.

D. Expansion

The expansion step involves generating the state that follows from taking the selected action a from state s. Given that a plant is planted in the selection step, this exploration step involves updating the board if any zombies have been killed, moving the zombies, and recording whether any plants have been eaten. In general simulations of the game, the zombies can only move with probability p_{move}, however since each zombie always stays in its own lane and can only ever move in the forward direction, the algorithm projects the worst case scenario with zombies moving forward deterministically at each step.

E. Rollout

Eventually the algorithm will either reach a state that has not yet been explored or will reach the maximum depth parameter (d). When it reaches the maximum depth or we run out of possible actions (ie. the game board has no free cell), the algorithm returns 0. When it reaches a new state, $N(s';a)$ and $Q(s';a)$ are initialized and a value estimate is generated through rollouts. Since zombies move slowly, there was plenty of time at each step to perform rollouts. Therefore, the decision was made to perform each rollout until a terminal state is reached. At each step of the rollout, a random policy is followed to plant plants with probability p_{plant}. Rewards are assigned according to the reward model described above and discounted accordingly by . Transitions occur according to the transition model.

F. Back propagation

Once an estimate for the action value function has been obtained through the rollout step, the algorithm performs the following update up the tree:

$$Q(s;a) \quad Q(s;a) + \frac{q \quad Q(s;a)}{N(s;a)} \tag{2}$$

where q is the discounted reward from the rollout. The $N(s;a)$ counters are updated accordingly.

G. Hyperparamters

There are three hyperparamters associated with MCTS: k_{max}, which is the number of simulations we run from the current state s, the depth (d), which is the depth of the rollouts and c, which scales the exploration bonus in the UCB. In an effort to allow MCTS to explore as many different actions as possible, I set $k_{max} = 20$ for the small board and $k_{max} = 150$ for the large board. As mentioned previously, the rollouts are performed to a terminal state so $d = 20$ for both the small board and large board. This ensures that a rollout always reaches a terminal state, otherwise the algorithm tends to generate sub-optimal actions. c was initially set to a value of 10 and the algorithm tended to perform well with this measure so this value was kept for the final version.

V. Results

TABLE I
WIN RATES FOR 3X3 BOARD

Policy	Win Rate (over 100 simulated games)
Random	11%
MCTS	39%

TABLE II
WIN RATES FOR 10X10 BOARD

Policy	Win Rate (over 100 simulated games)
Random	2%
MCTS	95%

As shown in the tables, MCTS not only greatly outperformed a random policy but also performed exceptionally well in the large board. Given the finite horizon nature of the game, MCTS was a very fitting algorithm as, at each step, it was able to perform multiple rollouts from the current state to the end of the game and decide the action using those rollouts. I imposed virtually no computing or time constraints on each application of MCTS which, in most applications, is unfeasible but this allows us to underscore how well MCTS can perform on this problem. As mentioned previously, I did observe that MCTS tended to choose sub-optimal actions if its rollouts did not reach a terminal state but this could be partially addressed by increasing the reward for killing zombies or adding new rewards for following common game heuristics.

MCTS did not perform as well on the small board. This is likely due to the fact that $p_{plant} < p_{new}$ and $p_{plant} = p_{move}$. The zombie only needed to advance two steps before sending the game into a losing state and so it was easy for one of the three lanes to go unprotected and for a zombie to enter the house on that lane. In fact many of the losing games simulated using the small board ended quickly within just a few time steps.

The true litmus test for any algorithm is not comparing it against a random policy but rather comparing it against a human player's heuristics and strategies. A common heuristic for playing Plants vs. Zombies is to plant plants at the leftmost side of each lane so that they have the maximum potential to kill zombies and not get eaten. Of course, this means that if the plant isn't able to kill the zombie as it advances, the player then has no chance of saving the game once the zombie eats the plant and has direct access to the house. Therefore, a human player must balance this risk and usually does so by putting more deadly plants closer to the house and more dispensable plants closer to the zombies. Another common heuristic is to make sure that each lane is protected before adding additional defenses to a lane. This is due to the fact that even in the real version of the game, the player has no idea which lanes the zombies will appear in and what kind of zombie will appear in a lane. Fig. 2 shows four winning terminal states generated