Ziling Ouyang

Wednesday 10:00-12:00

U7312578

Word Count: 1498

Program Design

Functions / Datatypes

`Rose a`: A datatype which represents a rose tree. It is used to implement a game tree for Othello. Game trees are just a type of rose tree.

`othelloTree`: Given a desired depth and initial game state, the function builds a game tree to given depth.

`greedyAI`: A function which, when given a game state, returns the best move. The best move is evaluated by giving `greedyHeuristic` a list of legal moves.

`greedyHeuristic`: Given a list of moves, the function compares the score given to moves by helper function `score` and returns the best one. The comparison is done by `maxTuple`.

`score`: A function which evaluates a certain move by calculating the amount of coins it gains the player. It outputs a `(Int, Move)` tuple.

`maxTuple`: Compares `(Int, Move)` tuples and returns the tuple with highest `Int`.

`minMaxAI`: Given a game state, the function generates a game-tree using `othelloTree` and evaluate the leaves using `convertLeaves`. Then, it traverses up the tree through `depthRepeat` to apply the minmax algorithm. The tree is then evaluated by `getBest` and `getMove` to return the optimal move.

`minMaxAI'`: A function which is the same as `minMaxAI` except lookahead is only 4.

`getMove`: A function, when given a game state and position of the best move, returns the best move by generating a list of legal moves from said game state and selects the move in given position.

`getMove'`: A helper function for `getMove` which searches the list of legal moves for the one which matches the given index.

getBest: A function, when given a game tree of depth 1, indexes the nodes and returns the node index which has the highest heuristic score.

depthRepeat: A function which recursively calls `minMaxHeuristic` until a game tree of depth 1 is returned.

minMaxHeuristic: A function which, given a game tree, compares leaf nodes and gives the highest or lowest value (depending on whether min or max) to its parent. The comparison is done by `comparison`.

comparison: A function which finds the minimum or maximum from a list of values depending on whether the player is minimising or maximising.

convertLeaves: Given a game tree and current player, it recurses through a tree until the leaves and evaluates the leaves through `returnScore`. A new game tree is outputted with the leaf values replaced by the heuristic score.

returnScore: A function which evaluates an Othello board based on the given heuristic. It calls `corner` for evaluation of corner occupation.

corner: A function which returns how many corners have wanted player pieces in them.

Explanation of Some Choices

Heuristic: My heuristic assesses three variables: mobility, corner occupation and coin parity. Mobility refers the amount of legal moves afforded to a player from a move. Corner occupation counts corner coins and coin parity measures player coins. While coin parity seems important coins can be flipped and so coin parity is a weak assessment of who is winning. Instead, good mobility indicates that you have more paths to win the game from. Additionally, corners can never be flipped so they are important positions. Overall, my heuristic is calculated by weighing all 3 variables and summing them.

Minmax AI: I first evaluated the leaves according to the heuristic. Subsequently, I traversed the game tree until I reached the node directly above leaves and compared the leaf values based on whether the player was minimising or maximising. This compared value then became the value

of said node and the leaves were deleted. This process is repeated until only a tree with depth 1 is left. The best move is selected from the depth 1 game tree.

Design Choices

Iterative Programming: I implemented AI in ascending order of difficulty because it allowed me to gain understanding in coding game AIs. By first making the greedy AI with no lookahead and inspecting the predefined AI, my understanding of coding for simple games grew. Through this, my latter minmax AI (with lookahead) was able to build on the understanding I had gained.

Higher Order Functions: While I initially wrote my code using recursion, I tried to replace instances of recursion with higher order functions when appropriate. In theory, this would ensure easier to read code and reduce unnecessary length.

`where`: I would try to replace any "values" which appear multiple times within a function with a `where` clause. This was an attempt to make the functions faster by only evaluating said "values" once.

Assumptions

Board: Both `corner` and `returnScore` assume an 8x8 board. `corner` only calculates whether pieces occupy corners in an 8x8 board while `returnScore` calculates heuristic values as if the board were 8x8. The `getMove` function assigns a number onto every move available as an identifier. However, it has a maximum of 64 identifiers because it is assumed that 64 legal moves is the mobility limit at any point during an Othello game. This assumption is obviously false when the board is not 8x8.

Move Selection: Whenever there are two moves which are given the same weighting, the AI selects the move which is parsed in earlier. This is under the assumption that all moves with the same weighting yield the same outcome. However, this does not account for the possibility that the opponent plays an unforeseen move. In this situation, choosing one move over another could limit counter-play opportunities.

Testing

Unit Tests: Unit tests confirm that the expected output of a function matches its actual output. By periodically running unit tests, I caught some errors. Specifically, `cornerTest` failed. Even with two player occupied corners, `corner` returned 0 not 2. I realised that I forgot to pattern match for situations where the opponent also occupied corners. Instead, `corner` counted cases where only the player had occupied corners and ignored when both the player and opponent had corners occupied. Upon fixing this, the test case passed.

Tournament: I entered the tournament ran by the faculty, competed against other people's AIs and my own weaker AIs. Through this competition, I was able to gather information about how my AI performed. By discussing heuristics with the people who beat me, I improved my AI. To test that my new AI had improved, I competed against more people and repeated the cycle.

Dry Run Testing: Dry run testing allowed me to improve my theoretical understanding of a function before attempting to implement it. For `othelloTree`, I drew out a game tree and imagined traversing the tree. Through this, I understood that I had to recursively apply the `othelloTree` function to each subtree of the original tree. Furthermore, it helped me consider edge cases such as the game tree having no further nodes. I then verified the function correctness by drawing a game tree for Othello's initial state up to depth 2 and checking that it matched the output of the function.

Issues

Minmax Defaulting First Move: Originally, the minmax AI would always output the first move. I realised this by comparing `minMaxAI` output with `firstLegalMove`. Upon reviewing the game-tree generated for `minMaxAI`, I realised that all the leaf values were identical. This meant that my heuristic was too primitive and did not value game states correctly. Upon updating my heuristic and testing, `minMaxAI` no longer returned the first move and now beat `firstLegalMove`.

Minmax Error: I noticed that running `minMaxAI` resulted in an exception being thrown. Therefore, I investigated this exception by using `--debug--lookahead`. This revealed that `minMaxAI` was unable to return anything when the lookahead of was 1. Upon checking, I

realised that I had not pattern matched for a lookahead of 1. By fixing this, there was no longer an exception.

Redoing the Program

Alpha-Beta Pruning: If I could redo the program, I would try to code alpha-beta pruning. I had written code to be able to return the values stored in any node in a game-tree, but I was unable to understand how to implement the actual alpha-beta comparison. One of my thoughts was to have a game-tree which contained alpha and beta values. However, I could not figure out a viable method of traversing through the tree to access nodes in the necessary order. Eventually, I deleted all incomplete code concerning alpha-beta pruning to reduce code clutter.

Improved Heuristic: Although I had improved on my heuristic from its original state, it is still lacking. My heuristic does not take into consideration how far along in a game the AI is called. This may have affected the calculation of mobility and coin parity. As a game progresses, mobility should become less of a factor while parity should become increasingly important. Additionally, I did not assess stability (how likely a piece is to be “taken”).

Testing Edge Cases: My unit tests could have covered a wider range of unique situations. As it stands, the tests I implemented only confirm that the functions work in the most general of situations. Tests for unique properties and situations may have uncovered potential bugs while also providing a more thorough confirmation of code correctness.