1. How My Program Works and Motivation for My Approach

I decided to implement an Iteratively Deepening Alpha-Beta Search Algorithm enhanced with a Transposition Table. I will go into more detail of how my program works by explaining how its design came to be.

1.a Choosing the Base Algorithm for My Agent

The first question I asked myself was if the main algorithm of my AI should be Monte Carlo Tree Search (MCTS) or Minimax Search. While MCTS seemed like it would theoretically perform better because of the high branch factor of the Pentago Game, I wanted to make sure. To compare both potential solutions, I implemented two basic agents, one was using MCTS while the other was using basic Minimax Search. What I found was that surprisingly, even if the Minimax Search agent was using a very simple evaluation function, it still won most games. This can probably be explained by the fact that minimax, tries its hardest to avoid moves that would lead to the opponent winning while MCTS just chooses its moves based on win rate probability. Nonetheless, I started building my own agent around minimax Search because of this result and the fact I found it simpler and more intuitive than MCTS.

```
if (boardState.getWinner() == currentPlayer) {
    score = Integer.MAX_VALUE;
} else {
    score = Integer.MIN_VALUE;
}
```

The simple evaluation function I was using

1.b Implementing Alpha-Beta Search

My first step to improving my basic Minimax Search Agent was to implement alpha-beta pruning. As we saw in class, the Alpha-Beta Algorithm is an improvement on Minimax. The algorithm finds the maximum value in the leaves and prunes the rest of the sub-trees when it is known that they will not change the maximum. On the other hand, Minimax tests all the leaves in the entire tree for the maximum value.

1.c Choosing A Better Evaluation Function

The next step was to write a better evaluation function. With a better evaluation function, each board state value is closer to what it truly is. This leads to a better search result when we are eventually forced to cut-off the search because of the time limit. Additionally, with a good evaluation function, our agent can both attack and defend by realizing when the opponent's position is better than its own.

It was hard to build a reliable evaluation function for Pentago because each move can totally change the situation. I decided to keep it simple so I could extend the search depth instead of wasting my time having to compute a complex evaluation function.

My function evaluates which player is more likely to win the position by weighing some straightforward patterns:

1. 5 pieces in a row is best
2. 4 pieces in a row is next
3. Followed by 3-in-row
4. If some number of consecutive pieces cannot lead to a win because it is blocked, it is worth less points
5. Pieces at the center of a quadrant are better then pieces on the edges of the quadrant.

I did not know the weights I wanted to accord each pattern. Thus, I built my evaluation function to be easily adjustable so I could play around with different values for each pattern. This made it possible for me to get multiple evaluation functions I could test against each other to see which worked best. I will go into more detail in a later part.

1.d Implementing Iterative Deeping

Next, I decided to implement Iterative Deepening. This is because regular Alpha-Beta is set to search at a fixed depth which is not suitable when the game is time-constrained. If the depth is set too high, the algorithm might not finish running and will most likely not return the best move. If the depth is set too low, you are missing out on potential refinement of the best move with following iterations. With iterative deepening however, depth is gradually increased meaning we return the best move even while being time constrained; the best move can be adjusted at each level.

```
// start at depth 1
int depth = boardState.getTurnNumber() == 0 ? 4 : 3;
// incremental depth minimax
while (true) {
    // run minimax at current depth
    TTEntry res = minimax(boardState, depth, true, Integer.MIN_VALUE, Integer.MAX_VALUE);
    // if score is better change result
    if (res.getScore() > bestResult.getScore())
        bestResult = res;

    if (cutoff) {
        break;
    }
    depth++; // increment depth
}

return bestResult.getBestMove();
```

Simple code for Iterative Deepening Alpha-Beta

1.f Implementing a Transposition Table

Especially with iterative deepening, a board position can be evaluated more than once. The number of re-evaluated board states is increased as the game goes along. This creates a lot of redundant computing. We might compute minimax for a state we have just seen in the previous iteration or in past searches. With a transposition table, we store results of running minimax on a state meaning we can potentially avoid having to re-compute the result if that state ever comes

up again. I chose to store all minimax results in a Hash Map. Hash Maps are fast and very efficient for storing elements that do not rely on partial ordering. Thus, before running each iteration of Alpha-Beta Search, the agent searches the Transposition Table to see if that state was computed beforehand. If so, there is no need to calculate the resulting evaluation at that state.

2. Analysis of my Approach and Algorithms

In this section, I will elaborate on my algorithms and explain their design inspirations.

2.a Alpha-Beta

Alpha-Beta search recursively processes each legal move from the current game state and computes a score based on the board state after playing the move.
My implementation of this algorithm was based on the following code I found at [1]. I had to modify it so it would work with our Pentago game and would terminate after the timeout was exceeded. To do this, I simply calculate the endTime (time the agent should return a move) at the start of its turn. This made it possible for me to find out if I had exceeded the endTime at any point during my algorithm.

```
// returns true if the timeout was exceeded
public static boolean timeExceeded() {
    return (System.currentTimeMillis() > endTime);
}
```

Function to indicate if Alpha-Beta Search must be cut-off

I will now go into the theoretical basis of Alpha-Beta Pruning. To understand Alpha-Beta search, we need to understand Minimax first. The Minimax algorithm is a brute-force search through the entire game-tree to return the best move for the player. The algorithm tries to simulate the player trying to maximize his future score and his opponent trying to minimize that.
To return the best move, Minimax evaluates a given position by expanding the game-tree up to a certain depth. The leaf game states are given a value based on if the player is more likely to win the position or not. Following the principle of the player maximizing his value while the opponent minimizes it, these values propagate through the tree recursively until the best move is returned.

As we saw in class [2], Minimax is very computationally expensive because given we search until depth d for a game with a branching factor b, searching the entire tree would take $O(b^d)$.

Alpha-Beta can cut down the computation time drastically. Alpha-Beta is built on the same foundations of Minimax but avoids visiting nodes in the tree that will not influence the final result. We explored this algorithm in class [2] and realized that with a good move ordering, the search depth can be halved. This equates to a time complexity of $O(b^{d/2})$.

2.b Evaluation Function

As I mentioned in the previous part, my function evaluates which player is more likely to win the position by weighing different patterns. To allow myself to play around with multiple potential evaluation function, I needed to be able to get any potential pattern from the board I wanted so I could analyze it.

I decided to turn each diagonal, row and column from the board into strings. With that, I could then split them based on the pieces to return any pattern I wanted. For example, as the white player, given the following row "021111" I can split it based on the black pieces to get consecutive blocks of either white or empty pieces. With the length of those blocks, I can evaluate if the row is blocked meaning it couldn't lead to a win. By splitting based on the empty pieces, I can count the number of consecutive white pieces and return the corresponding value.

I implemented everything myself except for the code to return the diagonals from a square matrix which I adapted from [3]. From an int[][] representation of the board, I can get diagonals of length > 2 as strings.

With this system in place, I could play around with different evaluation function for different patterns. I could lower the score for blocked patterns, only account for patterns of 4-in-a-row or higher, I could calculate both player's positions and compare them to get an overall rating of the board…

After some testing, I finally ended up with the following evaluation function:

Pattern	**Weight**
5-in-a-row	1000000
4-in-a-row	10000
3-in-a-row	1000
Piece at center of a quadrant	50
2-in-a-row	10
4-in-a-row blocked	100
3-in-a-row blocked	50

Weights for the current player

Pattern	**Weight**
5-in-a-row	-1000000
4-in-a-row	-10000
3-in-a-row	-1000
4-in-a-row blocked	-100
3-in-a-row blocked	-50

Weights for the opponent

The final value for a board position was obtained by summing up all the weights after finding all the patterns on the board. This evaluation function balances attacking and defending by lowering the final score if the opponent's position is good. It does not just focus on the pieces of the current player but all the positions on the board.

2.c Transposition Table

I already elaborated on the theoretical basis of transposition tables in part 1. Basically, whenever Alpha-Beta finishes calculating a state, it stores the result in a Transposition Table. With this, at the start of each future iteration, I search in the Transposition Table to find out if the current state was ever encountered before and if it was, I can return the cached result directly.

I wrote the entire code for my transposition table myself with a lot of trial and error in order to get it right. The Transposition Table is a Hash Map where I store each boardState along with the score and best move I computed from that position. Additionally, each entry has to have the depth

at which the result was returned. This is important because I am using iteratively deepening search. With no way to account for the depth of an entry, a boardState's best move would not get refined in future iterations. I also added the turnNumber for each entry so I could clean the hashMap before each turn to remove entries that would never be used because we had passed that turnNumber.

```
final class TTEntry {
    private PentagoMove bestMove;
    private int score;
    private int depth;
    private int turnNumber;

    public TTEntry(PentagoMove bestMove, int score, int depth, int turnNumber) {
        this.bestMove = bestMove;
        this.score = score;
        this.depth = depth;
        this.turnNumber = turnNumber;
    }
```

The value of each entry in my HashMap

I encountered a problem with returning the same key for 2 boardStates that should be equal. While 2 boardStates could have the same elements, they will not be equal. To fix this, I decided to use each boardState's board as a string for the keys of my HashMap.

```
// turn boardState into an int[][] board and return its string representation
public static String getBoardString(PentagoBoardState boardState) {
    String ret = "";
    int[][] board = getBoard(boardState);
    for (int[] i : board) {
        ret += Arrays.toString(i);
    }
    return ret;
}
```

Function that returns the key of a boardState

3. Advantages and Disadvantages of My Approach

3.a Advantages

- Time efficient
  With the benefits of Alpha-beta combined with iterative deepening and a Transposition Table, my Agent computes minimax very fast. This can be seen by the fact my search algorithm reaches depths 5 in the 1st move and up to depth 4 in the next moves. This is much higher than with my original minimax version that could only reach depth 2 within the time limits. With a higher depth, the returned move's value is a much better estimate of its true value.

- Attacks and Defends
  With my evaluation function, my agent is able to account for the opponent's good positions. It prioritizes exploring rows and columns that are not blocked while playing some defensive moves to block the opponent.

- Adaptable
  My agent is adaptable and can still work very efficiently even if the time constraint parameters are changed. It could also work on different versions of the Pentago game like the original version with only left and right rotations because changing the rules will not impact the way it functions.

3.b Disadvantages

- The Main Disadvantage of Alpha-Beta Search
  As we saw in class [2], the main problem with Alpha-Beta assumes that both you and your opponent are playing optimally with respect to the same evaluation function. This is almost never the case which can lead to my agent skipping a promising state because it wrongly assumed my opponent's actions.

- There could be a better evaluation function
  There is no way to know if my evaluation function was the best. My algorithm might work even better with another function. I wish I had had more time to test different evaluation functions against each other.

- My Agent relies heavily on the PentagoBoardState class
  My agent saves and computes on a lot of different instances of the PentagoBoardState. As I will elaborate in the potential improvements, all computations could be made way more efficient by not relying on this inefficient class.

- Disadvantaged with a smaller time constraint
  With a smaller time-constraint, I think my agent would perform worse than an improved version of MCTS.

- Disadvantaged when playing 2^{nd}
  While Pentago is a 1^{st} player wins game [4]. My Agent still appears to lose more than it should especially when it plays 2^{nd} . I think this is linked to the fact my agent prioritizes attacking more than defending.

- Agent does not recognize all good patterns
  While playing against my agent, I realized that some great positions are only interpreted has good positions by my agent. I only found one of those instances and tried to account for it in my evaluation function. However, I do not know if there are some other similar patterns.

```
// harcoded some great patterns my eval function doesn't detect as great positions just good enough
String[] greatPatterns = {"10111","11101", "11011","20222","22202", "22022"};
for(String pattern : greatPatterns){
    if(consecutiveBlock.contains(pattern)){
        return getEval(true, 4);
    }
}
```

Hardcoded some patterns my agent was not recognizing

4. Alternative Approaches

My first approach to obtain my patterns for my evaluation function was a modified version of what is used in the PentagoBoardState.updateWinner() method to check for a winning pattern. However, it was hard to change the code to obtain all the patterns I wanted. Especially, the blocked patterns that would never lead to a win. I therefore opted for my own solution using strings.

I tested my agent against multiple different versions of itself trying to improve certain aspects of it. This led to some small improvements such as hard coding the 1st move. One more was starting search at depth 4 for the 1st turn to populate the HashMap with very good values. I also tested different methods of cleaning the Transposition Table before each turn.

There were also some attempts at move filtering and move ordering before running minimax. While theoretically, these should have improved my agent, it turned out that it ran better without those. This is probably due to the fact my move sorting was way too computationally expensive which meant that any potential benefit was lost.

5. Future Improvements

The 1st potential improvement would be to add some simple move ordering. I found some examples of different simple move orderings that are done with chess AIs [5]. Some of these could potentially be adapted to fit my own agent.

Another improvement could be to add an undoMove() method similar to the processMove() method from the PentagoBoardState. This would reduce the number of instances of the class since there would be no need to compute so many clones.

Next, I could implement a bitboard instead of the current PentagoBoardState class. I was using a naive approach to construct the board using a 6×6 array while the same could be done using two 64-bit Integer [6]. Apparently, all my patterns could be detected in a much simpler manner with bitboards. There would be no need for the complex string manipulation I implemented. All my patterns could be detected through bitmasks.
Additionally, a lot of the elements of the PentagoBoardState could be derived from the bitboard when needed. All of this contributes to a bitboard leading to greatly improved performance.

I could also increase the accuracy of my heuristic's function with an evaluation function that would give each pattern its value based on results from a lot of played matches. With this data driven approach, the computed scores would be a much better estimation of the board state.

Finally, I could implement my agent in a faster language such as C or C++. With these languages using less memory, my agent could compute results faster and could maybe reach higher depths.

6. References

[1] https://stackoverflow.com/questions/64336251/how-to-stop-alpha-beta-with-a-timer-in-iterative-deepening
[2] Lectures
[3] https://stackoverflow.com/questions/20420065/loop-diagonally-through-two-dimensional-array.
[4] https://perfect-pentago.net/details.html#algorithms
[5] https://www.chessprogramming.org/Move_Ordering#Typical_move_ordering
[6]https://web.archive.org/web/20180822203740/https://chessprogramming.wikispaces.com/BitBoards