Luke Di Giuseppe, Audrey Webb
Dr. Sarah Erfani
COMP30024 – Artificial Intelligence
21 May 2019

**Project Part B – Playing the Original Chexers Game**

In order to build and implement this program, we considered numerous algorithms discussed in class and found through independent research that may be applicable. In addition, we had to understand the strategic and algorithmic aspects of the three-player Chexers game. This three-player game (Red, Green, Blue) have four pieces per player. The goal for each player is to exit all of its pieces from the board using legal actions when it is its turn. Legal actions include adjacent moves, passing, jumping over its own pieces or over another player's piece (if piece is directly adjacent to it). If your player jumps over another player's piece, it essentially 'captures' that other piece and you can replace it with a piece from you player. Due to the nature of the game, as well as comparisons made between the numerous game search algorithms, we concluded that the best algorithm for Chexers is $\boldsymbol{max^N}$**.**

**$\boldsymbol{max^N}$:**
Minimax is the 2-player variant of the $max^N$ algorithm. In a $max^N$tree with N players, the leaves of the tree are N-tuples, where the $i^{th}$ element in the tuple is the $i^{th}$ player's score. The $max^N$value of a node where $player_i$ is to move, is the child of that node for which the $i^{th}$ component is the maximum value. In our case, $N = 3$, and because Chexers is quite small and simple compared to other multi-player games we believed $max^N$would be the best algorithm to implement.

In our project plan, we detailed our decision to design a $max^N$algorithm and a TDLeaf($\lambda$) machine learning algorithm to refine the evaluation function used in our program. We also detailed our thoughts on advancing the evaluation function through considering distance from the goal, capture by another player, piece location on the board, and compiling a "book" of opening and end games. We stayed fairly close to our project plan with the biggest difference being choosing not to implement the TDLeaf($\lambda$) algorithm, or any other machine learning or semi-learning (Monte Carlo) algorithm for that matter. Our decision will be explained later in this report as we walk you through our game-playing program decision-making and design process.

The main components of our program, are as follows:

- **Player class:** initialises a player. Sets up internal representation of the game state, and other information about the game state we would like to maintain for the duration of the game. Calculates move strategy.
- **Board_State class:** board state for three players with multiple piece. Considers all piece locations. Generates possible moves.

The Player and Board State classes, the program's decision-making process, and the advanced evaluation function take the state and evaluate it in order to known what action the other players will make. We want to acknowledge that our work was heavily inspired by the "Artificial Intelligence A Modern Approach (AIMA)" textbook python code, $max^N$pseudocode provided in

Matthew Farrugia-Roberts's lecture slides, "A Comparison of Algorithms for Multi-Player Games" paper, and "N-Person Minimax and Alpha-Beta Pruning" paper.

Now that research and planning had commenced, our next steps involved implementing $max^N$algorithm with a basic evaluation function. From this implementation we had the following issues:

- Program ran too slow, particularly for $depth \geq 3$. By slow we mean it was near exceeding the spec's constraint of 60 seconds for each player per game. We believed this was due to the high branching factor as players had a high number of choices per move.
- The game appeared to value capturing other player's pieces more than exiting its own pieces off of the board.

In order to solve these issues, we considered pruning, performance boosting, machine learning algorithms, and designed an advanced evaluation function.

**Evaluation function:**
We chose to begin with creating an advanced evaluation function based off of our understanding of the Chexers game. The resulting evaluation function is:

$$utility_{vector} = average_{value} + 1000 \times total_{score} + 100 \times num_{pieces}, \text{ where}$$
$$average_{value} = \frac{total_{defense} - total_{distance}}{num_{pieces}}$$

The evaluation function encompasses the defensive value score, quadratic distance calculation, player score at current state, and the number of pieces on the board. The defensive value score defines strong positions on the board, or placing heavier weight on positions of safety:

- corner hexes: + 1.5
- edge hexes: + 1
- center hexes: + 0.5

Defining the players as defensive in this way resulted in players tending to charge forward towards the center, and whomever dominated there would typically end up winning the game.

For our **distance calculation**, we decided to take the square of the distance to the nearest exit, as this prevented pieces being 'left behind' whilst our program tried to exit pieces, because moving a further behind piece is considered a more valuable space gain. As a by-product, this required us to increase the value of capturing pieces (num_pieces) to avoid moving instead of capturing.

Defining the distance in this manner resulted in the same color pieces gravitating towards each other (small distance value between a player's pieces). This was a happy surprise as we had originally contemplated including in our evaluation function the relationship between "enemy" pieces and "friend" pieces, where "enemy" represented pieces that belong to different players (should not be near each other) and "friend" represented pieces that belong to the same player (should be near each other). We had decided not to consider distance between "enemy" pieces as we believed it would be computationally expensive, and before we could consider distance between "friend" pieces the quadratic distance calculation essentially did it for us. This was very helpful in playing the Chexers game as it allowed pieces of the same player to stay fairly close together creating a strong position for that player.

The weights given to these features in our evaluation function are:
$weight_1 = 1\ for\ feature_1 = average_{value}\left(\frac{defense\ value\ score + distance\ calculation}{number\ of\ pieces}\right)$,
$weight_2 = 1{,}000\ for\ feature_2 = total_{score}\ (player\ score),\ and\ weight_3 = 100\ for\ feature_3 = num_{pieces}\ (number\ of\ pieces)$.
The distribution of these weights solved our first issue, and exiting the board was now more valued than capturing other player's pieces.

**Action pruning:**
This advanced evaluation function was very helpful and solved one issue, but the program was still running too slow. This led to us implementing performance boosting, or a separate function with move possibilities. Essentially, we performed $max^N$ only on the best possible moves which we defined through limiting the types of actions being taken. This is how we chose to limit the actions of the player's:

- If EXIT is an available move for the player at that state, then EXIT.
- If there are no possible actions for the player at that state, then the player can move a hex back (backwards), but otherwise must keep moving forwards towards its exit hexes.

This addition to our program appeared to solve the time issue that we had. Our program was now running within the time and space constraints for depths greater than 3 (ran even quicker by the end of the game when less pieces were on the board).

As our issues were now solved, and the game was playing very well under the time and space constraints, we chose not to implement pruning. We believed it would make the program unnecessarily complex with not much more improvement than what we already had. Also, through research we found that $max^N$ game trees are less amenable to pruning than Minimax trees, and for $N > 2$, we can't safely prune these branches.

**Machine Learning:**
In our further research of TDLeaf($\lambda$), we found it to be quite complex, and decided it was not reasonably worth spending time trying to implement it. Although it would have been interesting to implement given more time, we judged that it would likely be hampered by the fact that the bulk of our learning would be facilitated by games against itself, limiting its effectiveness greatly.

**Note on Depth:**
In testing, we found that once several pieces had exited the board, we could increase the depth of our $max^N$ tree, and then again once a few more had exited and so on. This is simply because of the decrease in branching factor. However, on testing, we found that doing so provided no significant advantage to play, and that by the time there had been a few exits, the game was likely well on its way to being decided. Hence, we left this out of our implementation.

In conclusion, our program implements a $max^N$ game tree search algorithm with an advanced evaluation function and performance boosting. Based on the Chexers game and our $max^N$ algorithm's input, we are able to understand its best-case time and space requirements is

$O\left(b^{\frac{N-1}{N}}\right)$, where b is the average branching factor and N is the number of players. Through playing the game multiple times we found the following to be true:

- When playing against itself (Red vs Green vs Blue), sometimes there is a draw, but typically one player wins.
- When playing against a randomly generated player, our program wins every time. In other words, a player always wins against randomly generated players when using our program.

Therefore, we found that our $max^N$ algorithm was able to effectively play Chexers with three players within the time and space constraints.

## References


1. Farrugia, Matt. *n-Player Adversarial Search*. COMP30024 Artificial Intelligence lecture slides, University of Melbourne.

2. Fridenfalk, M. (2014) *N-Person Minimax and Alpha-Beta Pruning.* In: NICOGRAPH International 2014 (pp. 43-52).

3. Russell, Stuart and Norvig, Peter. (2010) *Artificial Intelligence A Modern Approach.* Python Implementation of Algorithms, https://github.com/aimacode/aima-python.

4. Sturtevant, Nathan. *A Comparison of Algorithms for Multi-Player Games.* UCLA Computer Science Department, https://webdocs.cs.ualberta.ca/~nathanst/papers/comparison_algorithms.pdf.