COMP30024 Artificial Intelligence
Project Part B - Report

Team: AlphaChexers
Authors:
Ben Mann, 911733
Simon Jerome Han, 912108

Outline
Welcome to our submission for part B of the COMP30024 project. This report consists of three core sections:

Section A - an introduction to MCTS. Our main bots rely heavily on Monte Carlo Tree Search, which isn't discussed in the lectures, so we provide a brief overview of the algorithm here.

Section B - MCTS with CNNs. The bulk of our time was spent attempting to write a bot that could self learn the game of Chexers by using the same algorithm as Deepmind's *AlphaZero*. Spoiler alert: we failed, but we have included our analysis of this experience anyway because a) it helped inform our final bot and b) it took up the bulk of our time.

Section C - MCTS with handwritten heuristics. This is a discussion and description of our final bot, *AlphaChexers* (but sometimes referenced in this report as *AlphaMCTS*, which was its development name) which was hacked together using modules from section B and other simpler algorithms.

Package structure
The *AlphaChexers* package consists of the following core modules:

1. AlphaRandom
   AlphaRandom makes random legal moves.
2. AlphaGreedy
   AlphaGreedy captures at every opportunity and moves its counters forward when there are no captures available.
3. AlphaMaxN
   AlphaMaxN uses the MaxN algorithm to search for optimal moves. This implementation can search up to a depth of 3 within about two seconds. It was eventually abandoned as we struggled to effectively prune the search tree.
4. AlphaOneB
   AlphaOne is one of our main modules. Inspired by Deepmind's AlphaZero, AlphaOne adapts the same algorithm to attempt (emphasis here is on the word attempt) to learn the game of Chexers *tabula rasa*.
5. AlphaChexers

Also known as AlphaMCTS, AlphaChexers is our final and strongest bot, combining all of the lessons that we learned from implementing AlphaMaxN and AlphaOne. It is powered by AlphaOne's MCTS module and AlphaMaxN's heuristic functions, and has some extra little heuristics built on top of it.

Section A - Introduction to Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a popular search algorithm amongst the AI community. While algorithms such as Minimax and MaxN search the sample space using a layer by layer methodology, MCTS focuses on subtrees that are 'interesting' before those that are 'uninteresting'. This style of asymmetric search results in an algorithm that is more intuitive and perhaps more 'human' than the brute force style search strategies of Minimax and MaxN, and allows it to adapt to limitations in time and computing power extremely flexibly. Because the algorithm outputs the best result given its limitations, and because its search process can be terminated at any time, it is an ideal candidate for programs that must run under strict time and compute restrictions. Given the restrictions placed upon our Chexers agents in this project, it is thus also an ideal candidate for our own program.

How it works

The implementation of MCTS is surprisingly simple. First, the algorithm starts with the root node of the search tree as its 'current' node. While the 'current' node has child nodes, the algorithm picks the child node that maximises a node evaluation function (to be discussed later) and traverses to that node, setting it as its new 'current' node. When the algorithm has reached a leaf node (ie. a node with no children), it evaluates the strength of the 'current' node and expands it by initialising its child nodes. The algorithm then starts again from step one, and this entire process is repeated until either a time limit has been reached or the search results have stabilised. Of the root node's children, the one with the highest visit count is then outputted as the optimal next node to move to.

Node evaluation

A core part of the MCTS algorithm is its ability to explore 'interesting' subtrees before 'uninteresting' ones. This is achieved through its use of an evaluation function to determine node traversal order. The function used in most MCTS implementations is the UCB1 function:

$$UCB1(X) = X.Q + c*sqrt(log(X.P.N)/X.N))$$

In the above formula, 'X' refers to the node that is being evaluated, 'X.P' refers to its parent node, '.Q' refers to the 'strength' of a given node, and '.N' refers to the visit count of a given node. 'c' is an adjustable parameter that is typically set at around sqrt(2). Higher values of X.Q will typically lead to higher values overall, but if X.N is too low in comparison to X.P.N, it will

result in the node's overall score becoming higher as well. Thus, UCB1 balances 'exploitation', represented by the first term, and 'exploration', represented by the second term.

Node 'strength'

The accuracy of node evaluation depends on finding accurate estimations of 'X.Q'. In classical MCTS, 'X.Q' is estimated through a process known as rollout, where the algorithm plays a random game upon reaching each leaf node. When using rollout, 'X.Q' is the proportion of the number of wins achieved by a node 'X' and all of its child nodes compared to the number of times it has been visited. While this allows MCTS algorithms to search the sample space without the need for complex gamestate evaluation functions, it can be highly inaccurate because of the high branching factors of most games, and thus negates most of the algorithm's flexibility with regards to time and compute flexibility. This shortcoming can be overcome by replacing rollout with other methods to estimate node strength - for complex games such as *Go*, advanced techniques such as the use of deep CNNs can be used, and for relatively simple games such as *Chexers*, a handwritten linear evaluation function is enough to suffice in most cases.

Section B - *AlphaOne*

MCTS with CNNs

This section will document our attempt and failure at building a self-learned Chexers player using MCTS paired with a Convolutional Neural Network (CNN), à la Deepmind's *AlphaZero*. We'll avoid discussing some of the nitty gritty details of the algorithm for the sake of brevity (how CNNs work, network architecture etc.), instead focusing on the overall idea behind *AlphaZero* and how we tried to replicate it with *AlphaOne*. While what's written here isn't strictly relevant to our final Chexers agent, we have decided to include our analysis anyways because this attempt helped inform our final implementation, and also because it's where the bulk of our time was spent (our final bot was hacked together in the span of about two days after we realised that this wasn't going to work out on time...).

Plus, Deepmind's *AlphaZero* is friggin sweet yo.

How does *AlphaZero/One* work?

In this section we'll describe the overall picture of how Deepmind's *AlphaZero* works. Our implementation, *AlphaOne*, relies on the exact same strategy as *AlphaZero* (or at least how we understood it from Deepmind's papers), and thus in the following section the terms *AlphaZero* and *AlphaOne* can be used interchangeably.

AlphaZero uses convolutional neural networks (CNNs) paired with MCTS to learn and generate powerful moves for static, discrete, fully observable, deterministic, sequential games. While

CNNs are incredible at detecting patterns in visual data, training them through self play is extremely unstable as it is difficult to avoid arriving at odd points of local maxima. The core idea of *AlphaZero* is to overcome this problem by using MCTS as a 'policy improvement operator' that guides and stabilises the CNN, thus allowing the program to train through self play.

The algorithm behind *AlphaZero* is surprisingly simple and elegant. The central idea behind it is that even when the initial net has randomised weights, search results obtained from MCTS combined with the CNN will always be better than those obtained from the CNN on its own. Thus, if the CNN is trained to fit to the MCTS results, it will gradually improve in a stable fashion as the average performance of both the CNN and the CNN paired with MCTS rise, with the latter being akin to a sort of 'upper bar' of performance that can be perpetually pursued by the former. Over millions of games, it becomes possible for the CNN to be trained to a degree where it can predict, generate and even innovate incredible new moves.

Algorithmically, the training process for *AlphaZero* is relatively straightforward. A CNN is initialised such that it takes the board state as input, and outputs two values - *p*, a vector containing a probability distribution that covers all possible moves, where a higher value for a certain move indicates that it has a higher chance of leading to victory, and *v,* a scalar value between -1 and 1 that represents the current likelihood of the player winning given the board state (-1 means loss, 1 means win, 0 means draw). Each MCTS search outputs another vector π, which is similar to *p* except that it contains the probabilities generated by MCTS instead of the raw CNN. To train, *AlphaZero* continuously plays games against itself, and with every move that it makes, it saves the current game state as well as the calculated value of π for that gamestate. Upon the conclusion of the game, the CNN is trained such that for every gamestate the difference between π and p is minimised, and the similarity between z and r (the final result of the game, 1 for win, -1 for loss, 0 for draw) is maximised. This process of game playing and training is repeated until convergence or a set time limit is reached.

Our modifications for *AlphaOne*

Despite the fact that it would be the cause of our eventual failure, we recognised from the very beginning that limitations with regard to compute power and compute time (for the final dimefox version) would be significant challenges to the viability of our program. To attempt to deal with these limitations, we made two main modifications to the network architecture:

1. *AlphaZero*'s network uses forty residual layers. This is ridiculous. We cut our network down to five, both for the sake of training (increased chance of convergence) and game playing (so it could actually maybe play a game within 60s)
2. *AlphaZero*'s input for its *Go* playing variation is a 19x19x17 tensor, which we can think of as a stack of seventeen 19x19 matrices. The first eight matrices represent the current and last seven boardstates from one of the player's perspectives, where a value of 0 in the matrix indicates that they didn't have a counter placed at a particular coordinate, and a value of 1 indicates that they did. The next eight represent the current and last seven

boardstates in the same fashion for the other player. The last matrix indicates which player's turn it is (all 0 for the first player, all 1 for the second player). The Deepmind team's reddit Q&A revealed that the purpose of including game history within the input tensor is to create some sort of 'attention mechanism', but we decided that this was adding too much complexity to our own implementation, so we cut it out. In any case, having an 'attention mechanism' seemed to be more important for games like *Go*, where counters can be placed anywhere and so the player needs to keep track of where the other player is focusing on, than games like *Chexers*, where counter placement is restricted to just about ten to twenty coordinates in the first place. Our final input was a 7x7x6 tensor, with the first three 7x7 matrices being the current boardstate from the perspective of each of the three players (using the same representation strategy as *AlphaZero*), and the last three being the number of exits made by each of the three players. In an effort to increase the chance at convergence, the boardstate was 'rotated' before being inputted into the CNN so that the current player's starting position was always the bottom left corner, regardless of if they were red, green or blue.

Training and results

To properly train our network we used Ben's desktop PC, which is equipped with an i5 CPU, 16GB of RAM of memory and an NVIDIA GTX1060. The CNN itself was implemented using Keras. After several days of debugging (which took far longer than expected because the program would sometimes run for hours before breaking), we finally managed to get our trainer program to run smoothly without errors. We left it to train for three nights, during which the program manage to play about 250 games against itself. For the first 100 games we used MCTS with only 100 simulations each, and for the rest we used MCTS with only 500 simulations each. This was done for the sake of time, as we realised during debugging that CNN powered MCTS was quite slow on our mediocre (in comparison to what the Deepmind team used, more on that later) hardware.

250 games using less than 500 simulations per move was clearly far from enough, and our program failed to learn much during this time. At the time we were somewhat inclined to observe that the program had started to learn to capture, but retrospectively this was quite possibly the placebo effect (or some equivalent phenomenon) in action.

Why did we fail?

According to Deepmind, *AlphaZero* went from zero (random moves) to hero (world champion vs both humans and other ai's) within a couple days of training. A major caveat to this is the fact that the team was using 4 TPUs to generate a single game, and were running 5000 TPUs in parallel to generate games for the overall training process. In addition to this, they had another 64 TPUs that they were using to train the CNN. This setup allowed them to generate 1,250 games (5000/4) every iteration, to generate new moves using MCTS with 1600 simulations in 0.4s, and to train the CNN on batch sizes in the hundred of thousands. By the time *AlphaZero*

was a world champion it had played almost five million games against itself. Retrospectively, the fact that we still decided to pursue this approach despite this massive caveat is somewhat embarrassing, and a true testament to our amateurism in this field. In the end, Ben's PC was simply not strong enough for the task we had in mind.

Beyond complaining about how we don't have access to Google levels of hardware, however, we can still reflect on our experience and look at other reasons for our failure. Firstly, our limited compute power had implications beyond that of making it impossible for our program to converge within a few centuries, as it made debugging extremely difficult as well. To account for this we should have been more rigorous in testing our helper functions independently before running the complete program. This would have bought us at least another week of time. In addition, we should have done more to simplify and abstract our problem - one possibility is that we could have abandoned the convolutions entirely, and instead used a standard (and shallower) neural net to only estimate *z*, perhaps even using a hand picked feature set instead of the overall boardstate as input. This might have made the program much faster. In this way, our over-reliance on the *AlphaZero* paper might have ultimately proved to be a hindrance to our success. Finally, we probably didn't utilise Ben's PC to its full capacity in the first place. More knowledge and expertise regarding leveraging computer hardware (CPU and GPU) to support software would have been useful here.

Section C - *AlphaChexers/AlphaMCTS*

MCTS with custom heuristics

Even though our *AlphaOne* implementation failed to yield any tangible results, we quickly realised upon shifting our focus back to traditional search/evaluation algorithms that we could reuse much of the infrastructure that we developed for *AlphaOne*. In particular, it became apparent that if we let go of our desire to build a self learned Chexers player, we could create a decent bot by using *AlphaOne's* MCTS module, and by replacing its CNN with a selection of hand written evaluation functions. This insight led to the creation of *AlphaMCTS*, our final and strongest bot. It doesn't achieve perfect play as it still makes some illogical moves at times, but for the most part we think that it is quite ok, especially given the fact that the vast majority of our time and efforts were invested into the development of *AlphaOne*.

Core modules/classes used for *AlphaMCTS*

1. player.py - this file contains the player itself, which is built on the template released by Matt as a part of the project B package. It references other modules for its decision making process, but has an additional timer function built in that lets it calculate how much time to allocate to the next move search.

2. mcts.py - this file was ripped straight from our *AlphaOne* module and only modified slightly. It contains our core MCTS algorithm and is the bot's primary decision making apparatus
3. boardstate.py - this file contains the BoardState class, which stores node data used for MCTS.
4. chexers.py - this file contains a bunch of helper functions that we developed for the game of chexers (eg. find all available moves, encode coordinates, find distances etc.).
5. puct.py - this file contains the node evaluation function that we used for MCTS

How it works

Like *AlphaOne*, *AlphaMCTS* relies on MCTS to search for moves. However, instead of relying on a self-learned CNN to evaluate board states, it refers to a handwritten board evaluation function and a small set of additional if/else heuristics that to encourage it to make the best decisions.

Board evaluation function

We spent some time experimenting with various board evaluation functions and feature sets, but ultimately decided that simpler was better. *AlphaMCTS* uses a simple linear function to determine the strength of each board state, which is as follows:

$$Eval(Player) = 10*numExits(Player) + \Sigma(1+k*distanceToObjective(counter)+m*f)$$

Here, the second term refers to the sum of scores for each individual counter. 'Objective' refers to where we want to encourage each individual counter to head - usually this is the player's exit point, but at times it may be an opposing player's exit point (eg. if they are closer to winning than we are and we need to block them) or even the center of the board (eg. if we have less than four counters on the board and limited options). 'Objective' is controlled by if/else logic that is in turn determined by factors such as how many counters are on the board, how many exits each player has made and how far each player is from winning. 'k' is an adjustable constant, typically set at around 0.05, that manages how important it is for the counter to move towards its objective (instead of, say, moving backwards for a capture). 'f' refers to the number of friendly counters that are immediately adjacent to the counter in question, and 'm' is another adjustable constant that determines how important we consider 'f' to be. It is typically set at 0.005.

The evaluation function that we have rewards exiting above all else. This leads to illogical moves where the bot will exit counters even if it doesn't have enough counters on the board to make four exits. To temper this, we added another layer of if/else logic to the move search function so that exiting isn't even a considered option when this is the case.

Node evaluation function

AlphaMCTS uses a slightly modified version of UCB1 for node evaluation that resembles somewhat of a cross between UCB1 and PUCT (which is what *AlphaZero* uses). It is designed to encourage deeper searches. We'll call it PUCT1, and it is as follows:

$$PUCT1(X) = X.Q + c*1/(X.N + 1)$$

'Q' and 'N' refer to each node's strength value and visit count respectively. We set 'c' to be sqrt(1.6) because this seemed to allow us to search at slightly greater depths and thus return generally better results. However, we didn't test these values exhaustively, so there is certainly a chance (it is even likely) that a more optimal value exists.

Pruning

Pruning in multiplayer games is extremely difficult. *AlphaMCTS* manages to prune a part of the search space by only considering moves made by opposing counters that are within two steps of one of its own counters. This allows it to search a little deeper - up to a depth of around five on average. The intuition behind this methodology is that the algorithm should focus on local, immediate threats, searching deeply to determine its best course of action against them, while saving time by neglecting opposing counters that are too far away from us for them to be a threat, or for us to threaten them. We use the word 'neglect' to somewhat hesitantly describe this pruning strategy, as it is quite possible that it has implications on *AlphaMCTS'* ability to recognise long run strategies. Nevertheless, without it *AlphaMCTS* only searches up to a depth of four at most, so we decided to keep it anyway as *Chexers* is mostly a greedy style game in the first place.

Time management

As alluded to earlier, player.py also contains a function for changing the time allocated to the calculations for each move to ensure we maximised the amount of time used while also not going over this limit. This function works by first calculating a base time that each move can take based on the maximum number of moves (256) and the maximum time (60 seconds) allowed in the game, and then roughly determining whether the game is in the early, middle, end or won stage and multiplying the base time by some constant factor accordingly. This base factor was determined somewhat arbitrarily based on assumptions such as that the middle of the game is where the most calculation time is needed compared to the very early or very late game. While determining whether a board is in a particular stage is also only an estimate based on the number of counters each player has and the distance the counters are from their goals, after some experimentation we feel that this function adequately allocates time for each move.

Effectiveness

AlphaMCTS handily beats *AlphaRandom* and plays strongly against *AlphaGreedy* as well. At times it is hard to assess exactly how much better it is than the greedy bot as both are still susceptible to repeated loops, but it is certainly better. Somewhat amusingly, *AlphaMCTS* tends to have more initial losses to the greedy player than it does to stronger players on the battlegrounds server. Our explanation for this is that the greedy player is more unpredictable than logic-based programs, and thus can occasionally place *AlphaMCTS* in awkward positions. Nevertheless, *AlphaMCTS* tends to win in the long run against these algorithmically simpler programs.

References

Silver, D., Schrittweiser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., … Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature, 550*, 354–359

Silver, D., Hubert, T., Schrittweiser, J., Antonoglou, I., Lai, M., Guez, A., … Hassabis, D. (2017, December). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science,* 1140-1144