1. How my program works, and a motivation for my approach

General Description of How my AI works:

My AI is a hybrid of Monte-Carlo Tree Search and Alpha Beta Pruning.
I use Monte-Carlo Simulations with Upper Confidence Trees (UCT) to filter out the worst moves and then, I run Alpha Beta Pruning with a custom Heuristic to calculate the best move from the best available. I give 800 ms to run Monte-Carlo and up to 1800 s from the start of the computation for the alpha-beta (1 second) before I return the best move I have calculated.

I start by generating all the legal moves from the current board state. remove all moves that lead directly to a Defeat. The rest are added into a new list. If a Pentagomove directly leads to a Win, I return it right away and skip and steps that come next.

Now given a slightly filtered list, I run Monte-Carlo Search on them:
I start with the default policy. Then I choose the next move to simulate again by balancing Exploitation and Exploration.

After time's up, I find the best K-moves (K = 40 by trial and error). I choose them based on the ratio of Win/TotalSims for that Move. Then I run alpha-beta pruning with depth 2 on the remaining moves. The base case is depth 0 where I calculate the heuristic function at that state or GameOverCheck.

Heuristic Function:

The evaluation of the heuristic at the leaf state of the exploration tree adds the worth of the AI's pieces and subtracts the worth of the opponent's position's worth.

The Horizontals and Verticals, and Middle Diagonals have the same heuristic calculation since they are symmetric. For example, I have combined both "Top" rows and greedily counted how many pieces of "my color" are on them. I add multipliers for every consecutive 2,3,4, and 5 pieces.

For the side diagonals, I calculate the strategic positions that form them: 1 corner of a quad (3X3) and a 2 X 2-piece-diagonal in 2 different quads:

```
_|X|_
_|_|X
X|_|_
```

The 4 corresponding side diagonals are symmetric.

I calculate how many "two's", "corners" and "free" cells I have to add up the cost of the current board state. Two "twos" is like 4-consecutive pieces. One "two's" and one "corner" is like 3-consecutive.
When I get 5 consecutive or a combination of 3 and 2. I return a very high cost.

Some hardcoded aspects:

I hardcoded the time for Simulations and alpha-beta. The depth of alpha beta is fixed to 2 but increases to 3 after the 10th round (which I found by trial and error). The depth was fixed so there is enough time to go through all the moves and find the best value. White starts at depth 2 since I want to be able to see more moves but not as far away into the game. Black starts at depth 3 since AI is ready to run out of time while checking moves (see less moves) but see further ahead - acts as a defence mechanism.

Opening moves:

I also hardcoded the 1st 3 moves for White and 2 moves for Black based on my understanding of the game and belief that those positions are the best opening moves.

Special Starter White Piece Strategy to for optimal game play (similarly in all directions with horiz,vert,diag):

1st move (b/w)	2nd move(b/w)	3rd move (white only)
_|_|_	_|_|_	_|_|_
_|x|_	_|x|_	_|x|_
_|_|_	_|_|_	_|x|_
_|_|_	_|_|_	_|_|_
_|_|_	_|x|_	_|x|_
_|_|_	_|_|_	_|_|_

The center of a quadrant can be a set up for a horizontal, vertical, and middle diagonal combination. It is the least affected by a swap.

If I am the White player, I go first. This gives me an extra move advantage over my opponent, and is a great way to start an “attack”. That is why the 3rd move starts to set up a winning position around one of the center cells I played in the previous 2 turns.

Motivation this approach:

We covered in class 2 main game playing algorithms: Alpha Beta Pruning and Monte Carlo Tree Search. I decided to use Alpha Beta Pruning to compute the best move. Nevertheless, the algorithm was too slow. Then I decided to create a filtering process to remove the most unlikely moves to be chosen, focusing on computing less moves but better moves.

1. Remove from all the legal moves, that lead to a loss.
2. Use Monte Carlo Simulations to compute the games won / total number of games simulated. Using Upper Confidence trees, I simulate more often more promising moves.
3. Moves with highest wins/games simulated get selected for pruning.

2. Theoretical Basis of the Approach

I used Alpha Beta Pruning based on minimax cutoff algorithm and Monte Carlo Tree Search around the next best move. I took from lecture slides all the theory behind my algorithm. My code for Alpha-Beta pruning is based on pseudocode offered on wikipedia: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

For alpha beta pruning, if a path looks worse than what we already have, discard it. If the best move at a node cannot change (regardless of what we would find by searching) then no need to search further.

With Monte Carlo Algorithm I use randomness to generate a sample for estimation of the goodness of moves. I also used MCTS to select a promising leaf node in the search tree using a tree policy.

3. Advantages, Disadvantages of my Approach, Expected Failure Modes, Weaknesses of my Program.

The advantage of my algorithm is that it is a hybrid. It has the properties of minimax, with the speedup of alpha-beta pruning, and the properties of Monte carlo. Hence, this algorithm tries to compensate for each other's flaws.

The disadvantages are the fact that it is really tight on time to find a good move. Also, the simulations give a non deterministic estimate on the worth of the moves but in no way are a certainty. The filtering process might work generally, can potentially oversee a good move in the long run that may seem bad at first glance.

The issue with alpha-beta pruning is that it is very expensive. It relies on having a reasonable evaluation function. Assumes that both you and your opponent are playing optimally with respect to the same evaluation function, which is clearly rarely the case. Most AIs are coded differently and interpret the same boardstates differently. It is also hard to design a good evaluation function in a game like Pentago where this is some non-determinism in the game environment.

<u>Pros:</u>

1) It is Time efficient: Using the filtering process, I don't waste time on bad moves. If a direct move on the current board state lead to the AI's loss, it will never execute it.
2) It only gets better as the game progresses: At the beginning there are alot of moves to do. The AI filters out the best moves. It sometimes doesn't have time to try them all, but as the game progresses there are less possible moves allowing the AI to to simulate the moves more frequently. More frequent simulations lead to better estimations on the value of a move. The search tree can also get deeper. The AI searches up to depth 3 starting from the 3rd round.

3) It Attacks more than defends: the heuristic calculated adds what is good for the AI and subtracts what is bad for it. But it might decide to build up pieces consecutively instead of blocking the opponent who is setting up 4 in a row with 2 spots available for a 5th: |_|X|X|X|X|_|
   After playing the AI against itself, I notice White always wins since the Black one doesn't defend as well as it tries to attack (by building good positions)

Cons:

1) It times out: I have a flag that stops the calculation once 1.99 seconds are up. This forces the AI to play the best move it has calculated so far (that might not be the best move). So it may not have tested all the moves available by that time.
2) Static parameters: it doesn't adjust to the game. The time to simulate and time to prune are fixed in the code as constants and don't adjust depending on the type of game and the current positions on the board.
3) In the rare case where no good moves available (happens very rarely), AI is forced to play a random move.

4. Other approaches attempted

The initial approach was pure alpha beta pruning. The problem is that the branching factor is so large at the beginning of the game that it played so badly that it would not "last" until the game progresses and there are less moves.

Moreover, I had to spend a lot of time developing a heuristic function that became way too time consuming as well. It would divide the board into quadrants and exhaustively check every position on the board. It was full of if-statements which made the logic hard to follow and maintain the code.

To speed up the pruning I decided to remove all losing moves. Then, I decided to use Monte Carlo simulations to find a subset of "best moves" to then plug into the alpha-beta.

Then, with all the constants, time limits for simulations, pruning, and the subset of best moves, I needed a lot of time to debug, and play against other student's AI's to tune those parameters to the best result.

5. Improving the AI Player Further

I was thinking on adding a move selection method: After the pruning, I would have a subset of moves with their heuristic value. Instead of being predictable and picking the "best" one all the time, I can pick one of the best moves with Gaussian distribution around the best value. This would add a little unpredictability in the AI that might break som predefined patterns that the opponent might expect.

In the timing side, I can use the given 30 seconds on the 1st move to build up a tree of all possible moves and their corresponding states. Then when it's time to run alpha-beta, I would not need to build the tree recursively but only any new states that are not yet in the current tree.

In Java, recursion is fairly expensive compared to iteration (in general) because it requires the allocation of a new stack frame. So going through the tree iteratively (since it is already built) could speed up the computation. This would allow to have a larger subset of “best moves”.

Lastly, I can mention supervised learning. I can run the algorithm for a long time storing the results in a text file and calculate the worth of each state. That way if that state appears again during a match, the AI would be able to quickly determine the next best move