COMP30024 Artificial Intelligence

Project Part B Report

Jiayi Xu 1165986

Un Leng Kam 1178863

1 Approach to playing Cachex

Our group believes that we humans are unable to create a convective evaluation function that can fully address any board situation, thus, we rely on the ability of machines being able to explore multiple positions and further down the game to create an evaluation function that is based on the states of the game.

1.1 Monte Carlo Tree Search (MCTS)

MCTS is a tree search algorithm that adopts the Monte Carlo method which uses random sampling for deterministic problems where other approaches pose a challenge in solving the problem. Nodes represent the state of the game, and the edges represent a move. The algorithm can be decomposed into four steps:

- *Selection*: From the current state of the board represented as the root node, the algorithm traverse successive child nodes until a leaf node is reached. The leaf node is a node that has a potential child node where no simulation has been initiated.
- *Expansion*: Given that the leaf node does not end the game immediately (win/loss/draw), create a child node that is any valid moves from the game state that are defined by the leaf node.
- *Simulation* / *Rollout*: Using the child node as the beginning game state, a series of random moves are taken until the game is ended.
- *Backpropagation*: The result from the simulation (win/loss/draw) is then returned to the root node.

Such a process is then repeated according to the restrictions, and then the child node with the most simulation or visited is chosen as the move to be played.

1.1.1 Why Monte Carlo Tree Search?

Depending on the board size of the game there are at most $n \times n$ moves, therefore assuming that on average there are $\frac{n \times n}{2}$ moves. If we use minimax search, both time and space complexity can go up to $O\left(\left(\frac{n \times n}{2}\right)^m\right)$ and $O\left(\frac{n \times n}{2} m\right)$ where m is the max depth of the tree and $\frac{n \times n}{2}$ is the branching factor. With the resource limits and the capture rule, the search can dive into infinite recursion, thus, the traditional minimax algorithm wouldn't be able to search to a deep depth. Hence, being unable to explore moves that are further in-depth which may or may not be better than the current chosen move. Action pruning or reducing cut-off depth can be implemented, to allow a better search time and space usage, but as mentioned before, reducing cut-off depth may enhance the horizon effect. Action pruning may hinder the performance of the agent, actions that are pruned may be optimal under the current game state since action pruning is based on the prior knowledge of the person implementing it. Furthermore, minimax requires a heuristic evaluation function implemented by the user, again prior knowledge is required which our team believe is useful but limited in such scenarios, as it is impossible for humans to consider every single position in Cachex.

Under such usage constraints and our group's belief, we decided to research further search algorithms, and we came upon MCTS. After understanding MCTS, we believe that it's suitable and effective in creating a high-level Cachex playing agent. MCTS is effective in the sense that it is a probabilistic and heuristic driven search algorithm that allows an individual with no heuristic domain knowledge to be able to use it, MCTS can make informative evaluations just from the random rollout phase. As MCTS utilises the Monte Carlo method which exploits the law of large numbers, if an experiment is simulated many times, the average should converge towards the expected outcome. Thus, with enough simulations, MCTS can search for the move with the greatest expected outcome for our agent under the current game state. As mentioned before, the horizon effect is always present in tree search problems, MCTS becomes useful as it continues to evaluate other alternatives when a perceived optimal is chosen. We believe such "randomness" in not picking the perceived optimal move is strategical and successful in playing Cachex, as we conclude that the general human perceived game strategy provides bias in looking at capturing squares or path constructing, thus, the created evaluation function contains features that may or may not be relevant towards the ending result of the game for different size of n.

1.1.2 Upper Confidence bound applied to Trees (UCT)

The selection process of child nodes to expand during the simulation has a flaw, the algorithm may favour a losing move due to a majority of losing counters by its opponent, but there are also winning counters by opponents. However, due to the vast majority of losing moves by its opponent, such move is chosen, thus, assuming the opponent is playing to win, it becomes losing for our game playing agent. Furthermore, how are the selection of nodes balanced between exploration and exploitation? We do not want a node that is winning for the agent to be disregarded and the algorithm explores other nodes. Hence, UCT is implemented to guide the selection of nodes, the formula is:

$$UCT_i = X_i + C\sqrt{\frac{\ln(N)}{n_i}}$$

Where X_i is the win ratio of the child node, N is the number of visits of the parent node, C is a constant to adjust the amount of exploration, and n_i is the number of visits of the child node. X_i is the component that corresponds to exploitation, as it favours moves with a higher win-ratio. $C\sqrt{\frac{\ln(N)}{n_i}}$ is the component that corresponds to exploration since it favours moves with few explorations. At the beginning of the MCTS, it focuses on exploring all moves but as it explores more and more game states, it focuses on choosing moves that have a better reward as the left component of UCT becomes larger. Thus, we are able to mitigate the exploration-exploitation trade-off and the flaw present in our MCTS algorithm.

1.1.3 Problem with Monte Carlo Tree Search with UCT

With enough simulations, MCTS will be able to explore important game positions and determine their expected outcome by the Monte Carlo method. But given the usage constraint, a higher number of simulations is infeasible for playing the game of Cachex. The problem of MCTS requiring a high number of simulations is mainly due to UCT, as UCT treats positions as random playouts that could lead to wins or losses, the position's strategical meaning is overlooked. Such as the diamond formation of Cachex that could lead to capture, creating a vertical diamond or horizontal diamond is evaluated the same for us humans, however, in MCTS it will require to explore the full game state of a vertical diamond and go on to explore the full game state of a horizontal diamond even if the previous result shows an advantage of creating a diamond for capturing. We could implement a hash

table that stores every game state, thus, improving the search time but this will hinder our space complexity. Furthermore, the limit on the number of turns induces a problem when the board size is large, the rollout phase could exceed the number of turns (343 turns) and thus result in a draw as the moves played are completely random. This hinders the ability of the MCTS algorithm to evaluate the game state's expected outcome, hence, reducing the skill level of our Cachex agent.

1.2 Alpha Zero's approach

Realising the constraints present by MCTS, we have adopted the approach taken by Alpha Zero. Through the combination of MCTS and deep convolution neural networks (DCNN), Alpha Zero is able to learn to play a deterministic zero-sum game through self-playing. Such an approach is taken rather than Alpha Go's approach due to the benefits of self-playing, we do not have to create a large dataset of experts playing Cachex (Also unable to do so, because our team is not good at this game), thus, saving us time. Alpha Zero improves by creating training datasets through self-playing and creating a policy-value network from such datasets that is able to evaluate moves and return the optimal one.

1.2.1 Adjusted Monte Carlo Tree Search

In the adjusted MCTS, nodes represent the state of the game, and the edges represent a move. The algorithm can be decomposed into four steps:

- *Selection*: Beginning at the root node *s*, for every child node that is already expanded in the Expand and Evaluate step, a valid action *a* is selected according to the highest *U(s, a)* which is calculated using the polynomial UCT. Such a process ends if the simulation encounters a leaf node.

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\mathrm{N(s)}}}{1 + N(s,a)}$$

  c_{puct} is a hyperparameter to determine the level of exploration, *Q(s, a)* is the action-value function that measures the value of a move, *N* is the visit counts of the node, and *P(s, a)* is the estimate of how "good" this action is, returned by DCNN. If a leaf node is found, the Explore and Evaluate step is called.
- *Explore and Evaluate*: Given the current node is a leaf node, the current state with the representation (explained in 1.2.2.1) is inputted into the neural network, where a vector of probability distribution *p* between [0,1] is returned, explaining how "good" the valid actions are under the current state, and *v* is a score between [-1,1] that is an estimate evaluation of the current state. Then, the Leaf node stored its value as *v* and expanded where child nodes are added, and each stored its prior *p* respectively according to the return *p* by DCNN.
- *Backup*: After we expand the leaf node, the parameters will be updated in a backward pass to all of the parent's nodes until the root node. Updating the visit counts and the action-value function which is the sum of *v* of the nodes below the current node divided by the total number of visits of the current node.

This process is iterated until the maximum number of simulations is reached, where the number of simulations depends on the parameter *n_playout*, and *play* (*get_action()*) is called.

- *Play*: At the end of the search, it is time to select the action *a* in the root node *s* based on the parameter updated at simulations. In competitive play, the action with the greatest visit *N* is chosen, but during self-play, the policy π is adjusted according to:

$$\pi \sim N^{\frac{1}{\tau}}$$

  Where τ is a temperature that is used to control the degree of exploration and the magnitude is infinitesimal.

1.2.2 Creating training datasets using Monte Carlo Tree Search

As we begin a Cachex game, let s1 be a representation of the current empty board state. We can run simulations within the MCTS algorithm to formulate a policy π1 where a move a1 is returned. The game continues by the agent taking the move a1, which creates the new game state s2. This process is then repeated until the game is ended at which we determine who won and represent it as z where z is (-1,0,1). Such a self-played game creates a sequence of input board position, policy and game result.

$$(s_1 \rightarrow (\pi_1, z) \rightarrow\ s_2 \rightarrow (\pi_2, z) \rightarrow s_3 \rightarrow (\pi_3, z) \rightarrow \ldots s_T \rightarrow (\pi_T, z))$$

Note that the z in the sequence is stored according to "is he the winner" e.g. (1, -1, 1, -1….), and s is stored according to the entry's player "perspective". Around thousands of games are played on average, and a large training dataset is therefore generated where samples are randomly selected and fed into DCNN. However, due to the range of size of Cachex board, training data is created specifically for each size of the board.
Under self-training $P(s, a)$ in PUCT is altered by adding Dirichlet noise to ensure the diversity in game states provided to DCNN as we tend to explore all moves during the root node, we want a policy-value network that under all circumstances is able to accurately evaluate the state.

$$P(s, a)\ =\ (1 - \varepsilon)P_a + \varepsilon\eta_a$$

η is probability distribution by using Dirichlet noise with the chosen parameter η as 0.3 and ε as 0.25. To compensate for the lack of computer resources, we also "rotated" the game board 180 degrees which creates a new game board. This allows more training data to be generated for DCNN.

1.2.3 How a game state is represented

The game state is converted into four feature planes of binary 2D arrays, the first two planes represent the current player's stone position and the opponent's stone position respectively. The position with a stone is 1, and the position without a stone is 0. The third plane indicates the position of the opponent's latest move, only one position is 1 and the rest are all 0 in the plane. The fourth plane indicates whether the current player is the first player. If it is the first player, the entire plane is all 1, otherwise, it is all 0. The last two planes were included as a feature as we believe that there is a first-mover advantage and usually stones are placed close to each other between players, also that the target goal of winning is based on whether the player first-moves or second moves.

1.2.4 Training the Neural Network

In our implementation of the neural network, the network structure was greatly simplified due to computational restrictions. It was a 3-layer convolutional network, using 32, 64 and 128 3×3 filters respectively, and using the ReLu activation function. Then it is divided into two outputs, policy and value. On the policy *p* side, we used four 1×1 filters to reduce the dimension, then connect them to a fully connected layer, and use the SoftMax nonlinear function to directly output the probability of each valid move that is available for the current game state. On the value side, we first use two 1×1 filters for dimensionality reduction, then connect to a fully connected layer of 64 neurons, and another fully connected layer which uses the tanh nonlinear function to directly output the evaluation score between [-1, 1]. Thus, the depth of the neural network is only 6 which allows a quicker training and prediction time.
The goal of our training is to make the action probability *p* output by the strategy value network closer to the policy π so that the situation score *v* output by the strategy value network can be more accurate in predicting the true game outcome *z*. Thus, to optimise such a goal we are looking to minimise the loss function:

$$l\ =\ (z - v)^2\ -\ \pi^T \log(p) + c\left|\left|\theta\right|\right|$$

Where, $(z-u)^2$ is the mean square error between our value function estimation v and the true label z. $\pi^T \log(p)$ is the cross-entropy for the estimated policy p and the policy π, and $||\theta||$ is an ADAM optimizer with the defined learning rate *kl* which controls the degree of learning when training every batch of data. To evaluate the performance of our agent during training, for every 50 self-played games, we have our model-based agent play against a pure MCTS agent that has been simulated 100 times, the model is updated and taken into use if it gets a higher win ratio from playing with the pure MCTS agent. If the win ratio reaches 1.0, we increase the simulation times of the pure MCTS agent. However, such an approach is used only when the board size is less than or equal to 8. With a larger board size, the pure agent is unable to roll out as there is a turn limit which leads to moving that leads to a draw, similarly to Alphazero's approach. Hence, after the second self-play game, there will be two models available which are put into play and whoever wins is used to generate a new game and the process continue, until our model converges.

1.3 Playing Cachex

When playing Cachex using Alphazero's approach, our AI has made moves that seemed like a blunder yet strategical like the "poison pawn" in chess. Such moves require significant calculations but are usually the winning moves that pivot the game towards our agent, however, this is also reflected in real-time. The running time has exceeded the time constraint, and thus, we had to divert into a more time-saving strategy. For board sizes less than or equal to 8 models that we have trained, we incorporate them into the pure MTCS agent where during the roll-out phase, it does not do random moves but is guided by our model which creates a search tree that focuses on optimal moves that are learned by our model. For board sizes greater than 8, we resolve by using our original train models, however, the training time for board size greater than 8 took too long, and it took us 2 days to train a decent player for size 8. Hence, we apply these "under-train" models and used them to play board sizes greater than 8. We also implemented a fail-safe measure that when the running time of our program gets closed to the running restrictions, in the rest of our game we will do random moves and hope to win luckily.

2 Past Idea

Below is an idea that our group thought of and roughly implemented but was disregarded based on our group's strategy and usage constraints.

2.1 Generating weights for our evaluation function using stochastic gradient descent

Features in our evaluation function include:

- Distance of stone towards the ending sides of the board (up and down for red, left and right for blue).
- Distance of stone toward the centre of the board.
- Given the distance of one, how many opponent stones.
- Given the distance of one, how many stones are placed by us.
- Given the distance of one, how many empty hexagons.
- Given the current hexagon position, how many diamonds can be formed that lead to us being captured.
- Given the current hexagon position, how many diamonds can be formed that lead to us capturing opponent stones.

A script was written that generate such features as counts, and we inputted our game played move within our group for 5 games of sizes 7, 8 and 9. We label the moves as won, tie or loss as 1, 0, and -1 respectively, inputted them into our script and trained it using the stochastic gradient descent model.

We obtained the weights and computed evaluation scores in different game states created by our group, but seemingly it couldn't return the position that should have been the highest evaluation score. Where we later resort to self-playing using MCTS and allow the AI to learn by itself.

3 Testing

In our script *Train.py,* we have implemented an *evaluate_policy* function which brings a pure MCTS agent and the current best model into playing each other, which helps us in finding out whether the current policy has improved and learn better moves or strategies. Also, due to the similarities of our approaches and the underlying algorithm we are able to create three agents that are *pure* MCTS which runs on the pure MCTS algorithm where the simulation count indicates the strength of the agent, *Alphazero_player* which uses the pure AlphaZero approaches and returns moves based on the underlying neural network model, and net_pure which combines both the pure MCTS algorithm and incorporates the neural network model in guiding the roll-out phase to obtain expected moves. They are played again each other, and under usage constraints and a strategical approach, net_pure won most of the games.

4 Evaluation

We believed that we are heading in a promising direction for our approach due to the creation of skilful AIs at the board size of less than 7, but however, due to a lack of training time for our AIs of greater board size, they are not performing very well by our own personal judgement. We believe with greater training time; our agents will perform like the well-known AlphaZero in the game Cachex.

References

1 *Alpha Zero And Monte Carlo Tree Search*. Joshvarty.github.io. (2022). Retrieved 10 May 2022, from https://joshvarty.github.io/AlphaZero/.

2 Augmentingcognition.com. (2022). Retrieved 10 May 2022, from http://augmentingcognition.com/assets/Silver2017a.pdf.

3 Bodenstein, S. (2022). *AlphaZero* |. Retrieved 10 May 2022, from https://sebastianbodenstein.net/post/alphazero/.

4 Cheerla, N. (2022). *AlphaZero Explained*. On AI. Retrieved 10 May 2022, from https://nikcheerla.github.io/deeplearningschool/2018/01/01/AlphaZero-Explained/.

5 *GitHub - junxiaosong/AlphaZero_Gomoku: An implementation of the AlphaZero algorithm for Gomoku (also called Gobang or Five in a Row)*. GitHub. (2022). Retrieved 10 May 2022, from https://github.com/junxiaosong/AlphaZero_Gomoku.

6 MiniMax?, W., B, R., B, R., & Soemers, D. (2022). *When should Monte Carlo Tree search be chosen over MiniMax?*. Artificial Intelligence Stack Exchange. Retrieved 10 May 2022, from https://ai.stackexchange.com/questions/20808/when-should-monte-carlo-tree-search-be-chosen-over-minimax.

7 Nast, C. (2022). *How the Artificial Intelligence Program AlphaZero Mastered Its Games*. The New Yorker. Retrieved 10 May 2022, from https://www.newyorker.com/science/elements/how-the-artificial-intelligence-program-alphazero-mastered-its-games.

8 Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., & Guez, A. et al. (2022). *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. arXiv.org. Retrieved 10 May 2022, from https://arxiv.org/abs/1712.01815.

9 *Simple Alpha Zero*. Web.stanford.edu. (2022). Retrieved 10 May 2022, from https://web.stanford.edu/~surag/posts/alphazero.html.

10 *Stochastic Gradient Descent—Clearly Explained !!*. Medium. (2022). Retrieved 10 May 2022, from https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31.

11 *UCT - Chessprogramming wiki*. Chessprogramming.org. (2022). Retrieved 10 May 2022, from https://www.chessprogramming.org/UCT.

12 Medium. 2022. *AlphaGo Zero — a game changer. (How it works?)*. [online] Available at: <https://jonathan-hui.medium.com/alphago-zero-a-game-changer-14ef6e45eba5> [Accessed 10 May 2022].