Australian National University

ASSIGNMENT COVER SHEET

ANU College of Engineering and Computer Science
Australian National University
Canberra ACT 0200 Australia
www.anu.edu.au

+61 2 6125 5254

Submission and assessment is anonymous where appropriate and possible. Please do not write your name on this coversheet.

This coversheet must be attached to the front of your assessment when submitted in hard copy. If you have elected to submit in hard copy rather than Turnitin, you must provide copies of all references included in the assessment item.

All assessment items submitted in hard copy are due at 9 am unless otherwise specified in the course outline.

Student ID	U6650903		
For group assignments, list each student's ID			
Course Code	COMP 1100		
Course Name	Programming as Problem Solving		
Assignment number	3		
Assignment Topic	Othello Bot		
Lecturer	Dr. Katya Lebedeva		
Tutor	Maddox Kam Kwo Wong (Lab time Monday 10am)		
Tutorial (day and time)	N/A		
Word count	1780	Due Date 29.10.2018	
Date Submitted	28.10.2018	Extension Granted	

I declare that this work:

- ☐ upholds the principles of academic integrity, as defined in the ANU Policy: Code of Practice for Student Academic Integrity;
- ☐ is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the course outline and/or Wattle site;
- ☐ is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- ☐ gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- ☐ in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

Initials

For group assignments, each student must initial. RM

Abstract

Developing an Artificial Intelligence that autonomously plays Ottello board game implementing minimax algorithm, alpha-beta pruning, and heuristic in Haskell programming language.

General approach

AI uses a key data-structure Rose-tree to record data of all possible moves and their consequent scores. To reduce processing time tree is pruned. At the leaves board states are heuristically evaluated, and values are lifted up to choose a corresponding to the best value move. To win the game AI employs heuristic:

- Strategically biased weight matrix for board evaluation.
- Mobility factor: considers the maximum difference between legal moves available to beneficiary player and opponent.
- Win factor. Prioritizes moves towards beneficiary player's win.

Operation and testing

- AIs prototypes were testing against each other by adding features one by one.
- Time limit altered for testing to compare the operation in longer terms
- Moves and scores which AIs consider are traced (with Debug Trace) and checked manually and mathematically.
- Test competing against peer student AIs.
- Playing against experienced human (Here and further it means the best players known of my mates)
- Tested different strategically biased score maps.
- Tested different weights for mobility complement.
- Other heuristic tested, compared performance.
- Performance playing Dark and Light parties were compared
- `Cabal run` warnings

Design choices and stages

1. As recommended Greedy bot was developed. But it was very weak AI.
2. Simple minimax algorithm featuring:
    - explicit game tree
    - depth pruning
    - real score board evaluation

    Benefits

    - easy to understand operation and develop the code
    - foresees at least 5 steps ahead within 1 second
    - defeats greedy bot
    - predominantly defeats human (me and my mates)

    Shortages

- loses to AI with weak heuristic (my mates' AI)
- extensive computation for small depth
- performance depend on the party (Dark/Light)

3. Minimax with Alpha-beta pruning featuring:
   - implicit tree
   - depth pruning
   - alpha-beta pruning

   Benefits

   - foreseen depth improved to at least 7 steps ahead within 1 second
   - defeats its predecessor
   - defeats human player

   Shortages

   - loses to AI with heuristic (my mates')
   - performance depend on the party played (Dark/Light)
4. Heuristic: strategically biased board evaluation. I did some research on papers about evolution of Neural Network bots playing Othello. The weight matrices evolved were tested and the best performing one was adopted. The matrix used is a result of 10,000,000 games, but it was asymmetric, so I guessed that symmetry may help it. Testing showed symmetrized by octants (like a Bresenham circle) brings improvement. However, Board state and weight matrix are represented as a 2-dimensional lists which makes computation expensive. It reduced the depth of looking ahead to at least 4 steps, but I decided to make this trade off because it outperforms bot looking 7 steps ahead with no heuristic.

   Benefits

   - defeats predecessors with no heuristic
   - defeats experienced human players

   Shortages

   - expensive computations
   - foreseeing depth reduced to at least 4 steps
   - loses to some AI with advanced heuristic (Best playing bots of my mates)
   - performance depend on the party played (Dark/Light)
5. Mobility factor heuristic. It prioritizes the moves which lead to more available moves for beneficiary player and less for opponent. This factor is summed with strategic board evaluation result, which caused the need of scaling it to achieve best performance. Best scalar was found practically by testing. The computation is cheap but makes improvement. However, there might be a better way to compute a mobility factor: The number of legal moves for beneficiary player and an opponent are calculated on the same board, but the moves available for opponent will be different when make a move, so they have to be compared on next levels.

   Benefits

- improved performance: wins predecessors
- doesn't affect the foreseeing depth because its cheap
- easy to combine with existing heuristic

Shortages

- improvable way of computation
- if opponent doesn't also use this heuristic, but AI assumes it does, it might not be the best strategy
- still performance depend on the party played (Dark/Light)

6. Next heuristic prioritizes the move toward the end of the game with winning beneficiary player, and deprioritizes the move to the opponent win. This helps to find a faster way to win and consequently reduce the number of unnecessary computations. This only becomes effective towards the end of the game when the finish is within less than 4 steps. The cost of weight matrix calculation makes this waste negligible. Computation is very cheap eap and still makes an improvement, so I decided to keep it.

   Benefits

   - low price performance improvement: does not affect the depth
   - closer to the end saves unnecessary computations

   Shortages

   - being checked all the time but mostly relevant towards the end
   - still performance depend on the party played (Dark/Light)

Further improvement and critique

Static heuristic

Improve mobility factor calculation.

4 steps ahead is quite a mediocre forecast, I believe it can be improved: optimize the most expensive heuristic, namely, biased weighted board evaluation. Change the way of processing lists or data structure.

Theoretically, the stability of the pieces is important, however my attempts to develop a stability factor heuristic lead to even more expensive computations and reduced the foreseeing depth to at least 3 steps. This worsen performance. Computing this factor more efficiently might improve performance.

Arrange win factor to check only towards the end.

Heuristic shows an improvement but attempts mathematically prove it might reveal weaknesses and let to improve it.

Apart from static heuristic a dynamic is needed.

The AI calls the move calculating function many times with increasing depth which slows down the game and causes wasteful calculations. This can be improved by developing a function of time to count a depth that we can look into. Or the external timer function which can signal when timeout is about to expire.

Recognize opponent.

It's rarely that we know the opponent's strategy; however, we can recognize it by assessing their moves. For instance, if the opponent's move is recognized as neglecting mobility factor, then we can cancel it from our minimizer board evaluation. It will save some memory for depth and help to better predict the foe. This would need a higher order AI with a set of strategy patterns, acting as a general function which would chose and assign the best fit soldier AI , and swap it if change in opponent's behavior is recognized.

Learning AI.

It's impossible to predetermine every possible opponent strategy, so it is better to develop a responsive weapons on the go. Employing Neural Network would improve the AI, however its learning time is long. I wish I would learn how to use it in my further courses and use it next time.

Technical issues

AI still keeps dependence on the party and performs better with Dark when plays against itself. This might be caused because the dark always moves first so starts always from the same board but white moves second and every time starts from different board. Dark moves first and have a priority to start attacking, then Light is forced to keep up with protecting. This area needs more study to find the way of improvement.

Implementation

Game tree is built with use of data and functions:

- `data type Game1` is an upgraded record data type. Added move and record allows extracting fields by name.
- `type Side` repeats Player but distinguishes the purpose to indicate the beneficiary party.
- `make_childern` creates list of alternative future game-states caused by available legal moves. Used as generator function to build game-tree.
- `gameTree` is a general way to built a multy-way-tree
- `weights` is a weight matrix evolved by Neural Network machine.
- `weightedScore` sums up weights for the beneficiary pieces and subtracts weights of the opponent's (details in code comments)
- `mobilityEvaluation` checks the mobility factor discussed above.
- `board_value` combines board value and mobility together
- `makeBestMove` and `makeGoodMove` are AI function which call calculator functions `alfaPlus` and `alfa` numerous times with increasing depth until the calculation time exceeds timeout. Then it takes the last result.
- `alfa` is a mother-function, it adjusts input type and extracts the required type value from lower level function. It does not apply any heuristic except the level cut off dictated by time limit.
- `Types Alpha, Beta, Depth, Move, Score` are to distinguish Int types, to achieve maintainable code.

- `alfaPlus` is another mother-function, which uses the same calculating core as alfa, but applies heuristic.
- `find_move_ab` is next level down function which serves for `alfa` and `alfaPlus`. It finishes the calculation on the surface level and deals only with the list of the resulting scores, finds the maximum score there and extracts best move by index. (more details see code comments) It also sends calculations down to `alphabeta`, setting up the initial alpha and beta values to practical infinities. It starts the depth countdown from the root's children level, which may lead to **confusion** because foreseen depth is depth parameter +1.
- `alphabeta` searches down the tree until zero depth reached where board is evaluated. In case the terminal node is not reached it passes the operation to `minimise_ab` or `maximise_ab` depending on whose turn: beneficiary or opponent respectively. It does additional heuristic by checking if the game is over and beneficiary player won/lost, and gives extra points. In this case it significantly prioritizes this option by 1000 points (function `who_won`) The tree is built implicitly as it needs it, that's why we see here `make_children` function.
- `Maximise_ab` returns a current board value if detects that `alpha` exceeds beta which indicates that other branches are not worth looking at and can be pruned, and checks the other children otherwise, setting a new a new alpha. New alpha is taken if the board value fetched from down below is greater than the current one. The fetching is done by `new_score` calling `alphabeta` for lower levels. The position is evaluated with `board_value` function described above.
- `Minimise_ab` does the similar job as `maximise_ab`, but it sets new beta if the board value brought from below is smaller than current beta. It is decreasing beta instead increasing alpha.
- `who_won` function issues huge additional score to simulate the action towards winning.

Assumption

- I had to make an assumption that I dint have to present all unsuccessful prototypes and only present the final version of the AI, nevertheless I have presented a second-best version as well.

Reflection

- I have learned about minimax and alpha beta pruning algorithms
- I learned how to test the code rigorously
- If I do such project again I would learn neural network before.
- I have experienced limitations and recognized the importance of intelligent approach.
- I got acquainted with important data structures efficient for problem solving.
- I had to practice a generalized problem vision and think in broader fashion.
- I recognized this task is a foundation for a huge field of practical implementation.
- This task motivates me to learn further about the Artificial Intelligence.

Bibliography

Dr. Katia Lebedeva, Lecture slides for COMP1100 Programming as Problem Solving, College of Engineering and Computer Science (CECS), ANU, 20118.

[Accessed 10 October 2018]

Thompson S., (2011) Haskell the craft of functional programming. Third edition. Pearson Education Limited.
[Accessed 11 October 2018]

Lipovaca M., (2011) Learn You a Haskell for Great Good. A Beginner's Guide. San Francisco. Publisher: William Pollock.
[Accessed 12 October 2018]

Szubert M., Jaskowski W., and Krawiec K.
Institute of Computing Science, Poznan University of Technology, Poland, June 7, 2013.
Learning Board Evaluation Function for Othello by Hybridizing Coevolution with Temporal Difference Learning.
Available at:
https://www.researchgate.net/publication/233530017_Learning_Board_Evaluation_Function_for_Othello_by_Hybridizing_Coevolution_with_Temporal_Difference_Learning
[Accessed 13 October 2018]

Hingston P., Masek M.
Edith Cowan University, ECU Publications Pre. 2011
Experiments with Monte Carlo Othello
Available at:
https://ro.ecu.edu.au/cgi/viewcontent.cgi?referer=https://www.google.ca/&httpsredir=1&article=5969&context=ecuworks
[Accessed 14 October 2018]

UncleK is playing data
Black and White Chess AI: Situation Assessment + AlphaBeta Pruning Pre-Search. Edited on 2018-05-09.
Original In Chinese language.
Available at:
https://zhuanlan.zhihu.com/p/35121997?fbclid=IwAR0pAdfPhqSDid5-gWaTJdR6M_gbaBsmBqUhAjezFOMQBc5Ue0kxmBUyotA
[Accessed 15 October 2018]

Kamko A., for UC Berkeley CS61B (Github project)
Alpha-Beta Pruning Practice
Available at:
http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/
[Accessed 16 October 2018]

Alpha-Beta pruning Wikipedia
Available at:
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
[Accessed 17 October 2018]