5. Submission

- Briefly describe your design to solve problem 1-3.
  - Problem 1:
    - I simply followed the given pseudo code in lecture. The code is recursive and the main method move has two helper functions - max_val and min_val. Both look at the utility of the next possible steps from the current state.

      In max_val, which takes in a state, it checks the maximum utility of the next step from state. It is a recursive function. The base case of that recursive function is when the state that is passed in is the terminal state or there are no possible moves past this state for either of the players. Besides the base case, it looks at all of the possible actions and chooses the one with the most utility and recursively updates the cache with the action with that has the maximum utility between the minimum value of the next state and the stored min-max value. Whatever move has the maximum utility, it stores it in the cache.

      Min_val is similar to max val except for the fact that it checks for the minimum utility of the next step. With the base case being the same as max_val's. It looks at all of the possible actions and and it looks at all of the possible actions and chooses the one with the least utility value and recursively updates the cache with the action with that has the minimum utility between the maximum value of the next state and the stored max-min value. Whatever move has the minimum utility, it stores it in the cache.

      Both of the above recursive methods are brought together with the method move which takes in a state. It first calls the max_val on the current state and stores its value. It then looks at all of the possible actions from that state and and looks for the results of the state in the cache and returns the action.

- Problem 2:
  - This part of the assignment deals with the alpha beta min-max. The logic is similar to problem 1's logic, which deals with min-max only. The only differences are that the pruning happens before the helper methods max_val and min_val returns anything and the min_val and max_val methods take in two more parameters (alpha, beta) and return 2 more values alpha, beta. The main logical difference from problem 1 is how the tree is pruned.

    In max_val, the parent value is updated based off of the the beta value. If the beta value is greater than the maximum utility value of a specific action, then it returns the current maximum utility value along with the given alpha and beta value. Else, it finds the maximum alpha between the given alpha and the recently found maximum utility value of a specific action.

    In min_val, the parent value is updated based off of the the alpha value. If the beta value is less than the minimum utility value of a specific action, then it returns the current minimum utility value along with the given alpha and beta value. Else, it finds the minimum beta between the given beta and the recently found minimum utility value of a specific action.

- Problem 3:
  - The evaluation heuristics simply calculates the evaluation number based on player's points plus the stones on his side, subtracted by the other one's points and the stones on the other one's side. This number is divided by all the stones in

the game to create the possible percentage of how many stones a player could take or the other player could take.

The evaluation equation used a simple helper method in order to get the number of stones on either player's sides.

- Describe the approach in details you used in Problem 4 and other approaches you tried considered, if any. Which techniques were the most effective?
    - What we essentially did was combine the algorithms for problem 2 and problem 3 together to form problem 4's algorithm.

      The max_val and the min_val of p4_custom_player is like problem 2's alpha-beta function but with problem 3's evaluation put into consideration. The only time we use the eval function is when self.is_time_up() returns false or depth has reached zero. If it has reached this if statement and this is the end state then we set the cache of the current state equal to state.utility(self) or else we set the cache of the current state equal to problem 3's eval function. And then within the if-loop that checks if time is up or the depth has reached its limit, we return the cache value of the current state and the alpha and the beta. This goes for min_val as well - the majority of the function is the same as the min_val in problem 2 but with the above changes.

      In further detail, the method that implements this is move in p4_custom_player.py in the solutions file and it takes in a state. It first calculates the utility from minimax-alpha -beta and the the depth is unlimited. It goes through the max_val function that takes in state, alpha, beta and decreasing depth by 1. For all possible actions from that state, if the result of the action on the state is in the cache and the choice in the cache is equivalent to the current state and the minimum utility is less than the returning action of the max_val function given the current state, then that action is the best action for now and has the best utility. After going through all the possible actions, we return the best action.

- Evaluate qualitatively how your custom agent plays. Did you notice any situations where they make seemingly irrational decisions? What could you do/what did you do to improve the performance in these situations?
    - Qualitatively it works just as well as problem 2 and problem 3 and that is mainly because we made problem 4 a combination of problem 3 and 2. Because self.is_time_up is always set to false, p4_custom_player would always run in an infinite loop. So I created a depth variable that is useless in the algorithm but useful in preventing an infinite loop. The depth variable decrements within the while loop in move(self, state) function.
- **What is the maximum number of empty pits and stones per pit (i.e. M, N) on the board for which the minimax agent can play in a reasonable amount of time? What about the alpha-beta agent and your custom agent, in the same amount of time?**

The running of Minimax for M = 4 N =4 goes over timeout = 120. For other programs, Like Problem 4, it runs in less than a second to generate the next action. AlphaBeta also had some problem with time outs at about the same timeout as Minimax.

- **Create multiple copies of your custom agent with different depth limits. (You can do this by copying the p4_custom_player.py file to other *_player.py and changing the class names inside.) Make them play against each other in at least 10 games on a game. Report the number of wins, losses and ties in a table. Discuss your findings.**

Players	**Wins out of 10**	**M/N/Timeout = 1**
Minimax vs p4_AlvinPlayer	**Minimax = 4** **AlvinPlayer = 1**	**4/4** **Minimax timed out**
Minimax vs p4_AlvinPlayer	**Minimax = 3** **AlvinPlayer = 2**	**8/4** **Minimax timed out**
Minimax vs p4_500_Depth_Player	**Minimax = 3** **AlvinPlayer = 2**	**4/4** **Minimax timed out**
Minimax vs AlphaBetaPlayer	**Minimax = 1** **AlphaBetaPlayer = 4**	**4/4** **Both timed out**
AlphaBetaPlayer vs p4_100_Depth_Player	**AlphaBetaPlayer = 4** **100_Depth_Player = 1**	**4/4** **AlphaBetaPlayer**
AlphaBetaPlayer vs p4_AlvinPlayer	**AlphaBetaPlayer = 4** **AlvinPlayer = 1**	**4/4** **AlphaBetaPlayer**

It looked like Minimax timed out quite a bit but were always managing to create a board state that was better for itself in the future while the Problem 4 programs were generally looking into one state further and thus lost frequently. The time outs were mostly seen in the beginning 3-4 turns however, and when it hit a certain point, most of the programs would go through the rest of the states very quickly.

- A paragraph from each author stating what their contribution was and what they learned.
    - Alvin
        - Wrote the alpha-beta pruning algorithm. Learned that each node had to keep its own alpha/beta values and compare them to their children's utilities. Then adapted the alpha-beta algorithm by copying it over to minimax and removing the lines of code that had alpha and beta. Implemented the iterative deepening minimax with alpha-beta pruning using the heuristic from p3. Iterates through each layer of the minimax tree and keeps track of the best utility for max until the time is up. Also includes timeout checks in the base cases for min/max_val and in

the loop that iterates in each node and calculates the utility of each of its children. When the timeout occurs, it returns the best option it has so far according to the utility function or heuristic function, depending on whether it is looking at a terminal node.

- Woo
    - Worked on the evaluation function and a bit on the write up. Helped working on table creation and the evaluation of the results of programs going against each other.
- Sarah
    - I help plan how we were going to go about the assignment and created a whole to do list for it. Each person had his or her own role. I was originally assigned to complete the first problem. I found it hard to wrap my mind so eventually Alvin helped me on out with it. In order to be useful, I did the write up and descriptions of part 4 problem 1, part 4 problem 2, part 4 problem 3. For each of the problems I looked through each line of code, annotated it in the comments and turned those comments into paragraphs. After that I kept track of the logistics of the assignment and made sure they were being met.