Learning Pokemon Trading Card Game by using Monte Carlo methods

Luka Filipovic[1]

***Abstract*— In this paper, I describe Monte Carlo methods used to create an algorithm which plays the Pokemon Trading Card Game - a popular 2-player game played using collectible Pokemon Trading Card Game cards.**

I. Introduction

Pokemon Trading Card Game is a 2-player card game, where players build a 60-card deck with Pokemon Trading Cards. The game consists of players drawing cards from their deck, playing cards from their hand onto the playing field, and attacking with cards on their field. For the detailed explanation of the rules, please refer to the official rulebook[2].

This paper will employ Monte Carlo techniques[5] on a specific deck, which is known as a "Round" deck in the Pokemon Trading Card Game community. More information about this deck can be found in a youtube video made by a well-known competitive player Kyle "Pooka" Sucevich, where he explains how this deck works[3]. We use a slightly simplified version of the "Round" deck (i.e. we don't add some special cards which are only important for certain scenarios which can happen in high-level play), which contains 13 different cards. Here is the deck list:

4x Evosoda, 116/146 XY (ID = 0)
4x Tympole, 40/113 Legendary Treasures (ID = 1)
4x Palpitoad, 41/113 Legendary Treasures (ID = 2)
4x Seismitoad, 42/114 Legendary Treasures (ID = 3)
4x Meloetta EX, RC25/RC25 Legendary Treasures (ID = 4)
4x Muscle Band, 121/146 XY (ID = 5)
4x Cheren, 91/98 Emerging Powers (ID = 6)
4x Professor Juniper, 101/114 BW (ID = 7)
4x Heavy Ball, 88/99 Next Destinies (ID = 8)
4x Level Ball, Next Destinies 89/99 (ID = 9)
4x Pokemon Communication, 99/114 BW (ID = 10)
4x DCE, Next Destinies 92/99 (ID = 11)
12x Lightning Energy (ID = 12)

Information about each individual card can be found in the Serebii card database[4]. ID doesn't have anything to do with the actual Pokemon Trading Card Game; it's only important for implementation purposes, as explained below.

II. Methods and Data

My algorithm is implemented in C++. I use 3 main structures: "Deck", "Hand" and "Field"[2], to implement corresponding parts of the game. Field also has pointers to a class Pokemon, which stores information about Pokemon cards currently on the field[2].

Class "Hand" contains a function bool Hand::play(int id, Field* field, Deck* deck, int arg1, int arg2). This function effectively "plays" the card, which means that, if successfull, the program does everything that would happen when a person plays that card in a game. "play" is a boolean function - if the move is valid, program will execute that move and return "true". If the move is invalid, this function will do nothing, and return "false". From now on, I will refer to possible moves as a triple $(id, arg1, arg2) : id \in [13] \cup \{-1\}, arg1, arg2 \in [12]$ - call the set of these triples M_3. Triples in M_3 correspond to id, arg1, arg2 arguments to play function. The complete list of moves is provided in the Appendix.

The original plan for this project was to play games on the Pokemon Trading Card Game Online platform[1], while concurrently running our own program. We would input what happens during events in which a player has no control over (e.g. cards we draw), and then our program would return what move we should play. However, due to server complications related to the release of the hugely popular Pokemon Go game, we have used our own program to play the entire game for us.

We don't simulate our opponent's play. Instead, beween turns[2], we have an option to input our opponent's "threat" level, which simply indicates how much damage our opponent is expected to do to us in the upcoming turns, given his current field position. Before the beginning of our turn, we input our opponent's threat level, how much damage our opponent dealt to us on his previous turn, what prize cards[2], if any, we drew, and what Pokemon to promote[2], if our active[2] Pokemon was knocked out.

Before we're required to make a move, our program first finds all possible moves we can make in the current position, by using a function find_available_moves(Deck* deck, Hand* hand, Field* field). Then it loops through each possible move. It does a certain number of Monte Carlo simulations for each available move (I will refer to this quantity simply as a "number of simulations" from now on) by calling a function simulate_game(Deck* deck, Hand* hand, Field* field, int move1, int move2, int move3, int index). This function starts the game simulation by first making the move (move1,move2,move3) $\in M_3$. Index is simply an argument which denotes which move we're making - it's really only

[1] Luka Filipovic is with the School of Computer Science, University of Waterloo, 200 University Avenue, Waterloo, Ontario, Canada N2L 3G1.

important for implementation purposes.

simulate_game will simulate the game for the next 5 turns. First it makes the move we're simulating, (move1,move2,move3), and then it chooses all successive moves uniformly at random. It also updates the global variable Damage, which is important for decision making, as will be explained below. Here is an example of our program running:

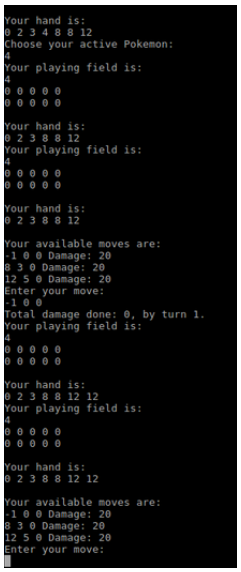

Fig. 1. Running the program from the beginning of the game for a single turn with a single move (attack and pass)

Before the turn ends, our Pokemon can attack and do damage, if certain conditions are satisfied[2]. In virtually all cases, more damage is better. Structure Field has a function int Field::calculate_damage(), which returns how much damage we do if we declare an attack. We set the global variable Damage to 0 before starting simulations for a specific available move. Then, we update Damage variable after declaring an attack in simulations (i.e. (-1,0,0)). There are two different rules used for updating Damage variable:

$$Damage = Damage + \frac{calculate\_damage()}{turn}$$

This is a Linear Damage rule, which simply adds all the damage done (with discount factor, as explained below).

$$Damage = Damage + \frac{e^{calculate\_damage()/20}}{turn}$$

This is an Exponential Damage rule. The idea behind this rule is that we should prioritize big damage moves a lot more, so that our program notices good, high-impact moves with greater certainty.

$turn \in \{1, 2, 3, 4, 5\}$ is simply a number representing how many turns have passed since the beginning of our simulation (starting from 1). We discount future turns in order to prioritize doing well immediately (it's much more impressive to do 200 damage on the next turn, than it is to do it after 4 turns).

III. Results and Discussion

In this section, when I refer to "damage done", I mean the actual damage we do as we play the game, and not the "Damage" variable from the previous section. Maximum achievable damage (theoretically) during the course of the first 5 turns is 770. I ran my program on a Lenovo X220 Thinkpad running ubuntu 16.04, 4GB of RAM, Intel Core i5 2520M CPU, and Samsung EVO 840 250GB SSD.

Games were played for 5 turns with 0 threat and 10 being a number of simulations for each available move. The following figure shows how I performed by playing the game myself (I choose all moves, "Damage" estimate is ignored):

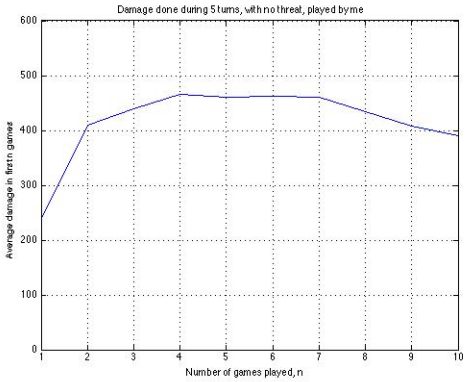

Fig. 2. No threat, played by myself

The following figure shows the same scenario, but picking moves based on the highest "Damage" calculation using Linear Damage rule.

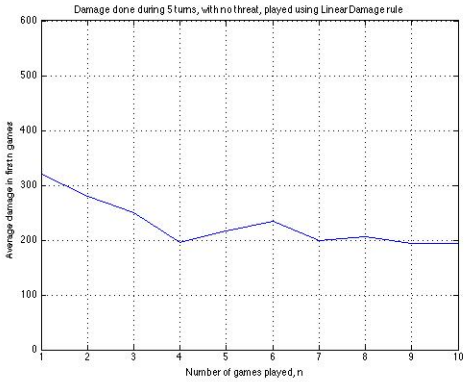

Fig. 3. No threat, Linear Damage rule used

We break ties by choosing one of the highest "Damage" moves uniformly at random.

The following figure shows the same scenario again, but only with Exponential Damage rule:

The time our program spends simulating games in between moves varies greatly on the number of available

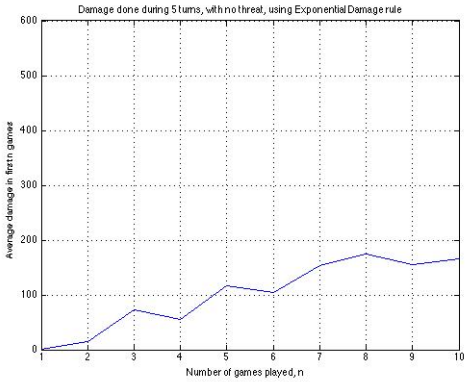

Fig. 4. No threat, Exponential Damage rule used

moves, but it appears to be around 6 seconds on average. This long running time, unfortunately, makes it difficult to collect data, and makes it virtually impossible to do a lot of simulations for every available move (1000+), so that we could guarantee, with high probability, that the recommended move based on "Damage" value is close to optimal.

Our alogithm's success greatly relies on the very beginning of the game - namely whether our first active Pokemon is Tympole (ID = 1) or Meloetta EX (ID = 4). Although Meloetta EX is an undesirable Pokemon to start with, I still output consistently over 200 damage when playing the game myself - I've started 3/10 games with Meloetta EX and my damage output over 5 turns was 240,200,240. However, 5/20 games were played using highest "Damage" estimate (this includes both linear and exponential rule) where Meloetta EX was the starting Pokemon, and these games resulted in a total damage output of 0,30,0,0,0.

The reason for this discrepancy most likely lies in the complexity of removing Meloetta EX from the active position. Our main attacker is Seismitoad, with Palpitoad being second best attacker, and Tympole third. Meloetta EX doesn't attack. Hence, if we don't remove Meloetta EX from the active spot, we will deal no damage. To remove Meloetta EX from the active spot, we need to do the following:

- Attach 2 energies to Meloetta EX, i.e. $(11, 5, 0)$ - this attaches two energies, or $(12, 5, 0)$ twice - this attaches one energy each time
- Put Tympole on the bench
- Retreat Meloetta EX from the active position, and replace it with Tympole/Palpitoad/Seismitoad, i.e. $(13, a, 0)$ where position a contains Tympole/Palpitoad/Seismitoad
- Attach energy cards to Tympole/Palpitoad/Seismitoad so that it can attack and deal damage

Given how we can attach only once per turn[2], and how each move is chosen uniformly at random in the Monte Carlo simulation, doing all listed things (in correct order) in 5 turns is very unlikely. With only 10 simulations per available move, all available moves will, in most cases, result in no damage being dealt, which means that we just randomly make moves. This has very low probability of being able to get Meloetta EX out of active position.

If games starting with Meloetta EX are ignored, we have the following results:

TABLE I
MELOETTA EX STARTS OMMITED

Type of simulation	Average damage dealt	Standard deviation
Playing myself	461.4	97.89
Linear Damage rule	238.8	71.67
Exponential Damage rule	235.7	145.2

As can be seen, both rules for Damage calculation result in a similar average performance. However, Exponential Damage rule has a significantly larger standard deviation. This variability is most likely caused by the fact that the Exponential Damage rule heavily rewards extremely strong moves, while mostly ignoring everything else. This is why the algorithm often successfully detects powerful moves if they are available, but acts pretty randomly when available moves aren't as strong.

IV. CONCLUSION

Although there is an online version of this game played by thousands of players, little progress has been done on creating an intelligent AI capable of playing the game at a high level. The current Pokemon Trading Card Game Online implementation does have an option to play against a computer opponent, but that opponent is very weak, so it is virtually never used by players who are preparing for tournaments, or are streaming for entertainment.

An important advantage of using randomized (Monte Carlo) approaches for this problem lies in a very dynamic environment of the Pokemon Trading Card Game. New cards are being released approximately every 4 months, and old cards are being removed from competitive play. Because of that, stochastic approaches have a great advantage over deterministic approaches, which use expert knowledge, due to their low dependence on the types of cards being used.

Additionally, randomized algorithms for playing the game could have other uses alongside simply making an intelligent computer player. Building strong well performing decks is a vital part of preparing for big events among competitive Pokemon Trading Card Game players. Strong players often rely on their experience from previously played games, as well as intuition, to build good decks.

However, due to new cards constantly being released, it can be hard to know exactly how certain decks stack up against other strong decks being used in competitive play

- there have been upsets featuring players with relatively unknown decks performing extremely well. This may have happened, in part, because other players weren't comfortable playing yet untested strategies at an important tournament. Hence, having a good stochastic simulator could prove extremely valuable for players preparing for tournaments, as it could provide statistical data on relative performance of different deck archetypes, which would assist deck building strategies.

V. Appendix

"Position" means an integer from 0 to 5, where 0-4 corresponds to the position on the bench, and 5 corresponds to an active slot[2]. Here is the complete list of moves:

- $(-1, 0, 0)$: attacks and ends the turn
- $(0, a, b)$: grab evolution with ID a in the deck, and evolve position b on the field
- $(1, 0, 0)$: places Tympole on the bench
- $(2, a, 0)$: evolves Tympole on position a into Palpitoad
- $(3, a, 0)$: evolves Palpitoad on position a into Seismitoad
- $(4, 0, 0)$: places Meloetta EX on the bench
- $(5, a, 0)$: places Muscle Band on position a
- $(6, 0, 0)$: plays Cheren - draw 3 cards
- $(7, 0, 0)$: plays Juniper - discard your hand and draw 7 cards
- $(8, a, 0)$: take card ID a from the deck, put it into hand
- $(9, a, 0)$: take card ID a from the deck, put it into hand
- $(10, a, b)$: shuffle card ID b into deck, and put card ID a from the deck into hand
- $(11, a, 0)$: attach DCE to position a
- $(12, a, 0)$: attach Lightning Energy to position a
- $(13, a, 0)$: discard 2 energy from active Meloetta EX, then switch it with position a

References

[1] http://www.pokemon.com/us/pokemon-tcg/play-online/
Pokemon Trading Card Game Online, general information and download client

[2] http://assets.pokemon.com/assets/cms2/pdf/trading-card-game/rulebook/xy8-rulebook-en.pdf
Official Pokemon Trading Card Game rulebook

[3] "Pokemon TCG Online - Round Deck!", The Top Cut, Youtube, https://www.youtube.com/watch?v=qDYwlPeEfpI

[4] http://www.serebii.net/card/english.shtml
Pokemon Trading Card Game database, cards can be found by clicking on corresponding sets (i.e. "Next Destinies")

[5] J.M. Hammersley and D.C. Handscomb, "Monte Carlo Methods", Methuen and CO LTD, London, 1964. Chapter 5