The University of Melbourne
COMP30006: Artificial Intelligence
Semester 1, 2024

Project Part B: Report

by Thanh Dat Nguyen (1370085) and Ernest Tay Jie Jun (1336997)

1 Approach

1.1 Primary Strategy Choice

With Tetress being a two-player deterministic game with perfect information, since the beginning it was clear to us that the two search strategies provided in the lectures would be our primary candidates for implementation.

- **Minimax with Alpha-Beta Pruning**: This was our initial choice, since it seemed like the "easier" choice to implement. As we made progress with this decision, its shortcomings soon became apparent: for an insignificant amount of game time, the branching factor of each position is quite large (150 - 250 moves), meaning within the roughly 2.5 seconds we have per move, we could only look ahead 1, 2 moves, making it not a very smart agent without further optimizations.
- **Monte Carlo Tree Search (MCTS)**: Alongside Minimax, we also decided to implement an agent using MCTS. After building MCTS, we realised that our iteration of MCTS would have to run at least 100 simulations for the generated move to be decently accurate, but with the limited time of 3 minutes per game, it ran way too slow. We tried to reduce the number of simulations, however due to the random nature of MCTS, reducing the number of simulations means that it ended up not being very effective.

Ultimately, since we were able to find more methods to improve our Minimax search (which will be explained in this report), our initial idea ended up being the final and superior choice.

1.2 State representation

Initially, the `State` class used to represent a node in our search algorithm was practically the same as Part A + a few extra information to improve searching time. The board uses:

- A dictionary with the key being a `Coord`, and the value being the `PlayerColor` at that coordinate.
- `List`s to keep track of the number of squares occupied in each row/column.
- The number of `RED`/`BLUE` squares on the board.

This remained the same throughout until the implementation of the Transposition Table and Bitboards (this will be expanded upon in their respective sections).

1.3 Evaluation function

Tetress is a zero-sum game, so if the game has reached a terminal state, the function will evaluate and return a `RED_WIN` (+1), `BLUE_WIN` (-1), or `DRAW` (0) accordingly.

As for the evaluation of a position, after assessing and playing Tetress, we reached the conclusion that there are 2 main ways to win the game: having more squares of your color than the opponent, and forcing the opponent into a position where they have no legal moves. Naturally, this meant that our evaluation function should incorporate both of these factors. Therefore, given a board state, our agent will evaluate it in the following manner: num of red squares - num of blue squares + the number of moves that can be made from this position (this will be positive if it is `RED`'s turn, and negative if `BLUE`'s turn).

1.4 Time Limit per Move

Since the maximum number of moves per player in a game would be 75 and each player have 180 seconds to "think", each move should take an average of $180/75 = 2.4$ seconds.

Setting this as a time limit for every move in the game wouldn't be ideal, since different phases of the game might require more in-depth searching. When there are not many squares placed on the board, the number of possible moves is higher and it is more difficult to make a move that immediately results in a loss. The inverse is true when the board is "denser". Therefore, our time limit policy is as follows:

- When there are less than 50 squares: The time limit is 1.5 seconds.
- When there are less than 80 squares: The time limit is 2 seconds.
- Otherwise: The time limit is 3 seconds.

This ensures that when there are fewer moves and it is easier to lose, the search is expanded to make sure that more nodes and possibilities are checked.

1.5 Minimax with Alpha-Beta Pruning

Our implementation of this search algorithm was more of a Negamax algorithm, and loosely followed the pseudocode from this Wikipedia article. Additionally, there was a minor improvement: If the search is on a leaf node (i.e. the `depth` variable is 0) and the number of moves available from this position is <= 5, the search is extended by a depth of 1 to better circumvent the Horizon Effect (when a move is evaluated as good at the cutoff depth, but a deeper search reveals that it will lead to a bad situation).

Theoretically, assuming the average branching factor is 150, the time complexity of this implementation is $O(150^{depth})$. With Alpha-Beta Pruning, this becomes $O(150^{3depth/4})$. Practically however, using pure Minimax with Alpha-Beta Pruning, testing showed that our agent could look ahead 1 moves (2 if we are lucky) most of the time when the number of available moves are high. When the board is denser and there are fewer moves to make, it could look ahead 2 or 3 moves.

1.6 Iterative Deepening

The idea for Iterative Deepening: Minimax search is ran first with depth 1, then depth 2, and so on until the time limit runs out. The result obtained from the last full search iteration will

be used.

At first, this might seem counter-intuitive, since time is being "wasted" searching at lower depths when it's the higher depths' results that are more useful. And implementing Iterative Deepening just by itself shows that this is the case: no improvements were observed, and at times the agent performed worse. But the benefits this feature brings will be more apparent when the next features are implemented.

1.7 Transposition Table with Zobrist Hashing

In a game like this, many move combinations can reach the same board state. This means that a decent chunk of the agent's search time will be spent evaluating positions it has evaluated before. A Transposition Table aims to fix that by storing positions and their evaluations, so that they can be used later and reduce the search time. And with Iterative Deepening, this improvement can be even more pronounced since we can use previous search iterations' evaluations in our Transposition Table, reducing the later iterations' search times. With Alpha-Beta pruning however, the evaluations are not always exact, since a state might be evaluated at a beta-cutoff (where moves are pruned due to it being too good), or when no moves in a position end up being better than a previous position. So, the evaluation from an entry in the Transposition Table can only be "trusted" when the depth at which it was searched is at least as high as the current search depth, and the following conditions for these 3 categories:

- `EXACT`: This is when a best move is found, and we can trust that the evaluation is accurate. No further conditions are needed.
- `LOWER_BOUND`: This is when a beta-cutoff occurred, and we only know that the actual evaluation is at least as good as the evaluation we got. We can trust this evaluation when it is higher than the current beta value.
- `UPPER_BOUND`: This is when no moves in the entry's position were better than a previous position. We can trust this evaluation when it is higher than the current alpha value.

In order to efficiently lookup the values for these moves, we need to implement our Transposition Table as a hash table, meaning we need to find an efficient way to hash a board state. This brings us to Zobrist Hashing, a common method used alongside Transposition Tables. The idea is the placement of a square on the board can be represented by an integer. There are 11*11 = 121 squares on the board, and 2 colors to place on a given square, so we can represent them by 121*2 = 242 integers, with an additional integer representing when the current player is `BLUE`. We can then use the XOR operator with these numbers to generate a hash key for a given board position. This key can be updated every time a move is made by XOR-ing the current key with the integers associated with the squares that were just placed. Any squares deleted can also be handled by XOR-ing with the integer associated with them. Since the XOR operator has the property: if a XOR b = c, then c XOR b = a, as long as two positions have the same exact square placement and current player, it doesn't matter how the position is reached, they will always be hashed to the same integer.

Currently, the Transposition Table doesn't have any collision handling, but any of the usual methods can be employed.

1.8 Principal Variation + Move Ordering

Since most moves available from a game state are likely to not be very good, the agent's shouldn't waste too much time searching and evaluating them. With Iterative Deepening, we

can keep track of the best move we found from the previous iteration, and with a Transposition Table, we can find the previously found best move of the current position (if we have evaluated the position before). These two moves can be ordered and put to the front of the move list to be searched and evaluated first. Additionally, any moves that are not these two (these two moves are known as the Principal Variation) can be searched with a "null window", where beta = alpha + 1 to speed up their search time. However, to ensure that we don't miss a potentially good move to this null window, if the result from these moves are still good enough, we will still search them in full.

1.9 Bitboards

After all of these previous features, we noticed that the agent still took a lot of time to generate a move at higher depths. Using python's cProfile tool showed us that the majority of the agent's thinking time was spent in the move generation process. From the start, our move generation was very rudimentary: the agent iterated through all of its squares on the board, finding all surrounding empty squares, and generated all Tetromino piece types on them. Generating the pieces every time like this took a lot of time, which led us to using Bitboards, a feature often employed in chess engines.

Before the game starts, in the pre-computation phase, the agent will generate bitboards: integers whose 121 bits represent the 121 squares of the board, where a 1 bit means the square is occupied, a 0 bit means that it is not. For each square of the board, the agent will generate bitboards that represent all possible piece placements around that square (a total of 164 pieces per square). Each `State` will also have their own bitboard representation. During move generation, the agent will iterate through all of its squares on the board, and retrieve all the bitboards of pieces that can be placed on each square. For each bitboard retrieved, the AND operator will be used on it and the current state's bitboard to check if the piece is a legal move.

This drastically reduced our search time since all the moves are essentially "pre-generated" and we don't have to generate them every time Minimax is called.

2 Performance

2.1 Time

With the time limits described in the previous section, it seems unlikely that our agent would fail due to the 180-second time constraint. If every move is limited at 2 seconds, each agent would need 150 seconds max to finish the game. That leaves around 30 seconds to spare, and since there shouldn't be too many 3-second limited moves, the agent should have time to play out 150 moves if needed. Additionally, most test runs of our agents against their previous versions didn't reach 100 moves, let alone 150, so time shouldn't be an issue.

2.2 Space

The size of our Transposition Table is set at 100,000 and the pre-generated bitboards are all integers, so with the very generous space limit of 250MB, we should never be at risk of running out of space.

2.3 Intelligence

With the current version of our agent, for each time limit, it can look ahead the following number of moves:

- 1.5 seconds and less than 50 squares: 1-2 moves (mostly 1 at the start of the game)
- 2 seconds and less than 80 squares: 2-3 moves (sometimes 4)
- 3 seconds and less than 121 squares: 4-7 moves

These number of moves are obviously more dependent on the number of available legal moves, but these are relatively close estimates. This should make the agent moderately averse to making moves that loses within the next few turns, especially since the search depth is extended if the number of legal moves are low.

The evaluation function currently is quite basic. It pushes the agent to make moves that either limits the enemy's moves or allows the agent to have more move choices. This is a decent enough strategy to not lose quickly while still making moves that force the opponent into difficult positions. Against an agent that makes completely random moves, it wins basically all of the time. Against an agent we created that always makes moves that limit its opponents' moves, it only loses 1 to 2 times out of 10. The same is also true against our version of MCTS. This result is somewhat satisfactory, but we suspect that against agents with better evaluation functions, it won't perform too well.

3 Other

3.1 Unimplemented Considerations

There were a few features we considered including in our agent, but ended up not following through due to various reasons.

- **Quiescence Search**: Quiescence Search is one method to solve the previously mentioned Horizon Effect, where even if the original search depth is exhausted, the search would continue for moves that were considered "noisy" (for example in chess, these would be captures, or moves that put the king in check), until a "quiet" position is reached (position without noisy moves). Our current depth extension policy is essentially a simpler form of Quiescence Search, but initially we wanted a quiescence search that kept searching until there are no longer moves that clears lines. In the end, we didn't think it was useful since line-clearing moves don't necessarily put the agent into any immediate danger of losing the game.
- **Collision Handling for Transposition Table**: Right now, the Transposition Table doesn't do any collision handling, and simply replaces the old entry with the new one if there are any collisions. This is likely not ideal, but since the size of the table is quite large + due to us wanting to move on to other features, this feature was never implemented.
- **More uses for Bitboards**: This is not really a unimplemented feature, but more of a wish that we could have done more with Bitboards, since it felt like using Bitboards have helped our agent drastically. But we couldn't think of another way to improve upon how we are currently using them, so the current version ended up being all we could do.

3.2 Acknowledgements

A large part of how we decided to approach this project was our familiarity with chess. We researched how people in the past have made of improvements to their own chess Minimax engine, and adapted it to Tetress. Notably, this video and its sequel by Sebastian Lague, have inspired many features that we ended up implementing.

A few articles that we read to further understand the concepts of these features are also included here:

- Every Wikipedia article on these features
- MediocreChess' guide to Transposition Tables
- Lars Wächter's article on Zobrist Hashing