Monte Carlo Tree Search for the Pokemon Trading Card Game

Tyler Uhlenkamp TAUHLENK@IASTATE.EDU
Iowa State University

Tyler Chenhall TCHENHAL@IASTATE.EDU
Iowa State University

Abstract

The Pokemon Trading Card Game is a popular turn-based card game usually played by two players. Its partially observable, stochastic, multi-agent nature combined with its complex rules, large state space, and demand for long-term planning make it an interesting target for modern artificial intelligence research efforts. An agent based on a Monte Carlo Tree Search (MCTS) algorithm with added knowledge via a heuristic function was developed and tested in games of the Pokemon TCG against human players and other agents. Although the MCTS agent performed well against random agents, its performance against human players was worse than expected. Possible explanations for this performance gap are explored and suggestions for future research and technical enhancements are proposed.

1. Introduction

The Pokemon Trading Card Game has several characteristics that make it an interesting problem in artificially intelligent agent design. The general format and objective of the game, as well as its formal environment characterization and other important characteristics are summarized below.

1.1. Background: The Pokemon Trading Card Game

The Pokemon Trading Card Game is a collectible card game inspired by the Pokemon TV show originating in the early 1990s. Games are typically played by two people, each of whom uses a separate deck of 60 cards. Designing a deck involves significant planning and strategy which is separate from game-play, and so we do not discuss the details in this document. Each player's deck consists of several types of cards: Energy cards, Trainer cards, and Pokemon cards. Pokemon cards represent the creatures from the popular TV show, and also represent the main players in the card game. A person plays Pokemon cards throughout the game, and attempts to strengthen them while doing "battle" in a turn-based format. Each Pokemon has several characteristics, including hit points, weaknesses, resistances, and attacks. Some Pokemon can also evolve to become stronger or more useful. Trainer cards serve as helpers in the game, often aiding to acquire cards needed for various strategies, or "healing" a Pokemon. Finally, Energy cards are needed to use certain actions such as using an attack or switching which Pokemon is currently active.

Game-play for the Pokemon card game is fairly complex. At the beginning of the game, each player draws seven cards from their shuffled decks. Re-drawing if necessary, they select basic (non-evolved) Pokemon to place in some of their six in-play Pokemon slots, one of which is considered "active". Then, each player deals out six face-down prize cards to set aside. These are collected one-by-one whenever a player defeats an opponent's Pokemon.

After set-up, the game proceeds in turns. During a single player's turn several actions may occur. Each turn begins with the current player drawing a card. The person may then play one or more cards from their hand, including up to one Energy card, and one Supporter Trainer card. Pokemon evolution cards can also be used to evolve those currently on the field as long as that card has been in-play for at least one turn. The player may also choose to switch their active Pokemon by paying its retreat cost (discarding a number of Energy cards from the active Pokemon). These actions may carried out in any order. The turn ends either when the player chooses to end their turn or upon choosing to attack with their active Pokemon. Note that Trainer cards are typically discarded after their effects apply, while Pokemon and Energy cards remain in play until the Pokemon takes more damage than it has hit points. Energies are typically not discarded for an attack unless otherwise

Com S 472/572 Final Project Report

specified by a specific card.

How does the game end? There are three ways to win. First, if a player has no cards remaining in their deck when their turn starts, their opponent wins immediately. A player also loses if their last in-play Pokemon is defeated, or when their opponent claims their sixth and final prize card.(Pokemon, 2015)

It is important to note that card effects in the Pokemon game are both diverse and disparate in their behavior. Some Pokemon attacks will simply apply damage to the opponent's active Pokemon, while others do damage to many or all of them. Other attacks apply status effects with compounding results, heal, or even cause an opposing Pokemon to de-evolve. Trainer cards also have a variety of effects, many of which allow the player to search their normally unseen deck in order to select one or more cards of their choice. These effects contribute to the complexity of the game, both in strategy and representation.

1.2. Summary: Game Environment

Several characteristics of the game environment pose challenges when attempting to develop an intelligent computer agent to play the game well. At any given point in time, a player cannot typically see the cards in their opponent's hand, deck or prizes. Similarly, they cannot usually see their own prizes or deck. On the other hand, all previously played cards (discard piles) are visible, as are all cards currently in-play. This indicates that the game state is only partially observable. The Pokemon game also has stochastic elements, including coin flips, shuffling of the deck, and card draws. As a two-player game with only one winner (draws are rare), it is also necessarily an example of a multi-agent environment. Finally, at any given point in a turn, the player may have numerous actions available, such as switching, attacking, playing cards from their hand, or simply ending their turn. This often results in a large branching factor, even before considering possible stochastic effects. Good human players are often able to concoct complex strategies combining the effects of multiple actions for maximum benefit.

Fortunately, some aspects of the card game lend themselves to standard analysis techniques. Each action in the game is executed sequentially, and the state is both static and discrete; the game state will remain the same even if an agent requires some time to select a move. The main limiter of agent analysis time is just the patience of any human opponents or observers.

In summary, playing Pokemon well requires long-term planning that accounts for a large and uncertain game tree.

2. Methodology

In an effort to develop an agent which could play the Pokemon Card Game competently against human players, the developers implemented a custom state representation and game logic from scratch, then created an agent based on a Monte Carlo Tree Search (MCTS) and a well-tuned heuristic function.

2.1. Game Logic and State Representation

All code representing game logic and state representation was written in Java specifically for this project. At any given point in time, the game state consists of a variety of information. This includes which cards are in both players' hands, discard piles, and prizes, as well as the order of cards in each deck. Each player can have up to six Pokemon in play, each with Energy attached. There are also temporary or persistent attributes to keep track of, such as damage done to each Pokemon so far, status effects on active Pokemon, whether an Energy or Supporter Trainer card has been played in a given turn and so on. Due to the large volume of information involved, the developers created a custom State data structure consisting of two identical objects representing the two players' cards, along with the previously mentioned attributes. The State can be cloned for use in agent decision making.

Cards are represented using a collection of 3 subclasses of the Card class: Energy, Trainer, and Pokemon. A single card is created by reading from a plain-text database which stores vital details as a list of attributes with some shorthand effect notation. For example, the Pokemon card "Hydreigon" has an attack called Consume. This attack requires 1 Dark, and 2 colorless Energy to use, has a base damage of 40 and heals the active Pokemon by as much damage as it does to the opponent. This attack is represented in the database as: `1D2C,Consume,doDamage:40,healDamage:EQ`. This format was chosen to avoid needing to write a new class for every single card, the convenience of brevity, and the ability to define a single "effect handler" which can parse and interpret the effects of complex, multi-faceted attacks. Additionally, new cards can be quickly incorporated for a new deck, simply by modifying the database entries.

At any given point in time, the code is also able to generate an exhaustive list of available commands. The same text-based command syntax is shared between human plays (input via command line) and computer agents

2.2. The Agent Program

The Monte Carlo Tree Search algorithm was chosen as the basis of the Pokemon Card Game agent because the algorithm has seen wide success in a variety of related games

such as Poker, Settlers of Catan, and the "Magic: The Gathering" card game. Furthermore, agents based on planning or rule-based approaches wouldn't be robust enough to create new strategies for brand new decks. The minimax algorithm also proves to be an undesirable candidate due to the partially observable and stochastic nature of the game environment as well as the large branching factor and need for long-term planning. The MCTS algorithm, on the other hand, has been applied to similar games and can be easily adapted to partially observable and stochastic environments. It also can exhibit long-term planning and works well in environments with a large branching factor, such as Go.

2.2.1. Monte Carlo Tree Search Implementation Details

A full description of the MCTS algorithm and its properties is left to previous research while a summary of the implementation used in the Pokemon TCG agent is provided.

The MCTS algorithm works by building a search tree node-by-node and estimating the effectiveness of each node by simulating many games which visit that node and keeping track of the expectation of reward (Browne, 2015).

The MCTS agent developed for the Pokemon TCG agent maintains a persistent game tree structure which is saved between turns. At the beginning of a turn, the current state is located in the current game tree and set as the new root node. In this way, results of simulations in previous turns can still provide information about the usefulness of nodes still under consideration. Next, the MCTS algorithm, consisting of Selection, Expansion, Simulation, and Back-propagation stages (described below), is repeated in a loop for up to five seconds. Each iteration may add up to one new node to the search tree and simulates one play-out to the end of the game. Usually, the PCG agent was able to complete 15,000 such iterations per turn on a consumer-grade laptop computer. Finally, the child of the root node with the highest computed value is chosen as the next action.

Typical MCTS implementations assume uniform node types, however the Pokemon TCG bot employed a unique tree structure in order to account for the stochastic nature of actions at each step in game play. Namely, it is assumed that each action results in one or more possible states and the state which is actually reached is the result of random chance. Therefore, the game tree consists of alternating levels of min/max nodes and chance nodes. A min/max node seeks to maximize or minimize the value of its children (depending on the current player, which is stored in the state), while chance nodes randomly result in one of their children (Roelofs, 2012).

Each iteration of the MCTS algorithm consists of four stages as noted previously: Selection, Expansion, Simulation, and Back-propagation (Browne, 2012). The implementation of each stage is described below.

Selection: Starting from the root node, "optimal children are selected recursively until a node which is a leaf node or a node which still has un-expanded children is encountered. When selecting children, one must be careful to balance the exploitation of nodes which are known to be favorable to the current player and nodes which have not been explored as much. To accomplish this goal, the Upper Confidence Bounds for Trees (UCT) formula was used (Kocsis & Szepesvari, 2006). Namely, at each step the child node which maximizes the following equation is chosen:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

where v_i is the value of the current node, C is a constant (chosen to be $\sqrt{2}$), N is the number of simulations that visited the parent node, and n_i is the number of simulations carried out which have visited the child node.

Experiments to incorporate the heuristic function into the Selection stage were attempted. Namely, the heuristic function was added as a factor in the UCT equation such that nodes with higher heuristic values would be more likely to be explored. Experiments showed this didn't significantly improve the performance of the agent.

Expansion: If the selected node has unexpanded children (actions available from the given state which do not have corresponding child nodes), then a new node is added to the tree for that action. If the selected node is a min/max node, then a chance node is added for the given action. If the selected node is a chance node, then a new min/max node is created as one of its possible resulting states.

Simulation: An entire game starting from the current node's game state is simulated. At each step, a random action is chosen until the game is in a terminal state. The result of the game is recorded.

Other options for the Simulation step were attempted. Namely, terminating after a given number of steps and returning a heuristic value was tried and didn't improve the performance of the agent (as one might expect since more playouts could be conducted each turn).

Back-propagation: Finally, the results of the simulation or play-out are propagated back up the path from the expanded or selected node to the root. The number of simulations of each node is incremented and the value of the node is modified based on the type of node. Chance nodes simply take on the weighted average of its child nodes (one of which has just been updated), while min and max nodes have their values increased or decreased depending on the

game result and the type of node.

2.2.2. Heuristic Functions

Two different heuristics or metrics have been defined for this project. The first heuristic helps an agent choose its initial set of in-play Pokemon. The second heuristic allows agents to assess roughly how desirable a particular non-terminal game state is, which is important since the game tree is large, and it is generally unreasonable to do a large number of searches which end in a terminal state.

The first heuristic estimates the desirability of a particular basic Pokemon by examining its hit points and retreat cost. A card is rated more favorably if it has lots of hit points, and a low retreat cost. Roughly speaking, this implements the strategy that a good starting Pokemon should be strong, but also easily switched-out if needed. There are certainly limitations to this fixed strategy. For example, another strategy is to use a weak or high-retreat cost Pokemon as an initial sacrifice, while the player uses the extra time to strengthen their roster of other in-play cards. A more sophisticated heuristic might also take into account attack Energy costs, and what other cards the player or agent has in hand.

The second heuristic evaluates game state. This heuristic evaluates both halves of the state in the same way, and takes the difference of the scores for each player. The result is a composite score, indicating a favorable set-up for player one when the score is large (and conversely, a large negative score is good for player two). A single player's cards are evaluated with multiple measures. Attributes resulting in a high (good) score include: high remaining hit points on in-play Pokemon, good allocation of Energy cards, having more than one Pokemon in play (to avoid loss scenario two), and collecting prize cards. Having an active Pokemon with a detrimental status such as "sleep", "poison", or "paralysis" is viewed as a negative. Due to the large number of variables involved, it is challenging to weight these parameters appropriately in order to avoid accidentally promoting bad agent behavior.

3. Results

Since the game interface supports play either by a human or computer agent, the project can be tested either by having a human player or by playing two different computer agents against each other. All tests were performed with both players given the same deck of Pokemon cards to avoid unfair card advantages.

3.1. Test Results

Tests were performed on the Monte Carlo Tree Search agent both against a random-action choice agent and against human players. Repeated trials indicate that the Monte Carlo-based agent consistently outperforms the random action agent. However, against a human player the agent is not typically competitive. While it does win the occasional game, the Monte Carlo Tree Search agent often struggles to find sequences of actions which can keep up with a typical person. In particular, some actions which seem obvious to a human player (such as using the strongest attack) are not selected as frequently as expected. This means that while the agent makes some good moves, it is unlikely to execute all the best actions in a given turn (though they may happen over several turns). Increasing the search time might yield more effective move sequences. Likewise, improvements in the heuristic might allow more accurate evaluation of partial rollouts, thus allowing the agent to search with more breadth.

3.2. Explanations

The implemented agent performed worse than expected. The mostly likely explanations include implementation issues and inherent difficulties of the problem.

Implementation Issues: It may be the case that the implementation of Monte Carlo Tree Search contains logical errors which prevent the agent from consistently carrying out the MCTS algorithm. Namely, there are several areas that are likely to be particularly error-prone.

First of all, the proper treatment of min/max nodes was not made clear in any of the reviewed literature. The back-propagation step description never indicates special treatment of min/max nodes depending on whether they are a min or a max node, however logic suggests that min and max nodes should behave differently in back propagation such that min nodes should later select children with smaller scores more frequently. This distinction wasn't explicitly described in any tutorials or literature reviewed.

Second, our solution to partial observability involved cloning the state such that all observable attributes are copied exactly, and all unobserved attributes are copied randomly (ex: simulated hand is drawn from the set of un-seen cards). This code may also contain logical errors which prevent proper simulation of actions. Furthermore, it's unclear whether or not this is the proper treatment of partial observability.

Large State Space: It's also the case that the branching factor in Pokemon TCG is fairly large (around 10-30 usually). It might be the case that 5 seconds is not enough time for the MCTS algorithm to gather enough information about the value of child nodes to make the optimal decision. Improvements may be made by increasing the search time or finding ways to limit branching and shrink the state space. One may also consider imposing some structure on the game such as an ordering to the moves within a turn in

order to limit the branching factor.

3.3. Future Work

One area of future work would be to fine tune the heuristic formula used in evaluating non-terminal states. Additional testing and fine tuning might improve the results of decisions made from partial rollouts. Alternatively, machine learning techniques might be used to construct a heuristic. This would be done by generating a large number of potential game states and having a human observer rate their favorability for winning.

Another possible area of study is to focus on smaller goals. While the entire game space is too large and stochastic to search completely, it may be possible to efficiently seek a smaller goal. These smaller goals might include "what to do to evolve a Pokemon and give it three energies?", or "what is the quickest way to defeat one of the opponent's Pokemon?". These strategies would allow the agent to focus on a much more attainable goal which still moves it in the direction of winning the entire game.

4. Conclusion

A system for simulating a match of the Pokemon Trading Card Game was implemented which manages game state and generates available actions. The game interface also supports play by humans or computer agents.

An agent based on Monte Carlo Tree Search was implemented, but it performed worse than expected. Possible explanations include implementation issues, state space complexity and inherent difficulties in problem domain, and inadequate computational power and time.

5. Individual Contributions

Tyler Chenhall: Wrote code for representing and manipulating game state as well as the command line user interface for playing and testing the game. Also wrote the random-move agent. **Tyler Uhlenkamp:** Wrote and tested code for Monte Carlo Search Tree agent.

References

Browne, Cameron. Monte carlo tree search - about. 2015. URL http://mcts.ai/about/index.html.

Browne, Cameron et al. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games*, 2012. URL ccg.doc.gold.ac.uk/papers/browne_tciaig12_1.pdf.

Kocsis, Levente and Szepesvari, Csaba. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, 2006. URL www.sztaki.hu/~szcsaba/papers/ecml06.pdf.

Pokemon. *Pokemon Trading Card Game Rulebook*. The Pokemon Company, 2015. URL http://assets1.pokemon.com/assets/cms2/pdf/trading-card-game/rulebook/xy8-rulebook-en.pdf.

Roelofs, G.J.B. Monte carlo tree search in a modern board game framework. 2012. URL https://project.dke.maastrichtuniversity.nl/games/files/bsc/Roelofs_Bsc-paper.pdf.