# ADVERSARIAL SEARCH IN THREE PLAYER CHEXERS

CHUANYUAN LIU, CALVIN HUANG

## MODULE STRUCTURE:

Our Artificial Idiot (aka AI) is composed of 5 components.

1. <Player> module. Player class defines abstractly how action and updates are carried out; translating actions from `referee` representation to Player's representation.
   An additional change we made to the system in player will be translating board `perspective` to `red player perspective` by rotating the board to red for all types of players.
   a. This means if you start with green colour, you will see yourself as red player, but now blue player goes first
   b. The intuition here is to utilise symmetry to reduce the number of possible board configurations. For this will significantly increase the chance for a machine learning based agent to converge to a global optimal. Additionally, this also makes it easier to encode actions.

   player.py file, in addition, also defined a set of wrapper class for different types of agents.
2. <game> module. This module composes of basic game and search setup needed by various algorithms.
   a. Starting with <state>, a state is the most basic unit representation of the game at a given time. It's also responsible for low level translation (positional translation) between different `perspectives` of players.
   b. The <state> is wrapped by <node>, which stores the `game history` or `decision tree` of a game or a search. In addition, various types of nodes are defined here as well by subclassing original nodes to provide additional functionalities to algorithms.
   c. Lastly, <game> stores all game/problem specific rules and can generate actions. Similarly, game is also subclassed like node and state to provide additional feature on request.
3. <search> module. In the search module, various types of search algorithms are defined and tested. Including, <minimax>, <maxN>, <UCT> (Upper confidence monte carlo tree search), <tabular search>, <reinforcement learning>, <random agent>, and lastly <composite> and <multiplayer search> to glue some of the above algorithms together.
   a. Accompanying book learning is the helper module <action book> in search module that stores different types of book search algorithms.
   b. As the branching factor and game tree is too deep to search until the end of the game. We in addition separated out the <cut off> criteria for search out in order to further customise it.
4. <evaluation> module. Evaluation module defines set of evaluation rules and ways of abstracting the representation of the board and provide convenient API for manually tuning the weights of different parameters.
5. <machine learning> module and <network> module for simple implementation of feedforward NN resembling `keras` style API for possibilities of training and tuning a better machine learning based model.
   a. And inside <network>, a set of simple architectures of neural networks are defined. But sadly we observe that neural network with insufficient depth often converges to a bad local optimum. And those with sufficient depth doesn't converge well.

## APPROACH AND EVALUATION

We explored lots of options for searching, including advanced technique like TD-leaf with Neural net and Monte Carlo Tree search. Regular tree search including MaxN and Minimax. But before precisely defining everything, we need to discuss the nature of the game to gain more insights of different approaches.

We define Chexers as a multiagent, observable, sequential, deterministic, and dynamic problem. Players observe the game through the referee and try to produce a sequence of actions to win the game. Actions dictate result states. Time constraints make the game dynamic.

A **utility-based agent** is a good choice because it stores a representation of the current state and rules of the game. A goal-based agent is not suitable for the Chexers problem because it is difficult to choose a goal. The goal to take all opponent's pieces then exit might not work with defensive opponents. The goal to move to exit as soon as possible might not work with aggressive opponents. On the other hand, a utility-based agent does not have explicit goals. The agent uses rules of the game to generate resulting states of all possible actions and then chose the action that maximizes utility.

There are two main search algorithms for Chexers problem: MaxN and MiniMax.

We used Paranoid Minimax agent with a tuned evaluation function for this task mixed with some amount of starting game and end game tweaks.

To rationalise our choice:

1. Based on empirical observation, Paranoid Minimax on average performs better than MaxN agents.
2. Better pruning can be achieved by using Minimax.

(Detailed comparison with other approaches will be presented later)

## EVALUATION FUNCTION

A full game search is too expensive. Without a full game search, an agent needs prior knowledge to make informed decisions. Since the agent is utility-based, we wish to construct a utility function (U) that takes in a state (S) and returns the utility (u).

$$U: S \rightarrow u$$

The utility of a winning state is set to be the maximum. The utility of an intermediary state is the utility of the best final state. However, it is too expensive to fully explore the game and find the best final state of all states, so we need to find a linear function ($\widehat{U}$) that uses a set of features (X) and their corresponding weights (W) to approximate the utility function.

$$\widehat{U}(S) = \sum_{i=1} w_i \cdot x_i(S)$$

The evaluation function is evaluated for each terminal state. The more features the evaluation function uses the higher the accuracy and the higher the computation cost. A good approximation function should assign higher values to states that might lead to the winning state.

Our evaluation function is composed of 6 different components.

1. Number of necessary pieces.
    a. This is an obvious choice, as for player who don't get enough pieces to win the game will for certain lose. And we put rather high weight on this.
2. Sum distance of nearest 4 pieces to the exit point.
    a. Agent not only need to take down opponent pieces, but also need to move toward the object to win the game. Thus such distance is considered. To help the agent move close to the objective
3. Sum distance between excess piece and exit of the winning opponent's exit place.
    a. This is taken into consideration because when excess pieces are held, it might be a good idea to also stop some near winning opponent from winning by blocking their path.
4. Number of exited pieces
    a. This is an important metric for guiding agent to achieve the objective. Otherwise agent might hesitate near exit and not exiting. This should have higher priority than excess pieces but lower priority than necessary piece to make sure objective will be achieved.
5. Worth of opponent pieces
    a. This initially seems unnecessary for this is already taken into account by our pieces. But on a deeper consideration that if one player dominates the other player, it should be considered much worse than two opponents have even power. Thus, this is taken into consideration
    b. This also makes the paranoid not so paranoid; the two opponents might also start conflicting each other in searching.
6. Winning opponent's total distance to their goal.
    a. This is used to encourage the agent to be less pessimistic and consider case when two opponents are competing each other.

Different feature weights will be used during different phase of the game. In the beginning, a rather aggressive weight scheme is used. Towards the end of the game, the agent is encouraged to exit as soon as possible.

## LEARNING THE BEST WEIGHTING

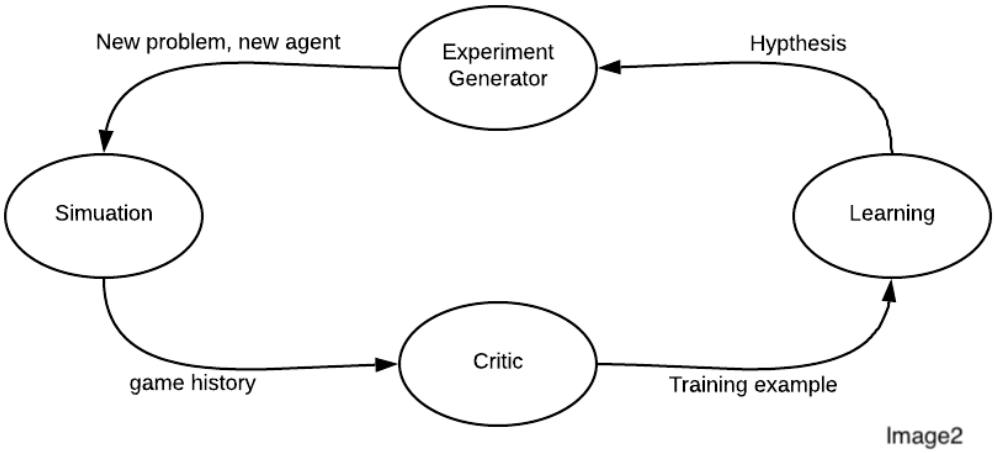


Image2

The inspiration of the learning system shown in image 2 comes from the exam preparation technique where the student would compile the most difficult questions into a book. Students would work on many exam papers and the questions that they got wrong goes into the book. The student would only study the questions that they got wrong. The idea is to never make the same mistake twice.

Firstly, an initial hypothesis about the weights is passed to the Experiment Generator. The Experiment Generator creates a new agent using the new hypothesis and picks a problem that will maximize the learning rate of the system. The Simulation outputs game history which the Critic analyses. The Critic picks out states where the agent failed to identify the correct moves, find the move the player should have taken and compiled it into a training example. The training example is stored as a table of correct moves in the Learning Module. The Learning Module then tests different hypothesis until the hypothesis produces moves matching the table of correct moves. The tested hypothesis is then piped to the Experiment Generator and the cycle continues until no improvements can be made.

A lot of the steps are done manually. We looked at the game log and pick what situations where we think the agents failed to identify the correct moves and add them as test cases. We then try out different hypothesis and run it against the test cases. We then try different agents such as greedy, random, or online battleground to find rooms for improvement.

The player's win rate increases as it becomes more aggressive. A good strategy exploits other player's mistakes by taking their pieces before the window of opportunity closes. The highest weight (1000) is given to number of completed pieces because it defines the winning condition. Higher weights are placed on the value of player's (100) and opponent's pieces (-60). Player also considers the distance to the exit. There is a higher emphasis on the leading opponent's distance to winning (-10) compared to player's distance to winning (5). If opponent is closer to the goal, the player should be rewarded more for blocking opponent's passage than moving closer to goal. The final weights are shown below.

| Features | Weights |
|---|---|
| Total number of pieces (pieces on board and exited pieces) | 100 |
| Leading player's distance to winning | -10 |
| Distance of excess piece to opponent exit | 6 |
| Net worth of other player's pieces | -60 |
| Negative sum distance to goal | 5 |
| Number of completed piece | 1000 |

## COMPARING SEARCH ALGORITHMS

MaxN algorithm makes the assumption that opponents choose actions that maximizes their utility. This assumption has two flaws.

1. How do we know opponent's utility function?
2. What if opponent does not play optimally?

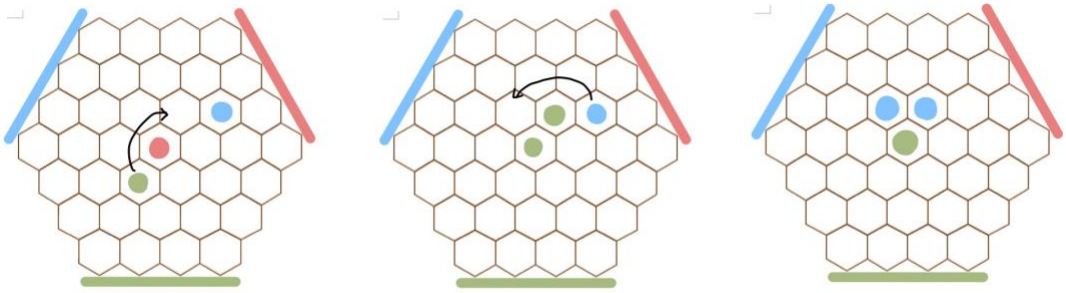

Image1.a Image1.b Image1.c

Image1.a shows a situation where Red thinks its position is safe. Assuming we are using MaxN 1 turn look ahead. Red will rationalize that if Green takes Red's piece, Green will expose its piece to Blue (image1.b) and then Blue will take Green's piece (image1.c). If green takes red's piece, the resulting state will be in favour of Blue. Because after the exchange, Green is 1 distance further from the exit, and Blue has 1 more piece. Instead, Green should move to the exit and win the game. However, a greedy Green player will eat Red's piece, and this is often the case.

Intuitively, if a player assumes everyone to play optimally, then he runs in the risk of over-analysing a situation. If a player assumes everyone to play sub-optimally, then he won't win against stronger players.

A safer approach is to assume everyone is playing against you. Their goal is not to win the game but to make you lose the game. To achieve this, we can assign opponents' utility to be the negative of the player's utility, so opponents' maximum utility is the player's minimum utility. An even better solution to this problem is paranoid Minimax.

Paranoid Minimax groups the 2 opponents together into 1 large opponent. Minimax algorithm is a good choice because compared to MaxN, it uses better pruning techniques.

By grouping 2 players into 1 big player, we decrease the depth of search by a factor of 2/3 but drastically increases the branching factor by the power of 2. Now agent needs to consider all the move combinations of the opponent.

| | Number of evaluations best case |
|---|---|
| Alpha Beta pruning (Minimax) | $2b^{m/2} - 1$ (Winston, 1984) |
| Deep pruning (MaxN) | $b^m + n - 1$ (Luckhart and Irani, 1986) |

Assuming: an n-person game tree with constant branching factor b, m levels of look ahead, and n players

Minimax pruning algorithm's high performance is based on the knowledge that your opponent will not allow you to select states that maximize your utility. MiniMax also has the benefits of needing 1 evaluation function, whereas MaxN requires n evaluations for n players. In practice, the amount of pruned node is reduced by a constant factor.

## MONTE CARLO TREE SEARCH

UCT is a type of monte carlo tree search algorithm, which aims to balance the search by a value term and an confidence term. The value term gives the agent an evaluation on how good a resulting state will be given all past searches. While the confidence term controls how much the agent should explore not frequently visited states.

- This algorithms has four major components: selection, expansion, simulation and backpropagation.
- During selection one will expand the known tree nodes based on formula: $\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$ where $w_i$ stands for the number of wins for the node considered after the i-th move, $n_i$ stands for the number of simulations for the node considered after the i-th move, $N_i$ stands for the total number of simulations after the i-th move ran by the parent node of the one considered, $c$ is the exploration parameter. We explored multiple options for this hyperparameter including the classical option of $\sqrt{2}$.
- Until selection found unexplored nodes, expansion is simply expanding one of the unexplored nodes.
- Up till here everything is all chosen based on `tree policy`, the next phase rollout or simulation repetitively samples the rest of the game episode based on `default policy`, normally random policy to get a game outcome of that node. Following rollingout, is the process of backpropagation, a process where all parent nodes from the expanded nodes onwards are updated with the newest information from the simulated game result.
- Sadly we see very slow convergence of this algorithm and as this is similar to tabular learning, it takes significant amount space to store all the data. And due to the limited timeframe, we don't have option to dig more into this option.

## VALUE BASED REINFORCEMENT LEARNING

Reinforcement learning are always composed of the following components:

1. Learning approach
    a. Value based learning approach including most classical Q-learning, SARSA and TD explores the game by estimating the value of a state of the game with regard or without regard of the presence of a chosen action.
2. Bootstrapping methodology
    a. Bootstrapping is an important concept in all reinforcement learning algorithms. A low bootstrap size can make the convergence of the estimation be much slower, while high size bootstrap algorithms are hard to implement and are trained slowly.
3. Approximation methodology
    a. Approximation methods are important for abstracting large search space with a simple or moderately less complex representation. Most classical take on this will be tabular learning: Using a table to store the approximation of all states. This suffers from overfitting, curse of dimensionality and will significantly increase the search time and search space due to the sparsity of the table.
    b. We have chosen the other, functional based approach, for estimating the state value. This way, estimation one states can also have effects on similar unexplored states, thus significantly reduce the search required. This is boosted by our board rotation idea, further reduce the search space by a factor of 3.

Based on all above discussion, we selected

1. Temporal difference as our learning methodology. As the action space for this problem is not fixed for all states. And applying Q-learning or SARSA will be a lot more tricker.
2. 1 step bootstrapping
3. Simple 2 to 5 layer shallow feedforward neural network with Relu or Sigmoid activation function with MSE loss or Xentropy loss for probability based prediction.

As found by our practical observation, the training process rarely converges or converges to a bad local optimal. Due to the limited timeframe, we are unable to apply more advanced regularisation techniques or better optimisation algorithms. So this idea doesn't really work well.

## ADVANCED TECHNIQUES APPLIED TO TRADITIONAL SEARCH

We also applied few advanced techniques into our search.

1. Preferential ordering for better pruning for Minimax algorithm.
    a. We applied preferential ordering based on the tentatively evaluated value of the states before expanding to find their subsequent states. This significantly reduces the search time in lots of conditions, by roughly a factor of 3.
2. Immediate pruning of enemy actions
    a. Enemy actions are immediately pruned if it will be an uninteresting action (moves that do not relate to our piece)
    b. This is roughly a similar idea as monte carlo tree search. The search space can be drastically reduced.

## BIBLIOGRAPHY

Winston, P. (1984). Artificial intelligence. Reading, Mass: Addison-Wesley, p.123.

Luckhart, C., & Irani, K. B. (1986). An Algorithmic Solution of N-Person Games.