The University of Melbourne
School of Computing and Information Systems
COMP30024 Artificial Intelligence

Project Part B: Report

Jin Han (843345), [1]

1 Code structure

The game-playing programs consists of the following 3 components:

1.1 Players

Players are the main component of the game-playing program, all of which implicitly implement the "`Player`" interface with the following 3 methods:

- `def __init__(self, color: Color)`
- `def action(self) -> Action`
- `def update(self, color: Color, action: Action)`

Each of the player classes are responsible of maintaining the states and other data that is needed by their respective algorithms, and use the algorithm to make decisions.

1.2 Board

`Board` class is an *immutable and hashable* representation of the game state. A `Board` is defined by (a) all of the Chexer pieces on the board, distinguished only by their colors, and (b) the number of pieces that has exited the board per player.

Besides recording information, `Board` has also provided several useful utility methods including (a) an iterator for possible actions by any player, (b) generate another board using an Action tuple, (c) get the winner of current board if available.

1.3 Type definitions, constants, and other utilities

Utilising the fact that Python 3.6 is used in this assignment, and considering the complexity of this project, type annotations [5] are actively used in all methods. Several types that has special meanings, like `Coordinate`, `Action`, and `Color`, are given aliases for a clearer code structure and preventing unintended type error while writing.

[1] Drawn to scale.

Also, other useful constants (number of pieces to exit to win, etc.) and utility functions (axial manhattan distance, etc.), which are being used in multiple players, are also lifted out to common sub-namespaces.

2 Strategy

2.1 Baseline Players

Two baseline players are created to measure the performance of other players. The *Random Action Player* picks an action randomly from the list of all possible actions in every round. The *Simple Greedy Player* picks the the action which brings any of its pieces closest to the exit hexes.

2.2 Max^n Player

Another player is implemented using the max^n algorithm [2].

Max^n is an adoption of minimax algorithm into multi-player games, where instead of alternating minimum and maximum nodes in a search tree, the node with maximum utility value of each player is chosen at their layers. In a max^n search tree, each node has n utility values for each player.

In our max^n player, we have chosen the following features into consideration:

`dist`
: Number of steps for the "nearest candidates"[2] to exit the board, assuming there are no other pieces on the board. A shorter `dist` value is better.

`n_pieces_exited`
: $\in [0, 4]$. Number of pieces the player has exited from the board. States with more pieces exited are preferred.

`n_pieces_to_exit`
: $\in [0, 4]$. Number of pieces needed to exit to win the game. $4 -$ `n_pieces_exited`

`n_pieces_missing`
: $\in [0, 4]$. Number of pieces the player is lacking of to win the game.
$\max(0,$ `n_pieces_to_exit` $- |\text{Pieces}|)$

`jump`
: Number of `JUMP` actions possible in this state. `JUMP` actions are particularly helpful in both moving faster to the exit hexes and converting pieces of enemies.

`conv`
: Number of pieces of other players can can convert our "nearest candidates". This attribute is used to minimise the chance where our pieces are converted by other opponents, especially when we are in short of pieces to win the game.

`coherence`
: Number of "nearest candidates" that are next to each other. Through observations, we found out that moving multiple pieces together and keep them next to each other is an efficient way to move forward with jumping and lower the chance of being converted by other players.

3 Machine learning techniques

3.1 Monte Carlo Tree Search

At the beginning, as an alternative strategy, we explored Monte Carlo Search Tree Search (MCTS) for Chexers Player. MCTS algorithm [3] is a search algorithm that randomly sample the search space through Monte Carlo simulations, and going through the "selection-expansion-playout-backpropagation" cycle to decide a move. A significant advantage of MCTS is that it does not require an heuristic function to work, which has allowed us to quickly produce a usable agent for benchmarking purpose.

3.2 TDLeaf(λ)

TDLeaf(λ) algorithm is used to train the weights for our heuristic function in the max^{n} player. TDLeaf(λ) [1] is an variant of TD(λ) – a reinforcement learning algorithm usually used with neural networks. TDLeaf(λ), however, are adapted to train weights of heuristic functions in a *minimax* search algorithm. The agent is trained with hyperparameters $\lambda = 0.7, \eta = 1.0$.

$$r(s_i^l, w) = \tanh(\text{eval}(s_i^l), w) \tag{1}$$

$$d_i = r(s_{i+1}^l, w) - r(s_i^l, w) \tag{2}$$

$$w_j \leftarrow w_j + \eta \sum_{i=1}^{N-1} \left(\frac{\partial r(s_i^l, w)}{\partial w_j} \left(\sum_{m=1}^{N-1} \lambda^{m-i} d_m \right) \right) \tag{3}$$

Due to the nature of TDLeaf(λ), it is common to see that a weight set falls in a not-so-effective local optimum. We have also created a script to automate the process of training through self-playing, logging down the weight changes, detect convergence and re-seeding the initial weights. This has helped us to reach a better training result.

4 Effectiveness

On the max^{n} player, we added both immediate pruning and shallow pruning [4] for better performance. Both of the pruning techniques are commonly used in max^{n} searhes. Immediate pruning is to stop searching whenever the heuristic value of a node is found with the maximum value, in this case–when the player wins. Shallow pruning is to stop expanding a node when it knows that this node will not be chosen by examining the current best heuristic value of its parent and the designed total maximum value of all players.

Also, we implemented lazy evaluation so as to prevent unnecessary computation when not needed. This is achieved by only compute the value when first accessed by max^{n} algorithm

[2]"Nearest candidates" are the n pieces of the current player that are nearest to his exit hexes. n is determined by the number of pieces needed to exit to win the game at the point of time.

through customized getters and setters in `Maxⁿ_Player.Score` class, which is also responsible for the computation per se.

Through a one-vs-two tournament-like experiment among all agents we prepared, we have come out with the following ranking of effectiveness, from the most effective one to the least:

$$\texttt{Tkinter}^{3} > \text{Max}^{\text{n}}(3)^{4} > \text{Max}^{\text{n}}(2) > \text{Greedy} > \text{MCTS} > \text{Random}$$

5 Creative techniques

5.1 Algorithms

In this project, we utilized various algorithms, training, and optimisation techniques for a better performance, including MCTS (section 3.1), TDLeaf(λ) (section 3.2), immediate and shallow pruning (section 4).

5.2 `Tkinter` Player

In order to play the game and testing our agents within the team, we implemented an GUI player using Matt Farrugia's `texgen.py`[5] based on `Tkinter` GUI library.

The `Tkinter` Player consist of 2 parts, a "server" that communicate with the Referee, and a "client" that communicate with the user for game status and moves. This design decision is made due to the fact that `Tkinter` window cannot run in a separate thread or non-blocking, which is not feasible when working with the unmodified version of the Referee program. "Servers" and "clients" communicate with the Python's built-in XMLRPC protocol.

5.3 Miscellaneous

In this project, we have implemented various data structures specially for the game and some algorithms, including `Board` (section 1.2), `Maxⁿ_Player.Score` (section 4), etc. Lazy evaluation (section 4) is also introduced for a more efficient computation. Also, other techniques such Type Hinting are also utilized in the code (section 1.3). An automated script is also written to speed up the training process for TDLeaf(λ) (section 3.2). Last but not least, we actively used Unicode mathematical notations in the source code, which makes it look more similar to what is shown in the report, reference papers, etc.

[3] Played by a human player.

[4] $\text{Max}^{\text{n}}(n)$ indicates a search depth of n layers.

[5] `https://app.lms.unimelb.edu.au/webapps/discussionboard/do/message?action=list_messages&course_id=_382870_1&conf_id=_814031_1&forum_id=_455420_1&message_id=_1870319_1&nav=discussion_board_entry`

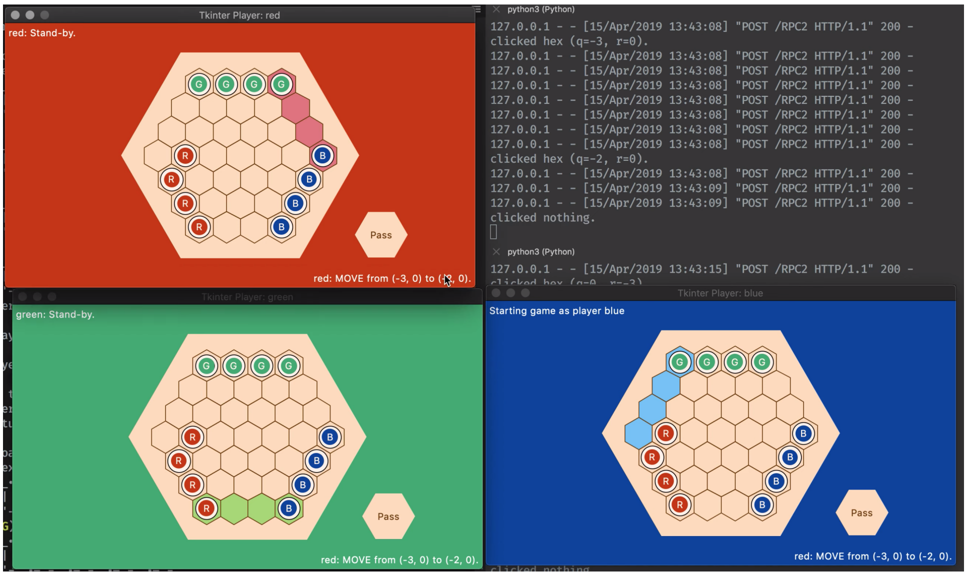

Figure 1: Screenshot of `Tkinter` Players at work [YouTube]

References

[1] J. Baxter, A. Tridgell, and L. Weaver. Tdleaf(lambda): Combining temporal difference learning with game-tree search. *CoRR*, cs.LG/9901001, 1999.

[2] C. A. Luckhardt and K. B. Irani. An algorithmic solution of n-person games. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, pages 158–162. AAAI Press, 1986.

[3] J. A. M. Nijssen. *Monte-Carlo Tree Search for Multi-Player Games*. The Dutch Research School for Information and Knowledge Systems, dec 2013.

[4] N. R. Sturtevant and R. E. Korf. On pruning techniques for multi-player games. In *AAAI/IAAI*, 2000.

[5] G. van Rossum, J. Lehtosalo, and L. Langa. Pep 484 – type hints. *Python Enhancement proposals*, Sep 2014.

Appendices

A Git commit statistics

	Lines added	**Lines deleted**	**Commits**
Jin Han	+4187	−447	30
Mingda Dong	+44	−13	1

Table 1: Git commit statistics