1 Introduction

Primarily, the program, written in Haskell, is an AI agent which plays Fanorona, using a variant of NegaMax with alpha-beta pruning to inform its decisions.

In particular, the AI implements several methods to achieve improved efficiency of alpha-beta pruning, as well as encouraging searching more relevant gamestates.

2 Structure Overview

The source code is divided into several modules, including test suites. Those relevant to the main AI are:

- `NegaMax.hs` contains the tree-searching algorithm used by the main AI.
- `SimpleAI.hs` exports definitions for the game tree and small functions.
- `Zobrist.hs` is the AI's transposition table.
- `Keys.hs` stores the randomly generated integers used in `Zobrist`.
- `Eval.hs` defines different evaluation functions for the game.
- `RBTree.hs` implements red-black trees.
- `AI.hs` contains a list of the AIs with different parameters.
- `FanoronaExtra.hs` has extra functions dealing with game states.
- `Fanorona.hs` is a representation of the game, including legal move generation.

While the remaining modules are for testing or contain outdated AIs:

- `AITests.hs` includes unit tests and black-box tests for the AI.
- `ExtraTests.hs`, `ZobristTests.hs`, `EvalTests.hs` contain unit tests for their corresponding modules.
- `RBTests.hs` tests self-balancing of the red-black tree.
- `FanoronaTests.hs` and `Testing.hs` were included in the assignment code.
- `BoardStates.hs` Example gamestates for testing.
- `AlphaBeta.hs` an initial implementation of AB pruning.
- `AdvancedAI.hs` was a precursor to `NegaMax`.

3 The Algorithm

3.1 The Game Tree

To represent the search space, we use a rose tree of `GameStates`. Move generation is implicit in the tree structure, and lazily defined by the `gameTree` function in `SimpleAI`. To restrict search depth, we cut the tree using a `qcut` function.

Repeated turns are represented as different nodes in the tree; dealing with this is left to the search function.

3.2 Evaluation

Values of leaf nodes in the tree are given by a heuristic function examining the number of pieces and how many neighbouring tiles each piece has. This implemented using a weighted list, assigning higher values to tiles with more connections.

The rationale of this is to encode the idea of attacking many positions while also having many options for retreating pieces.

In the late game, it has the effect of encouraging the AI to spread out its pieces rather than forming clumps, which suffer from poor mobility and are easily wiped out.

After the initial evaluation is calculated, we use a sigmoid to restrict it between -1 and 1. This was to ensure evaluation would never exceed the initial alpha-beta window during search.

3.3 Quiescence Search

With a fixed-depth cutoff, it's possible that searching can end in the middle of significant capture sequences or in volatile positions where the evaluation heuristic is weak.

To address this, the cutoff function `qcut` is modified to only cut the tree at 'quiet' states where only `Paika` moves are available. This makes the tradeoff of reducing the tree height in lines of play not involving many captures, while focusing more on lines where a lot of material may be gained or lost.

3.4 NegaMax

To calculate optimal play, we use NegaMax, which is an equivalent reformulation of MiniMax.

Instead of considering minimising and maximising players separately, we define one maximising function which alternates sign. In practice, this simplifies the program, and also reduces the amount of near-identical code.

In `NegaMax.hs`, the search is a recursive function, which performs at a given node, a recursive maximum of its children. The recursion is structured so evaluation is performed in-order so that table information, as well as alpha-beta windows can be passed along.

3.5 Alpha Beta Pruning

We use deep alpha-beta pruning to improve NegaMax's performance while preserving equivalence with regular MiniMax. In `NegaMax.hs`, pruning works like so:

At a node, alpha is defined as the maximum of left-siblings, while beta is the best score the opponent can guarantee at some ancestor.

When alpha exceeds beta at a child, we know the opponent will never let us reach this node. Hence, values of the right-siblings are irrelevant and so we stop calculation.

3.6 Transposition Table

The transposition table is a red-black tree of key-value pairs where keys are (hashed) game states and values are relevant calculations.

Since `GameState`s are not an ordered type, integer keys are used instead. A type of XOR hash called Zobrist hashing is used to convert `GameState`s to `Int`s.

This involves creating a binary-flag representation of a state and assigning a random integer to every flag. For a 9x5 Fanorona board, I generated 137 random 32-bit integers which are supplied in `Keys.hs`.

With keys available, hashing a state entails doing a bitwise XOR of all keys corresponding to active flags (for example, there is a flag for whether a white piece is at `Location 0 0`).

This hashing method has the advantages of being compact, having good randomness properties and also being very fast to compute.

Typically, some form of constant time access structure is used for the table, rather than a tree; however, I wasn't sure how to implement this in Haskell.

3.7 Iterative Deepening

To take advantage of the Transposition Table, the AI uses a move-ordering technique which involves shallow searches succeeded by deeper searches.

In general, a search of depth n is a constant factor more expensive than searching to depth $n-1$. Hence, if the $n-1$ depth search can inform the n depth search, the overall cost of searching depths $1, 2, \cdots n-1, n$ ends up being cheaper than directly searching to depth n.

To make the shallower searches worthwhile, `NegaMax.hs` uses the transposition table to track which moves led to beta-cutoffs, and which nodes ended up being the best scoring.

At each node, children are sorted according to their score in the table and visited in the new order. The sorting occurs in the grandparent node, to avoid the cost of this from blowing up during recursion.

4 Testing

4.1 Unit Tests

The program comes with automated tests, a combination of unit tests and black-box tests.

The unit tests such as those in `EvalTests.hs`, `ZobristTests.hs`, `ExtraTests.hs` and some in `AITests.hs`, are very simple test cases to check that small functions are sensible have sensible outputs given typical inputs.

`RBTests.hs` contains tests of whether `RBTrees` really produces approximately balanced trees. On the other hand, `ZobristTests.hs` contains the tests to check that getting, inserting and overwriting with the red-black tree all work correctly.

4.2 Black Box Tests

The main source of confidence for the correctness of `NegaMax` come from the black-box tests, where the AI is tested in different ways on 4 test trees, which are designed to exhibit different important properties:

1. The first is a basic, uniformly 3 deep tree, where each layer corresponds exclusively to the min or max player.
2. This tree tests the AI's move ordering feature. Although the ordering of the first layer is $(9, 2), (10, 2)$, the shallow search of depth 2 indicates that the two should actually be reversed; in the 3-deep search, $(10, 2)$ is checked first which results in a beta cut-off at $(7, 10)$, a child of $(9, 2)$.
3. I extend the second tree with some capturing moves. If quiescence is working, the score at the root should change from 4 to 5.
4. I add repeated moves to the tree, to check that the AI can tell min player from max player.

Thanks to the transposition table, it is very easy to observe the alpha-beta pruning at a detailed level, simply by checking the scores of states in the table, and whether a cutoff was performed at them.

4.3 Fixed Depth Equivalence

There is also a long set of tests which check that all of the AIs included are equivalent to minimax. By restricting the agents to identical search depths and having them play long games, the program checks that the AIs come up with identical moves.

This kind of test is very sensitive to problems such as good moves being pruned, or other minor bugs in agent code. However, it doesn't say anything about pruning efficiency or performance.

5 Issues, Possible Improvements

It's clear that iterative deepening results in a more efficient search, however playing strength is comparable with or without deepening. This could be due to the $O(log(n))$ access time of the red black tree, or the cost of sorting during search. Other ideas such as aspiration windows could also help to make the most out of iterative deepening.

The effectiveness of the AI's move ordering strategy also depends heavily on the evaluation's ability to predict its own future value. Since the evaluation used was very primitive, I don't expect it to be especially strong in this regard. Automated tuning, or a more sophisticated evaluation could improve the search depth.

Regarding transpositions, note that the game is symmetrical in two ways; vertically and horizontally. If the hashing code could be modified so that these mirrored boards were given the same hash, a lot of computation (especially in the early game) could be saved by reusing computations from the table.