COMP30024 Project B - Report

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

12 May 2021

Contents

1 Approaches **1**
1.1 Monte Carlo Tree Search . . . 1
1.1.1 Issues . . . 1
1.1.2 Optimisations . . . 2
1.2 Reinforcement Learning with SIMPLE [1] . . . 2
1.3 Greedy . . . 2
1.4 NegaMax with Alpha-Beta pruning . . . 2
1.5 Final Approach: Greedy Agent with Heuristic Optimisation . . . 2

2 Implementation **3**
2.1 Data Structures . . . 3
2.1.1 Player . . . 3
2.1.2 Board . . . 3
2.2 Heuristic . . . 4
2.2.1 Heuristic Parameter Optimisation . . . 4
2.2.2 Zero Sum Heuristic . . . 5
2.3 Algorithmic Optimisation . . . 5

3 Performance **5**
3.1 Win Rates . . . 5
3.2 Time Complexity . . . 5

A PowerShell Script for Agent Competition **6**

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 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, loss or draw. 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 is a parameter that can be set to determine the trade off between accuracy of moves and time taken.

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.

1.1.1 Issues

Extreme Branching Factor In a worst case scenario the branching factor for a single move of RoPaSci 360 (by one player) can be grater than 200. 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 183 potential moves before even considering the slides and swings of each of the up to 8 pieces already 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 1.1.2.

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 becomes 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, 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 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 1.1.2.

1.1.2 Optimisations

Combatting the Branching Factor Using the heuristic defined as explained in section 2.2 it is 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.

Rollout 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.

1.2 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 either of the members of the group possessed.

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

1.3 Greedy

The Greedy agent is the simplest of the agents 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 hard to assess their quality when considering the progression of the game far into the future.

1.4 NegaMax with Alpha-Beta pruning

A MiniMax agent seemed to be the most appropriate for the game of RoPaSci 360, and seeing as it a zero sum game, an alternative implementation of the MiniMax algorithm called NegaMax was used. This implementation performs identically but simplifies the coding implementation. Similar to Monte Carlo Tree Search, NegaMax represents the board as a tree where children of a given state are board states reachable by legal moves. It searches through child states to a certain depth, choosing moves from the perspective of each player to maximise the value of a heuristic. Once it reaches a terminal state or the depth limit, it propagates the result back up the tree in such a way that ensures it will play optimally (to the extent that the heuristic is optimal), assuming the opponent will also. Alpha-Beta pruning is an addition to NegaMax that helps prune certain parts of the tree than cannot contain better moves than the ones already found. In this way it reduces the work the algorithm has to do and speeds it up.

Unfortunately, the NegaMax algorithm, even when restricted to searching only a small number of the best children (see section 1.1.2), does not provide results in an appropriate time. The maximum workable search depth which could be achieved with this was 2, which - after testing over a large sample size of games - ended up performing worse than the base heuristic agent. Due to this drawback, it was not selected as the final agent.

1.5 Final Approach: Greedy Agent with Heuristic Optimisation

Since attempts at a more nuanced adversarial search algorithm proved less successful than expected, it was decided that the simple, effective heuristic agent should be used. In order to tune the heuristic weights to be the most successful, a testing tool was used to run a large number of test games between two agents with different heuristics. This is detailed in section 2.2.1 and the optimised heuristic weights are provided there.

Through a machine learning method it may be possible further tune and optimise the weights, a gradient ascent method may be appropriate for this. Alternatively a heuristic function that takes into account more features of the board state may be able to achieve superior performance.

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 detailed below as a showcase of the merits of this data structure:

```
row *= np.array([int(not bool(hex[i+1] or hex[(i+4) % 6])) for i in range(3)] * 2)
```

Listing 1: The python code to perform a battle on a single hex.

```
 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 were implemented using efficient algorithms sourced from *Red Blob Games* [4]. These, and other algorithms performed on the board but not shown in this report are subject to similar optimisations.

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.

In code this was achieved by choosing the agent'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 were identical to their parents (with the move simply stored in memory) 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 4 factors and places different weights on each. These are:

- The number of extra **throws** the agent has compared to their opponent. More is better.
- The number of **dead opponent tokens** minus the number of **dead player tokens** where more is also better.
- A **distance** score based on the how close the players tokens are to opponents token's that they can beat (*offensive distance* - low is good) and that they get beaten by (*defensive distance* high is good.)

  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 **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:

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

The final score is calculated by finding this diversity score for the player and subtracting the diversity score for the opponent.

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

The parameters for the heuristic function were optimised by repeated play between agents with differing heuristic values. A script was created that pits two agents against each other and records the result over a number of iterations. This is included in appendix A and discussed in more detail in section 3.1. Repeated testing yielded the weight values:

- **throws**: 1
- **difference in dead tokens**: 5
- **offensive / defensive distance scores**: 1
- **diversity**: 1

This combination, as determined by automated optimisation, is intuitively balanced towards aggressive play, but with conservative considerations - 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.

2.2.2 Zero Sum Heuristic

The NegaMax algorithm requires a zero sum heuristic to make evaluations of each board state. For this situation the heuristic values *(eg. throws, dead opponent tokens, etc.)* were modified to be equal to the score for the player, minus the score for the opponent. This guaranteed that the heuristics summed to zero and were still an accurate reflection of the agents desirability of a given state.

2.3 Algorithmic Optimisation

Apart from algorithms for manipulating the board state, which were able to be optimised using the linear board representation, the most important optimisation was attempted with the adversarial search methods. Since these searches have too large of a branching factor to consider searching all children of a node, the heuristic value was used to eliminate all but the top n children of any node expansion. This drastically cuts down the amount of nodes searched and is unlikely to prune an optimal move, but since the size of the tree remains exponential with depth d ($O(n^d)$), this optimisation, even in combination with α, β pruning, was not enough to make the adversarial search techniques outperform a greedy algorithm.

3 Performance

3.1 Win Rates

To optimise the parameters of the heuristic, and compare the viability of the various agents, the agents were battled against each other between 100 and 1000 times, depending on the execution time. This was done with a simple PowerShell script which battled the agents using the referee code and recorded the output. This code is included in appendix A.

Because of the time efficiency of the heuristic agents it was possible to test approximately 50 sets of parameter values in a tournament setting. The results of this are too large to include, but some important statistics are:

- The best performing set of parameter values, which are used in the final agent and stated in section 2.2.1 beat the worst performing set in 100/100 games.
- The best performing set did not perform statistically significantly better than any of the top 5 other combinations over a sample size of 200 games each, though these had only slightly modified parameter values.
- The best performing set of parameter values beat the average set in 73% of games, tied 16%, and lost the remaining 11%.

This demonstrates the value of optimising the heuristic parameters.

Comparing the greedy heuristic agents with the adversarial search agents, when constrained by time, the greedy heuristic agent won every single game against the Monte Carlo Tree Search agent. Against the NegaMax agent, it tied in 61% of games, won 28%, and lost the remaining 11%.

3.2 Time Complexity

The heuristic agent takes a relatively constant 0.015s per turn to make a decision. This is the largest advantage of the greedy heuristic agent as the other agents are able to outperform it, but only if provided with significantly more time. It makes sense that the heuristic calculation time is constant over a large sample size since the amount of computation time does not depend on the variable number of children of the node.

The adversarial search based agents are significantly slower and much harder to time. This is because the time taken to perform a search and generate a move for these agents depends on the branching factor. For the NegaMax agent, it was possible to get a move in approximately one second with a search depth of two turns and a branching factor capped at less than 10. Unfortunately, as was discovered in project part A, the testing servers are much slower than the computers used for development, so this was too long for a final agent.

Increasing the depth past two, or the branching factor limit significantly above 10 caused the agent to take in excess of 10 seconds per move. These were not timed accurately since they had no prospect of performing well in competition.

Finally, the Monte Carlo Search Tree was the slowest algorithm of all. When performing a large enough search with enough rollouts to make a quality prediction the Monte Carlo Search agent took in excess of 60 seconds per turn, which is completely unusable for the competition.

Appendices

A PowerShell Script for Agent Competition

```
$n_games = 1000
$filepath = "competition/OUTFILE.txt"

Remove-Item -path $filepath

for ($i=1; $i -le $n_games; $i++)
{
    Write-Host $i
    $OUTPUT=$(python -m referee PLAYER1 PLAYER2 | tail -n 1)
    Add-Content $filepath $OUTPUT
}
```

Listing 2: The PowerShell script to record the results of repeated battles.

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.

[4] Amit Patel. *Hexagonal Grids*. May 2020. URL: https://www.redblobgames.com/grids/hexagons.