Artificial intelligence assessment 2 report

Henry Routson

Note :
Teamwork issues were encountered during this assessment, so the all code and report for this project were completed by one person rather than two. Any use of we or other plural terms are simply for report writing style purposes.

Background
This report details our strategy for creating an agent to play the Tetris like game (Board Tetris) in outlined in assessment specification.

Approach

Our approach for how to create an effective game playing agent changed over time. Our first approach was to use Monte Carlo Tree search (MCTS) to save time creating a heuristic, as it was outlined as the most effective algorithm in the report and in further research. Although we were warned that this was "challenging" to apply successfully, we attempted to mitigate this with a novel approach.

We defined the simplest possible game we could, where the move choice was always an objective choice, allowing us to rapidly determine the effectiveness of our MCTS implementation in a way that would be impossible naively implementing it into the Board Tetris task. The game (Add) is simply defined by the state being an integer and possible actions being a list of integers. For example a list of actions being [2,1,-1-2], the optimal min max path would be [2, -2, 2, -2, …], and alterations of this can be programmatically asserted in thousands of test cases. Further, this allows bugs such as non optimality ([1, -1, 1, -1, …]) or min max issues ([2, 2, 2, 2, …]) to be highly interpretable.

After implementing, debugging, and transferring MCTS into the assessment 2 skeleton code, we discovered that the number of iterations we could do was far below what would be effective. From testing Add, we found it needed around 150 iterations, with the actions defined above, to find an effective min max path to a depth of around 3. However Board Tetris has a significantly higher branching factor than this and would require significantly more iteration, but we found we could only do around 1 iteration a second, roughly in line with others from posts on the ed form.

However, although we had hoped there would be a simple bottleneck, all of our optimisations only allowed us to double the number of iterations we could do, revealing that the real issue was the branching factor of the problem and not something we would be able to diminish with naive optimisation.

We decided to make the switch to min max to create a more effective game playing agent, and used the same strategy of using the Add game to quickly debug in an enclosed environment away from the full logic of the skeleton code. After transferring this implementation into Board Tetris, we found is was too slow and so went back to implement alpha beta pruning. However we again found this was too slow.

We decided to make a tradeoff for performance at different states of the game, because when the board was less dense and you couldn't plan many moves ahead with the high branching factor we used a greedy algorithm at this stage. Then when the agent was in a dense board, with the danger of running out of moves to make, it would think ahead to make more intelligent moves at the cost of taking more time. If the time limit ever got close the behaviour would switch back to being greedy.

Heuristic
After a significant amount of tuning, we have settled on a few important aspects to the heuristic.

1 : suffocate
This restricts the number of free adjacent squares to the other player
It adds a negative value to the heuristic for every free adjacent square to the other player.

2 : square count difference
If there is the option to eliminate a row or column,
evaluate the change in the number of squares for each player.
It overpowers other heuristics, and adds a value to the heuristic according to this formula.

$ deltaThisPlayersPieces - deltaOtherPlayersPieces - MAX_PIECE_SIZE - 3

where -3 is a bias to delay eliminations.
to try get the opponent to place more squares in the column or row.

3 : elimination prevention
Early in the game, it is easy to use a number of line pieces,
especially if you are trying to cover a large space.
But this can leave you open for easy eliminations. This value limits your exposure to eliminations.
This adds a negative value to the heuristic for the longest columns or rows.

Bugs
Being limited on manpower, there was a few issues with our final submission. We found a bug close to the deadline where the agent wasn't seeing moves occasionally. We tried visually debugging by render the board and found that the logic for adjacent squares and empty adjacent squares was correct, but did not have time to find the bug in the 'place actions through squares' logic.

We think this is the reason our submission was struggling to beat random bots in the submission task, despite in our environment it being capable.

Performance evaluation
Our code is highly functional, with access to the most advanced algorithms including alpha beta pruning and Monte Carlo tree search. However it is not tuned enough to be highly effective. With a small number of fixes and some more tuning it could be significantly more performant.

For features of our heuristic, in testing …

+We found the suffocate feature to be highly effective at carving out a bubble for the agent to use when it needed later placements.

+Square count difference is highly effective in being conservative with when to perform elimination, not simply performing them when they just break even to do so, but waiting for the board to fill up for them to be more effective.

+Elimination prevention is an important strategy for surviving early game even with the most advanced opponents, as they simply can't perform enough eliminations to take out the agent. Further it aids late game, as although there will be eliminations, the heuristic minimises the downside of them by distributing squares across all columns and rows.

+The use of min max in dense stages of the game is important in avoiding move-less positions, and in forcing the opponent into them.

Optimisation

Applying numerous optimisation strategies to increase the number of iterations, primarily through profiling and optimising functions which contributed most to the run time, we made significant improvements. Our most effective changes were simply caching. We had already cached a list of rotated pieces to place, but we optimised this for dense board states. We then cached the left, right, up and down neighbours of each board square and every piece offset, and this has a significant performance benefit. We then reduced the average and best case complexity of the frequent function to apply actions to the board state from O(n^3) to O(n^2) where n is the board size on one axis, and this also unearthed a bug relating to simultaneous column and row eliminations.

Testing philosophy

Although we could have written a program to test our agent against random bots a hundreds of times, we thought it was better to simply look at the moves it was playing and evaluate them. This allowed us to effectively form a heuristic. We found that our original heuristic code was non optimal in a number of ways, including performing eliminations which eliminated more of the opponents pieces, but which ignored the fact that it lost out on placing 4 of it's own pieces. Further, we found that elimination prevention and number of available moves were generally opposing heuristics. So through testing decided to use elimination prevention early game, and min max late game.

Conclusion

While thoroughly limited in comparison to other groups, we managed to implement 90% of the code required and use the most advanced strategies available.