Artificial Intelligence:
Chexers Agent

Akira Wang 913391 and Callum Holmes 899251
(Dated: May 20, 2019)

Our agent is inspired by the MP-Mix algorithm, proposed by Inon Zuckerman & Ariel Felner (in this paper), and drew ideas from Nathan Sturtevant's paper on 'A Comparision of Algorithms for Multiplayer Games' (linked here).

I. STRUCTURE

The game was modelled as an N-player adversarial search problem for a zero-sum game. Through the development cycle, our algorithm evolved from a standard 'greedy' algorithm that sought to make the best possible move at 1-ply, to a more sophisticated 'runner' that used the 3-ply paranoid N-player algorithm and a 'slow' player using a 3-ply MaxN algorithm. Our final strategic approach 'mix' partitioned the game into three separate stages, at which different tactics and techniques were used.

A. Early Game

For the first 12-ply, our algorithm uses a small book of opening moves that takes center presence as quickly as is possible in the initial game stages - this was found experimentally to be strongest opening, compared to purely random actions.

B. Mid Game

After the book is exhausted, the algorithm proceeds to various different strategies, depending on certain observations of the game state. First, we calculate the evaluations for each player, and rank them in decreasing order as the 'leader', 'rival' and 'loser', before the following decisions are made:

- By default, the algorithm adopts the N-player MaxN adversarial algorithm at 3-ply, assuming all players simply aim to maximise their own score. This is the least efficient, and thus many edge cases are captured below to optimise runtime and effectiveness. The depth of 3-ply was calculated to be most optimal with a simple script (see **calculations.py**).
- When we are the leader (by our own evaluation), we adopt the N-player paranoid adversarial algorithm. That is, it assumes both opponents will attempt to minimise our score, and so our player always avoids actions that would allow the opponent to do so. This is implemented by use of alpha-beta minimax (treating both opponents as the minimising player).
- When we are in second place or the 'rival' (by our own evaluation), and are in a stable position: We use the N-player Directed Offensive adversarial algorithm; while we assume our opponents will maximise their own scores, our player deliberately attempts to minimise the leader's score to regain a lead, as long as it is not at our players' expense.
- If we are 'rival' but not in a stable position, or we are the 'loser', we opt for a "desperate" approach, but utilising a Paranoid algorithm with the only goal to regain captured pieces.

C. End Game

The below strategies are most representative of an endgame strategy, however can be achieved earlier or later in game play-outs:

- Whenever one opponent loses all their pieces and consequently 'dies off', our algorithm adopts a 2-player minimax with alpha-beta pruning approach, which is a solid and efficient algorithm to use for 2-player zero-sum games.
- If both opponents have died off, or one opponent has died off but our evaluation determines that we are significantly ahead of the remaining opponent, we model the game as a search problem. The algorithm will attempt to optimally exit the 4 pieces closest to the exit hexes - identical to the problem explored in Part A. Our algorithm processes the board, determines the optimal moves and converts it into a 'book' from which the algorithm reads until the goal is achieved.
- If the time used by algorithm approaches the specified time threshold of 60 seconds, the algorithm switches to a 'runner' approach (1-ply paranoid). We concluded that a generous 45 seconds would be given before switching to this approach.
- If we are close to completion, the algorithm overrides any other behaviour and returns the winning move(s) (to prevent any haphazard evaluations from discouraging an exit).

II. MODULES AND CLASSES

The directory can be broken down as follows:

- **state.py**:

  Contains globals and functions that define the base of the game board, piece manipulation and action computations, as well as the State class that encapsulates a game board and its methods.

- algorithms (subdirectory)

  - **adversarial_algorithms.py**:

    Implements the adversarial algorithms used in search (paranoid, MaxN, Minimax with pruning, Directed offensive) and their two-player optimisations. Also includes the Part A Search call.

  - **book.py**:

    Records the first 4 opening moves for our player in each colour.

  - **heuristics.py**:

    Contains all state evaluations used in the game (See Part 1 Section E).

  - **logic.py**:

    Contains the main function for our agent logic, which determines which strategies to use as discussed prior.

  - **PARTA** (subdirectory):

    Contains the code used in Part A section of the Project with minor tweaking to interface with the new additions to the project.

- deprecated (subdirectory):

  Contains any deprecated files/other documents used throughout development, including bash scripts, other players and unused algorithms. As may be expected, these may not necessarily cohere with the methods and classes in the above documents.

III. HEURISTICS

The below heuristics were used throughout the agent logic. Output of every was a vector returns the evaluations for red, green and blue respectively, and all s are intended to be maximised:

- **Exits**:

  Returns raw number of exits achieved. Absolute benchmark, and best captures a state's utility and should be maximised by all players.

- **Desperation**:

  Calculates number of pieces in possession, less the number of exits left to attain. Players with high evaluations are significantly ahead piece-wise, whilst those with low scores (¡ 0) are considered desperate and must re-capture to stay in the game.

- **Achilles**:

  Calculates the negative of (number of vulnerabilities) - a player with 2 pieces captured by an opponent (immediately) scores a -2 here. We allowed this to both evaluate 'real' threats, or 'unreal' (potential) captures: the former helped to avoid captures, the latter to maintain a reasonable piece structure on the board (e.g. a 'diamond' or 'wave' of pieces with few weaknesses).

- **No. Pieces**:

  Calculates the number of pieces a player has. Only used explicitly in 'runner' and 'greedy' algorithm evaluation functions, used implicitly to derive other heuristics.

- **Displacement**:

  Calculates the raw number of 'rows' by which a player's pieces have moved. If all pieces are at starting positions, this is 0: if all pieces are about to exit, this is 24. Not used explicitly, but rather to derive other s, as it is unreliable. For example, players with 10 pieces but near the start could score higher than a player with 1 piece and is close to exiting - clearly, the second player is better off.

- **Speed Demon**:

  Calculates as displacement / no. pieces: in other words, it is the 'average' displacement of a player's pieces. This is a better metric than displacement, as only players with significant overall piece progression can score highly here.

- **Favourable Hexes**:

  Assigns a point for each player's piece that is in a corner hex (no vulnerabilities) or an enemy hex position (blocking their exits).

- **Block**:

  A two-player analog for Favourable Hexes.

- **Endgame**:

  Calculates a weighted sum of the evaluations of 5 heuristics experimentally observed to be most effective:

  - Exits $\times 2.5$
  - Desperation $\times 1.2$
  - Achilles $\times 0.25$
  - Favourable Hexes $\times 0.1$
  - Speed Demon $\times 0.2$

The weights above were experimentally derived, but are intuitive. Exits and Desperation best capture the extent to which utility has been achieved: any player with poor Exits/Desperation is highly unlikely to win against even a random player. Thus, they are highly weighted.

Speed Demon, Favourable Hexes and Achilles are informative: they should not fully define behaviour, but rather in the absence of the ability to increase explicit utility, should be maximised. Thus, they are weighted less to prevent them from 'overriding' any gain in exits/desperation.

- **Two-player**:

  A two-player analog of the Endgame, with slightly higher weighting of exits and desperation so as to prioritise achieving the goal.

- **Runner**:

  A weighted sum of Speed Demon, No. Pieces and Exits evaluations. This was subjectively deemed to best encapsulate a 'runner' strategy: it is greedy, ignores other player progressions and will capture to get ahead and win.

IV. OPTIMISATIONS/CREATIVE TECHNIQUES

The following is a detailed but non-exhaustive overview of optimisations used:

- **Winning Move**: Code was implemented to force algorithms to exit to win where possible.
- **Hashing**: A minimal-collision custom hash function was used for compressing states.
- **Symmetry Reduction**: Using the above hash, the algorithm will avoid making any moves that would repeat a previous state unless it has no other choice.
- **Re-use of Part A**: The entirety of Part A implementation was recycled for Part B (see Structure for uses).
- **Move/Jump Ordering**: Computation of possible actions for a state always returned exit, then jumps, and finally moves, to maximise potential shallow pruning. At times, the jumps were also sorted so that opponent-capturing jumps were first evaluated, to enable more aggressive play (such as in Directed Offensive).
- **Cubic coordinates**: Cubic coordinates were used to optimise the runtime of the Displacement .
- **Numpy**: Numpy arrays were used as output for cleaner code and faster evaluation
- **Bash Scripts**: As discussed below, bash scripts were used to automate testing.
- **Complexity Estimation**: Wrote a script (**calculations.py**) with some simple functions to calculate a reasonable ply-depth for adversarial algorithms, which was used to justify the depth of search used in adversarial algorithms.
- **Booking**: The first 4 opening moves are available for the algorithm to use to grasp strong presence in the opening.

A. Honorable Mentions

- **Monte-Carlo**: Though not utilised in the final algorithm, monte-carlo searching methods were greatly explored during testing. Due to time constraints, we found it infeasible to use and when optimised to operate under the resource constraints, failed to exhibit strong performance (overly aggressive). It was used to derive strong opening moves (used in **book.py**) but even this was eventually unused as it only worked against optimal players (as found through experimentation on Battlegrounds).
- **Paris Heuristic**: This evaluated the number of captures that a player could perform. This was depreciated as it over-weighted aggression at the expense of achieving the goal.
- **Retrograde Dijkstra Heuristic**: The heuristic used in Part A to measure minumum number of moves required to exit all pieces. This was found to be over-simplified and 'distance' was better estimated by speed-demon which adjusts for piece count.
- **Greedy Heuristic**: A heuristic only used in Greedy players and was not relevant to final submission.
- $\mathbf{TT}_{Players}$: Customised implementations of the core files that encapsulated states into nodes and facilitated interactive debugging of a game (and any state that was explored).

V. AGENT APPROACH AND PERFORMANCE

A. Method

The following steps were taken, with bash scripts to execute each of them:

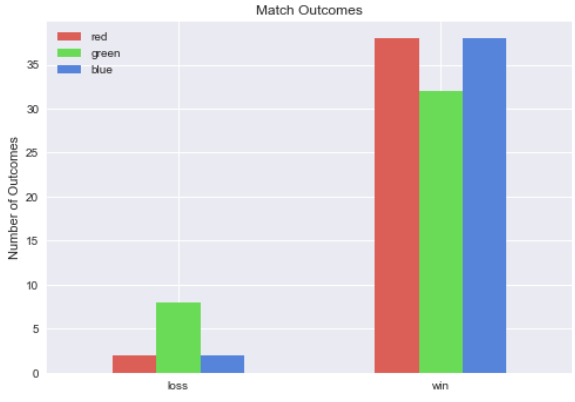
- First, we ran our algorithm against A) Random-Random, B) Greedy-Greedy, C) Runner-Runner and D) MaxN-MaxN, with all colour permutations.
- Second, we repeated the above experiment but threw in a random, potentially sub-optimal action choice every few ply for the deterministic B, C and D players, to test for bias and discourage overfitting of our weighting.
- Third, we ran our own automated script to play 10 rounds of Battlegrounds opponents. This was supervised in execution to scrutinise any strategic mishaps on our part.

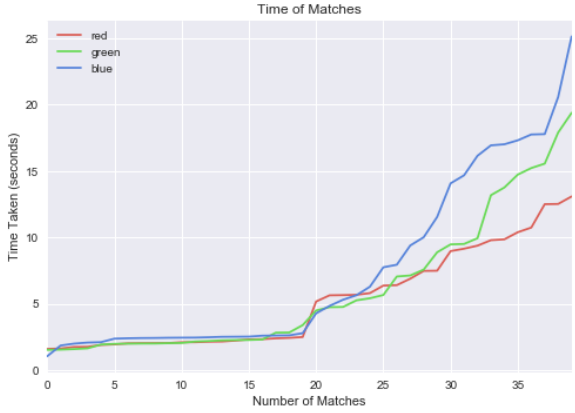
B. Results

Looking at outcomes against the potentially sub-optimal opponents, we are clearly dominant with a win rate of 90.0% out of the 120 matches. Our agent also never exceeded 25 seconds in the simulations - this is likely due to our choice of 3-ply in the adversarial algorithms.

Our losses out of 120 games are:

- **MaxN**: 5 losses
- **Greedy**: 5 losses
- **Paranoid**: 2 losses





C. Battlegrounds Performance

It should be noted that a significant numbers of games were terminated due to server issues or suspected bailouts on the opponents' part. In the end, our results showed us a win rate of about 64.2% (including the suspected terminations, rather than game failure when we were the last remaining player).

VI. FUTURE CONSIDERATIONS

The following are areas for improvement/extension:

- Greater use of Monte-carlo and perhaps combination with TDLeaf(λ)
- Further research into Chexers-appropriate heuristics
- Greater book usage to use full/close to 100 MB of space (we had negligible memory usage).
- Use of statistical approaches to tree search (such as Best Note Search or Reinforcement Learning).

VII. REFERENCES AND ACKNOWLEDGEMENTS

The following sources and individuals are acknowledged:

- Inon Zuckerman & Ariel Felner's paper on 'The MP-Mix Algorithm Dynamic Search Strategy Selection in Multiplayer Adversarial Search' (linked here)
- Nathan Sturtevant's paper on 'A Comparision of Algorithms for Multiplayer Games' (linked here).
- Matt Farrugia's notes on N-player algorithms
- Redblobgames notes on hexagonal grids (linked here)