Final Project - COMP 424

Jimmy Khairallah - 260724798

April 13 2021

1 How the Agent Works

The following is how the code is structured with the classes' corresponding methods:

```
/
├── MyTools
│   ├── Tuple computeScore(Piece p, Neighbour n, int score, int
│   │   pieceCount)
│   └── int getScore(PentagoBoardState pbs, int player, int alpha,
│       int beta)
├── StudentPlayer
│   └── Move chooseMove(PentagoBoardState pentagoBoardState)
├── AlphaBetaPruning
│   └── Tuple alphaBetaPrune(PentagoBoardState pbs, int player,
│       int alpha, int beta, int depth)
└── Tuple
```

We were given a time constraint for this project where every move should take at most 2 seconds (excluding opening move). At first, MiniMax was the impulse decision, given the game involved 2 players playing against each other. However, due to the given time constraint, I decided to go with Alpha Beta Pruning since the latter reduces the amount of computation and searching during minimax. Also, a depth-first search shows more promise with a 2-seconds time constraint.

Firstly, the agent loops through all legal moves (legal moves are generated by an existing function in the codebase) and checks if a move is a winning move by processing the given move on a cloned board state. If it does, then it returns that move. Otherwise, the agent runs alpha beta pruning on the board state with a depth of 3, which in short returns the best move that maximizes the score.

For the algorithm, I got the Java code from this link where a node represents a board state and a move is the action/operator that is executed with boardState.processMove(move).

I also created a Tuple class with generic types so that it can be used throughout the code (since Java doesn't have a tuple class):

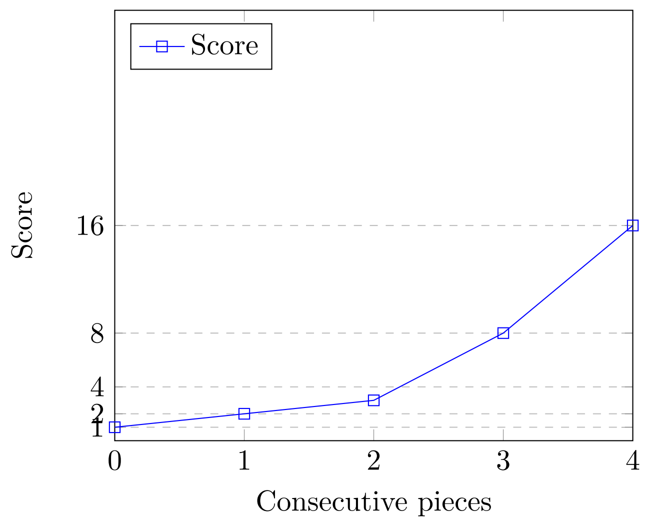
```
   public class Tuple<T, F> {

   public T key;
   public F value;

   public Tuple(T k, F v) {
      this.key = k;
      this.value = v;
   }
}
```

For the evaluation function (i.e the logic behind calculating the score to use for alpha beta pruning), all of the logic is written in MyTools.java. The score is evaluated as the following: $2^{numberOfConsecutivePieces}$ whether horizontally, vertically, or diagonally. The score can be decreased the same way if the opponent has consecutive pieces.
The reason I went with this evaluation function is because the change of score with respect to the number of consecutive pieces is very realistic when using it with a base of 2, since longer consecutive pieces are more valuable to complete rather than having multiple short consecutive pieces that would most likely result with a draw:

Change of Score with respect to Consecutive Pieces



Hence in alpha beta pruning, I loop through the legal moves and check every score evaluated in a cloned board state, and return the move that yields the highest score. That way, the max player will only update the value of alpha, and the min player only updates the value of beta. When the 2 seconds end, the algorithm returns a tuple with the recently updated alpha (highest score) with its corresponding move, or a tuple with the recently updated beta (lowest score) with its corresponding move.

Previously, I had a different evaluation function that was overly complicated and less optimal in terms of time complexity, where the score was calculated based on hard-coded board states with a non-generic heuristic. This resulted in the code taking more than 2 seconds to return the best move, hence returning a move with a worse heuristic. I played against some classmates with the previous evaluation and lost around 90% of the time. Afterwards, I fixed my evaluation function and tested it against the old evaluation function and won 100% of the time, which motivated me to stick with the newest evaluation function.

2 Theoretical Basis of the Approach

2.1 Algorithm Design

First algorithm that comes to mind in a game, specifically a 2-player turn-taking game is minimax. As stated in the book: The minimax algorithm

computes the minimax decision from the current state and is best used when we are encountering turn-taking games. (Russel, S. Norvic, P. Artificial Intelligence A Modern Approach 3rd edition. Chapter 5, Section 2.1)
In order to get a more optimal result from minimax (since the number of states it has to examine is exponential in the depth of the tree), we optimize the searching using alpha beta pruning. "The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of pruning to eliminate large parts of the tree from consideration." (Russel, S. Norvic, P. Artificial Intelligence A Modern Approach 3rd edition. Chapter 5, Section 5.3)

As a result, the final algorithm's design was influenced by those 2 AI approaches.

2.2 Heuristic Function

As mentioned earlier, the heuristic function used to calculate the score of a given state uses the following form: $2^{numberOfConsecutivePieces}$. For example, if my agent has 3 consecutive pieces horizontally, 2 consecutive pieces diagonally, 4 consecutive pieces diagonally, the score would add up to $(2^3 + 2^2 + 2^4) = 28$. The same applies for the opposing agent's pieces where the overall score is subtracted instead of added.

3 Advantages and Disadvantages

3.1 Advantages

The main advantage of my agent's implementation is that it is optimal with the given time constraint of 2 seconds per move. Since minimax doesn't prune any branches and has to execute all possible searches, the time complexity of alpha beta pruning proved to be a better implementation over minimax. In addition, I can flexibly change the given depth if the time constraint is longer, hence resulting in a more optimal move to execute.

3.2 Disadvantages

One disadvantage of my implementation is that I don't store data of visited states, hence doing the searches all over again in every new turn. This could be improved if I store data files that can be read before searching for a move in order to skip over searching for the non-optimal moves.

Another disadvantage can be found in my heuristic function, where I prioritize a vertical solution of 5 consecutive pieces, as shown in my code:

```
  for (int i = 0; i < 6; i++) {
            for (int j = 0; j < 5; j++) {
                // |
                Tuple<Integer, Integer> verticalPair =
                    computeScore(pbs.getPieceAt(j, i),
                    pbs.getPieceAt(j + 1, i), score,
                    verticalPieceCount);
                score = verticalPair.key;
                verticalPieceCount = verticalPair.value;
                // --
                Tuple<Integer, Integer> horizontalPair =
                    computeScore(pbs.getPieceAt(i, j),
                    pbs.getPieceAt(i, j + 1), score,
                    horizontalPieceCount);
                score = horizontalPair.key;
                horizontalPieceCount = horizontalPair.value;
            }
        }
```

Hence, if the opposing agent can manage to successfully and consistently block vertical consecutive pieces, then there is a higher probability that my agent would not win against the opposing agent.

4 Improvements to Submitted Implementation

As mentioned earlier in the disadvantages, my implementation does not store data files that relate to visited states. Hence, an improvement over the current implementation is using IO to read and write to data files that can store information about the visited states so that in the next turn, my agent would skip over searching already-searched states and try to look for unvisited states, hoping for a more optimal solution. This is also known as dynamic programming.
This is a trade-off between time complexity and space complexity, where the time constraint is a time-complexity matter. So we can use the unused space complexity to store those files.

Another improvement to the agent includes learning from previous game by storing game files and learning from them to better optimize for later games. However, this is not permitted as stated in the project description.

This would result in a drastically better time complexity and stronger agent that reduces its chances of losing the more the games it participates in.