Prof. Robin Dawes

CISC 499

February 1, 2019

Designing an Effective & Adaptive A.I. for Stupide Vautour

PREFACE

I will do my best to provide only important details and not get too into the nitty gritty of implementations. This report will read more like a story about an adventure I undertook to find the holy grail. The holy grail being an A.I. that could outsmart a human or atleast make choices as if it was human and could think for itself. I also hope you do not mind reading a relatively informally written, yet hopefully inciteful report. My source code is also available on github for you to tear apart if you will. Word of warning, being a curious yet nieve programmer, I took a stab at coding this entirely in javascript with node.js in hopes to eventually make this a web app for everyone to enjoy and love. The issue being, that the only thing I really knew about creating a web app was my experience making a movie ticket ordering system in php, html and css. In other words, I had no idea what I was getting into with javascript, but I was determined to learn it.

INTRODUCTION — "'Stupide Vautour' is a simple card game in which players try to anticipate the other players' choices. The goal of this project is to design, implement and test an effective and adaptive simulated player." (Prof. Dawes, 2018). Throughout this report, I will consider various approaches that may solve this rather complex problem. These approaches will be first generally analyzed to understand how they may work in relation to the game. Secondly, if I deem that an approach is feasible to implement, I will attempt to do so and analyze the results. Lastly, adaptability will be measured by playing against my A.I. and attempting to throw it off by changing my strategy every few games. The resulting data should show that the A.I. adapts to the constantly change game state and to the choices made by it's opponent.

— APPROACHES —

The "Monte Carlo Tree Search" algorithm is a heuristic search algorithm that determines the next move for a game by looking at a set of probabalistically weighted choices. The 'tree' is traversed in a manner such that the choice that has a higher probability to lead to a win state is chosen. The general idea for the A.I. is to play a few games making choices based on a predetermined strategy (like choosing randomly) and then use the results of these games to compute a move that has the highest chance of leading to winning the game. In other words, when the A.I. wins one of these random games, it will favour similar choices made that lead to that win in future games. In hopes that similar choices will be more likely to lead it to winning again. Conversely, when it loses one of these random games, it will try to make different choices than the ones that originally lead it to defeat. This is an oversimplified explanation of this algorithm, because I haven't explained how it takes into account the massive number of possibilities that would have to be computed to traverse for the absolute best choice. In a nutshell, the algorithm doesn't create a tree for all possibilities, instead it dynamically either "explores" or "exploits" nearby nodes (nodes are just possible moves/choices). Exploring nodes is usually done when a good move node hasn't been previously found in the nearby search space and thus isn't exploitable (worth choosing)- which just means that the algorithm will simulate whether some other unexplored node (choice) is worth making.

The implementation of this algorithm for Stupide Vautour doesn't really work as one might imagine. The reason for this is that, player's do not have all the available information for all the variables like they do in a game of chess where both players know exactly where every game piece is on the game board and where the pieces may go. In Stupide Vautour we do not know what prize will be drawn the next turn, however we technically know all the possible plays that can be made if one was to track what cards each player has already played and what prizes have already been drawn. This makes it so we can not create a search tree that determines the best move based on a weighted win ratio because prize X doesn't always lead to prize Y, and vice versa. Every new game begins with a shuffled deck of 15 prizes that does not change for the duration of the game and thus every game has exactly 15! ways that prizes can be drawn (fifteen factorial is 1.3 followed by 12 zeros- that is more than a trillion different ways). Additionally, each player has 15 'money' cards and thus 15! possible combinations of choices they can make each game. The more players that are added to the game, the more variability is added and the more data the A.I. has to keep track of to make an appropriate choice. As a result a two player game has 15!*15!*15! (2.24 followed by 36 zeros) ways of playing out.

Name	# of Zero's	# of group's of 3 zero's
Trillion	12	4 (1,000,000,000,000)
Quadrillion	15	5
Quintillion	18	6
Sextillion	21	7
Septillion	24	8
Octillion	27	9
Nonillion	30	10
Decillion	33	11
Undecillion	36	12

For perspective, a high-end GPU (Nvidia GTX 1080) is capable of 10 teraflops. 10 teraflops means it can do 10 trillion floating point operations a second (addition, subtraction, multiplication, division). While my 2017 Macbook Pro's Intel Iris Plus 650 GPU can only perform around 58 gigaflops at it's base frequency (300 MHz). If we assume each simulation for a game state was just a floating point operation and it consisted of 2 players, and my A.I. could use the power of the Nvidia GTX 1080 to do these simulations:

2.24 undecillion / 10 trillion = 2.24 x 10^{23} seconds which is equivalent to around 7.1 x 10^{15} years.

My point is that it isn’t currently feasible to simulate each and every possible move for every given turn and so we must turn to alternatives like performing simulations with locally available data. By simulating with locally available data, I mean try to answer questions like “what might happen if the A.I. was to acquire the currently drawn prize, based on all the cards remaining” instead of “what might happen if the A.I. acquires the current prize by choosing X card and the opponent chooses Y card, and the next prize prize is Z. To make this even more hellish, we could begin to assign each question with possible player scores ie. “… when my score is 5 and the opponents score is 2”. You might imagine how this would lead to a recursive nightmare that would takes ages to determine because of how unfathomably massive of a search tree this would result in.

So like I hinted, after a lot of trial and error - I implemented a montecarlo search algorithm that asked the question: “what would happen if the A.I. played X card in their hand for the current prize based on the current remaining prizes, and each players remaining cards and their scores?”. The question is answered by playing a set of thousands of simulated games for each X card in the A.I.’s hand. X card is selected to be the A.I.’s first move in all of these simulated games to determine the chance of winning if it is played to acquire the current prize. The rest of the simulated game is then played out by all simulated players choosing random cards in their hands. A ratio of simulated game wins verses number of simulations played out for each card in the A.I.’s hand is deteremined and then pushed to a card win ratio array. The card with the highest win ratio in the array is selected to be the A.I.’s chosen card for the game.

Now the obvious issues with this method is that this will take a toll on performance. When I run the game on my machine, the A.I. usually takes a couple seconds to “think” but in a turn based strategy game like this, that is of course forgivable but if we begin to apply these concepts for where every split second is important such as a FPS game - it would be advisable to go a different route.

So you might be asking if the performance hit of all these simulations is worth it. Is the A.I. relatively actually smarter? The objective answer is: yes. Just imagine playing this game with your friends around a table and being able to keep track of all the cards they have played and the prizes that remain, while simultaneously being able to determine the mathematically optimal card to play by determining the chance each card in your hand has to win the game in the current state of the game. All by simulating what would happen if you played each card in your hand relative to the cards available in your opponents hands a thousand times.

The reason we have to simulate so many times is because each simulation is essentially played out randomly not knowing anything about the opponent other than the cards they have in their hand and of course their score. So to get the most accurate win ratio for each card, we must test it against our opponents cards as many times as possible without taking an absurd amount of time. Because like I mentioned before, we know the game can play out in an unfathomable number of different ways and thus we cannot feasibly test every possibility as we are restrained by our limited computation power.

Now to address this limitation and more importantly make the A.I. actually take a more holistic approach to determining it’s choices that would help taking into account that it's playing against a human. A human that may choose to employ strategies it is unaware of. I also implemented what I call "player models". Player models are essentially a record of all the choices made by the opponents the A.I. has played against. The record keeps track of all game variables such as the player's current score, the total prize value, the player's hand, the card they chose for in that state, the other player's variables etc… Just a cluster of data associated with each player's name. The reason for collecting this data is so when the A.I. play's against them again, it can predict what card they might choose when it finds their opponent in similar conditions to what it "recalls" from the data that it collected previously.

The current implementation essentially breaks apart the collected data in a JSON file and assigns to the appropriate player. When the A.I. wants to predict their probable choice it just essentially uses a fuzzy string similarity algorithmn. The prediction value is chosen based on the record with the highest similarity score to the current game state. Though I have noticed this approach is prone to errors and I would have to create a more detailed similarity algorithm that takes into account actual values and their relation to the game rather than their shallow string values. Regardless, once a predicted value was spat out, the A.I. would look through the rules I had provided for it and then would return the appropriate best choice.

I thought of this possible improvement to the montecarlo A.I. too late. The possible improvement being, that to make the choice more accurate, instead of testing the A.I.'s cards against it's opponent's completely randomly - I weigh the possibility of the opponent playing certain cards higher and essentially test the A.I.'s cards against a skewed random distribution of player cards. In other words, the simulations will happen more often against the cards that the player may more likely play. ‘A controlled randomness’ perse that would help cater and improve the A.I.’s results against the person it is playing against.

— CONCLUSION —

The biggest issues I’ve faced while doing this project were a whole lot of Javascript quirks; many of them to do with it being mainly asynchronous. Before this project I had no idea what it even really meant for a program to run in an asynchronous fashion - and I still am grasping my head around. For this reason I wasted countless nights smacking my head agains the table and make snail like progress and what hurt the most - I wasn’t able to get the GUI made. Regardless, I’m surprised I’ve got my code working at all, considering the the mess it currently is. I must say I have learned my lesson and now I am foolishly trying my hand at another project in the same language but much more cautiously. In terms of the actual A.I. itself, I’m proud I actually got the prediction part working, it was a pain trying to figure out how to output information to a JSON file- which would have been a breeze if I had just stuck with doing this all in Java. In the future I definitely want to interconnect the prediction and montecarlo. I’m certain it has high chance of being a success.

Well, that’s all. Thank you for reading and be sure to check out my code. To play against my A.I., make sure you have node.js installed and then cd into server/src/ and run “node runGame.js”. Additionally, if you would like to see the montecarlo algorithm and prediction in play - comment out the two lines as indicated as mentioned in “prediction.js”.