Das Bohnenspiel is a 2x6 mancala game in which individuals are required to maximize the number of beans inside their pits. The outcome of each game can be a win, draw, or lose for either players depending on the moves of the opposing player. Mancala games are a prime example of perfect information sequential deterministic game, meaning players have complete information of their current state and possible action sets. This allows players to accurately predict and evaluate opposing moves efficiently, enabling accurate predictions of their utility or payoff for every turn. For this reason, in my project I chose a minimax algorithm with alpha-beta pruning and a greedy heuristic.

In my algorithm, I made the key assumption that each player will try their best and play their best move.  Optimally, it would have been best for me to skip until I made the second move. However, because of the rules of the game where consecutive skips are not allowed, statistically over a larger sample size, it would not change the outcome of a game between two truthful players. Another assumption I made was that most live players I will be competing against will be using a minimax algorithm with alpha-beta pruning and that our heuristics will be the deciding factor between winner and loser. I considered doing a monte-carlo search tree, however, decided against it because each move is timed. And to run a good enough sample size of each simulation my program will likely timeout or waste time making an informed decision each turn. In addition, I would have to implement a minimax algorithm anyways when choosing which node to simulate with a monte-carlo search tree.

For my minimax algorithm, I also implemented a depth limited search since the competition will be timed and it would not be very effective to go to a very deep node. My algorithm also incorporates a best value variable that essentially take the descendant leaf value from the current node and allocates it to the current node. My heuristic also takes the best utility for an individual player at the current state of the node and measures the chance of that node for winning. This heuristic is calculated by sum of the amount of seeds that each player has and the number of seeds inside their pits. This is a decent heuristic to use because the player with more seeds on their side of the board and in their pit, is likely to have more points at the end of the game. My code also implemented for each player an alpha-beta pruning code that looks for the highest utility value for each player. If the utility is lower for the adjacent node, that search will stop and assume the previous node's value. This greatly improves the efficiency of my code and allows me to go to a greater depth for each game.

The theoretical basis for my algorithm is that minimax provides a great way for players to optimize their outcome by simply applying the rules of game theory – in which every player will play their dominant strategy. This allowed me to write a simpler code since I did not need to evaluate every possible scenario, but rather apply only the optimal scenario and predict the best move the opposing player would make. Minimax does exactly this because every maximizing player on a certain level would choose only the node with the highest heuristic value. And when it's the minimizing player's turn they would only choose the heuristic value from the next level[1]. When searching back up the tree, the minimizing player usually has a better chance of winning. Because Das Bohnenspiel is such a large game with so many possible moves, implementing alpha-beta pruning greatly reduced the number of possible moves that needed to be evaluated. The use of alpha-beta pruning comes back to the idea that each player will try their best and play their best

[1] **"Minimax." Wikipedia. April 05, 2017. Accessed April 11, 2017. https://en.wikipedia.org/wiki/Minimax#Combinatorial_game_theory.**

moves. Alpha-beta pruning allowed me to play more games against a random player and allowed me to reach deeper states in each individual game while retaining the same outcome[2]. The theory behind my heuristic evaluation function is that each player is trying to win by having the most seeds in their pit. Therefore, if I took the current number of seeds on their side of the board and added it to the total number of seeds in their pit – it would provide an accurate account of who is currently winning.

The advantage of my approach is that my algorithm is simple and quick. It should give similar results to a more complicated monte-carlo search tree (MCST) if the MCST is also implementing minimax and assuming players to be truthful. Another advantage of my approach is that I used alpha-beta pruning which greatly reduced the processing time of each node's utility since pruning will result in the same outcome values for each parent-node. My code is also advantageous in which it uses a heuristic that evaluates the possible score for each player thereby allowing the maximizing player to maximize the opposing player's value, and minimizing player to minimize the opposing player's value. However, a player with a better evaluation or heuristic function will yield better results.

The problem with my code is that it heavily relies on an explicit evaluation function. Creating this evaluation function was difficult and consequently I chose a very simple one that just compared the scores between two players. A competition player that has a better evaluation function will surely beat my game every single time. A competition player who uses a MCST would have not needed an evaluation function and will likely beat my code every time. In addition,

[2] **"Alpha–beta pruning." Wikipedia. April 04, 2017. Accessed April 11, 2017. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning.**

an opposing player who makes random moves arbitrarily will result in the breakdown of my code since I assumed that every player will play their dominant strategy. Also, my algorithm will not be able to handle a very large number of games because its depth is limited. However, limiting the depth of my algorithm is a double-edged sword since it will also prevent my code from looking too far down a game's possibilities and reducing the hindrance of limited memory and processing power. This also eliminates the possibility of time-outs while selecting a move.

During my attempts at making a working algorithm, I initially attempted to create a MCST. However, I quickly realized that it was too complicated and that the limitation of the processing power, as well as the game's time constraints greatly reduced the efficacy of using a MCST. I was unable to have a working MCST.

I can improve my algorithm's code and the outcome of each game by creating a better evaluation and heuristic function. This would have allowed me to select a better move each turn. In addition, I could have created several different heuristic functions depending on the state of each game. For example, I could have had a heuristic function for when I was winning the game at its current state, a heuristic function for when I was losing the game at its current state, and another heuristic function for when I was at a draw at the game's current state. Running these heuristic functions simultaneously would have allowed me to have better moves each turn and granted me a higher chance of winning when playing against a MCST player or a player with better heuristic functions. Another way of improving my chances of winning would be to implement a function that would make random moves sometimes. This would throw off the opposing player if the opposing player was using a minimax algorithm as well.