COMP30024 Project B - Report

Lucas Fern (1080613),
Hugo Lyons Keenan (1081696)

9 May 2021

Contents

1 Approaches

Iterations of the player agent used a variety of different search algorithms to determine their future moves. This section will outline the algorithms which were either successfully or unsuccessfully applied to the RoPaSci 360 agent.

1.1 Final Approach: Negamax Search with α, β and Heuristic Pruning (?)

This is my prediction of what our final approach will be. Waiting to write this section until we actually know.

1.2 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) explores deep into the game's search tree by making repeated random moves in order to determine whether an immediate action will continue to be rewarding as the game progresses. It operates in multiple stages, first expanding its exploration of the game tree (the expansion or exploration phase), then performing a 'rollout' from one of the leaves of the tree it has expanded [3]. The rollout consists of making random moves for each player until a terminal state is reached, then the back propagation stage is started, where the node at which the rollout begun will be considered more or less desirable depending on whether the terminal state reached was a victory or loss. This change in desirability is propagated up to the parents of said node. The rollout phase is performed repeatedly while there is time left to make a move, and the amount of time allocated was a parameter that could be set to trade between time and accuracy of predictions.

There are, however, multiple issues with the implementation of MCTS in RoPaSci 360, which is why it was not the algorithm selected in the final implementation. These will be outlined below.

1.2.1 Issues

Extreme Branching Factor In a worst case scenario the branching factor for a single move of RoPaSci 360 (by one player) can be well into the 200s. This happens frequently on a player's 9th throw, as they have the option to throw each of 3 tokens onto any of the 61 hexes, resulting in 180 potential moves before even considering the slides and swings of each of the up to 8 pieces they already have in play. The branching factor is the largest issue with the implementation of any tree searching agent for this game, and is especially detrimental to MCTS since it is impossible to perform a reasonable amount of rollouts from every leaf node in the internal tree in an acceptable amount of time. Some attempts to navigate this problem were made, and the methods implemented to manage this are outlined in section 1.2.2.

Inefficiency of Random Moves The rollout phase of MCTS relies on making random moves repeatedly until the game ends. With a large enough number of rollouts the win rate of the random games is supposed to become a good indication of the desirability of the initial state. This is not what happens in MCTS of RoPaSci 360. Instead, considering the enormous amount of moves available at most turns of the game, random moves are so significantly worse than what a reasonable player would choose that the rollouts frequently exceed the turn limit of 360 moves, and when this doesn't happen, the moves vary so wildly in quality that the winner of a random game is an extremely poor indication of the value of the initial state.

A secondary effect of this is that the rollouts take a large amount of time to complete, since so many run the game to the upper turn limit. This constrains the amount of rollouts that can be performed, and makes it unreasonable to achieve a large enough sample size to accurately determine the desirability of a node. Attempted methods to overcome this issue are also outlined in 1.2.2.

1.2.2 Optimisations

Combatting the Branching Factor Using the heuristic defined as explained in section 2.2 it was possible to only consider a subset of the moves from each board state. Reducing the state space to the 10 moves with the maximum heuristic value *significantly* reduced the size of the search, however even this reduction seemed unable to bring the search and rollout time down to a reasonable period. Perhaps further optimisations to the data structures used may have yielded another improvement, but the result at this point left so much to be desired that this was not pursued.

Using Random Heuristically Favourable Moves Instead of using completely random moves in the rollout phase an attempt was made to select moves from the subset of the top 10 most desirable children, according to the heuristic value. This did perhaps slightly improve the performance of the agent, but as shown in section 2.2 the heuristic calculation is somewhat time consuming when performed on so many boards and so this did not yield the desired improvement.

1.3 Reinforcement Learning with SIMPLE [1]

SIMPLE, an acronym for **S**elf-play **I**n **M**ulti**Pl**ayer **E**nvironments, is a Python library built on OpenAI's Gym [2]. The library specifies an interface for agents similar to the RoPaSci 360 `referee` module, and trains a neural network by using Reinforcement Learning and playing the agents against each other over many iterations. The library requires that the action space (the set of all actions available to a player at any point in the game) and observation space (format of observations provided to the agent before making each decision) are predefined. This guided the design of the data structures used in the board representation of all of the agents that were designed, but ultimately Reinforcement Learning was not pursued for the final agent since it required a much more in depth knowledge of neural network architecture than any members of the group had.

The design of data structures is discussed in more detail in section 2.1.

1.4 Greedy

The Greedy agent is the simplest of the agents which were designed, and either beat or performed competitively with all other agents. The greedy agent evaluates the heuristic function as defined in section 2.2 on all children states of the current board and simply selects the move which yields the maximum heuristic value. This agent is extremely time efficient, as it does not construct a search tree, but for this reason it also does not have the same level of foresight as the adversarial search agents. After optimising the heuristic parameter values this agent makes moves which seem consistently logical, though it is harder to assess their quality when considering the progression of the game far into the future.

2 Implementation

2.1 Data Structures

2.1.1 Player

The interface to the player class is defined by the referee. All of the agents designed for the project had a similar player class which initialised and stored a `Board` object representing the current state of the game, and updated this each turn with the moves returned from the referee. The adversarial search agents used their `action()` call to construct trees, where the greedy agent simply generated heuristic values for the current board's children.

2.1.2 Board

The `Board` class is where the majority of the operations for each agent were stored. This class also stores the positions of all the pieces on the board, and other statistics such as the number of remaining throws of each player at said board state. The interesting aspects of the board class are briefly outlined below.

Piece Position Storage The board class leverages the significant speed advantages of `numpy` data structures over native python ones to store, manipulate, and generate inference from the board as quickly as possible. Since there 61 hexes in the RoPaSci 360 board, and 6 possible pieces which can exist on any hex, the board is stored as a 61×6 `numpy.array` of unsigned 8 bit integers. This is possible since the amount of pieces on each hex is never less than 0 or more than 64 (in theory it could be up to 18).

To achieve this, the game's axial coordinate system is mapped to the integers 0-60 from left to right, top to bottom (English reading order). An example board with one Upper `S` on hex $0 = (4, -4)$ and two lower `p`'s on hex $59 = (-4, 3)$ would therefore be stored as shown to the right.

```
    R  P  S  r  p  s
 0 [[0, 0, 1, 0, 0, 0],
 1  [0, 0, 0, 0, 0, 0],
                ...
59  [0, 0, 0, 0, 2, 0],
60  [0, 0, 0, 0, 0, 0]]
```

Efficient Operations This data structure for the board not only confers the speed advantage from the use of `numpy` but also allows for some extremely compact operations. An extreme example from the code is that battling the tokens on a hex can be performed in one line. The code is not at all intuitive, but extremely efficient for an operation which is called upon during the creation of any new board. This code, and a brief example of what happens when the operation is performed are shown below as a showcase of the merits of this data structure:

```
 R  P  S  r  p  s
[1, 0, 1, 0, 0, 2]      This represents a hex with the tokens {R, S, s, s}.

[R||r, P||p, S||s]      Do an elementwise OR on the same lettered pairs.
       = [1, 0, 1]
          <  <  <       Offset this by one (with the i+1 and i+4 indices).
       = [0, 1, 1]
                        Repeat this array twice (python array * operation).
 [0, 1, 1, 0, 1, 1]
                        Perform the NOT operation to swap 0's and 1's.
 [1, 0, 0, 1, 0, 0]
                        Then multiply this by the original array elementwise.
 [1, 0, 0, 1, 0, 0]
*[1, 0, 1, 0, 0, 2]
------------------
=[1, 0, 0, 0, 0, 0]     As required, the rock has killed all scissors.
```

Many other operations on the board are subject to similar optimisations but will not be shown in the report.

Information Sets for Simultaneous Play Since RoPaSci 360 is a simultaneous play game it is important to be able to generate a game tree with information sets. In game theory an information set between two nodes of a tree means they don't know which path they have traversed from the parent node. This applies to RoPaSci 360 as when the simultaneous game is modelled in an extensive form game tree, moves should not be reflected on the board until both players have submitted an action.

Might include a tikz image here of an extensive form game with information sets. I made a few when I took ECON20002/20005 which I should be able to adapt.

In code this was achieved by choosing our player's team (`upper` or `lower`) as the root node, and only applying moves to the board in sets of two when it was their turn in the tree. This meant that children of these nodes saw an identical board state to their parent and so the simultaneous nature of the game is preserved.

2.2 Heuristic

The heuristic is an essential element of almost all of the strategies implemented by all of the agents. It returns an estimate of the desirability of a given board state for the agents team relative to other states. The heuristic value considers 5 factors and places different weights on each. These are:

- The number of **throws** the agent has remaining. More is better.
- The amount of **dead opponent tokens**, where more is also better.
- An **offensive distance** score based on the average distance between each of the opponents pieces and closest allied piece which can kill it.

  eg. If the opponent has one `p` token and the agent's closest `S` token is 3 hexes away, the offensive distance score will be 3. This is averaged over all of the opponents tokens.
- A **defensive distance** score based on the average distance between each of the agent's pieces and closest enemy piece which can kill it. This is the same as calculating the **offensive distance** for the opposing player.
- A **diversity** score which rewards heterogeneous piece placement by adding a value proportional to the entropy of the agent's piece distribution to the heuristic. This is desirable as it avoids playing too many of the same kind of token which could all be wiped out by a single opponent. It is calculated as:

$$\sum_{i\in\{r,p,s\}} \frac{\#(i)}{\#(r+p+s)} \cdot \log_2\left(\frac{\#(i)}{\#(r+p+s)}\right) = \sum_{i\in\{r,p,s\}} \Pr(i)\cdot\log_2\Pr(i)$$

All of these scores are normalised to be between 0 and 1 by dividing by each of their maximum possible values. They are then multiplied by a weight, and summed to achieve the final heuristic value. The optimal weight values are discussed in the following section.

2.2.1 Heuristic Parameter Optimisation

This hasn't been done, but I feel like we should do this to make the heuristic as good as possible. Find an evidence based method to optimise the parameters.

The current weight values are:

- **throws**: 1
- **dead opponent tokens**: 10
- **offensive distance**: 5
- **defensive distance**: 3
- **diversity**: 1

This strikes a good balance between conservative play - avoiding opponents it is vulnerable to - while aggressively pursuing kills, giving consideration to retaining a few throws for the late game, and maintaining a diverse selection of pieces.

2.3 Algorithmic Optimisation

3 Performance

3.1 Win Rates

3.2 Time Complexity

References

[1] David Foster @davidADSP. *SIMPLE: Selfplay In MultiPlayer Environments*. March 2021. URL: https://github.com/davidADSP/SIMPLE.

[2] Greg Brockman et al. *OpenAI Gym*. arxiv:1606.01540. 2016. URL: http://arxiv.org/abs/1606.01540.

[3] John Levine. *Monte Carlo Tree Search*. YouTube. March 2017. URL: https://www.youtube.com/watch?v=UXW2yZndl7U.