__author__='jix029@ucsd.edu,A99076165,jih089@ucsd.edu,A99037641,z6hu@ucsd.edu,A99056145'

1 Briefly describe of your design to solve problems 1 - 3.

- problem 1: Minimax algorithm evaluates all possible actions and choose the one that can make the player to win. Considering all possible results: win, tie and lose, we need three numbers to indicate those results, thus we choose 1, 0 and -1. Consequently, the default value for the parent nodes can be any number larger than 1 when we search for the minimum child, and any number less than -1 when we search for the maximum child. We choose 2 and -2 respectively. Then we recursively computes the values for the interior nodes in the game tree according to the maximum rule until we reach the leaf nodes. Finally, we get the action that can give us the best result.

- problem 2: One problem of minimax algorithm is that it also traverses redundant nodes which cannot generate the maximum or minimum value we need. For example, if the parent node has already updated the maximum value to be 1, it is unnecessary to traverse other child nodes because 1 is the highest value we can get from the whole game tree. Alpha beta pruning algorithm solves this problem by offering an extra check. We basically reuse our code from p1 but separate the step of choosing minimum value and maximum value. In this way, we can easily check whether it is necessary to explore the child nodes and skip over all the redundant nodes.

- problem 3: This evaluation function selects the action that can produce more stones in the player's pits. It only loops through all possible resultant states and make mathematical calculations. To differentiate the player's row and the opponent's row, we parse the player's row to our evaluation function. After that, it will return the action that can produce the most stones on the player's side. If there are multiple actions that produce the same result, we will choose the one with smaller index number.

2 Describe the approach in details you used in Problem 4 and other approaches you tried considered, if any. Which techniques were the most effective?

In problem 4, we implement an iterative deepening minimax search with alpha-beta pruning. Since the main algorithm is based on minimax search with alpha-beta pruning, we start from what we've already finished in problem2. To implement an iterative deepening algorithm, we need to keep track of our current depth and the depth limit. Thus, we add one more argument (depth) when calling our alpha beta algorithm. After finishing the iterative deepening minimax search with alpha-beta pruning, we then work on the order of our DFS. The original order was based on the indexing order, we change it to the order we get from the evaluation function of problem 3. In this way, we more likely get the best action at the early search, which is beneficial if the time is limited. When time is up, we choose the best action we've already had from the last search.

Another approach we considered is to start from the iterative deepening depth first search and then add minimax search with alpha-beta pruning. However, we find it less effective because we almost need to rewrite the whole algorithm whereas the other one only needs to add a variable to keep track of the depth. It is because minimax search with alpha-beta pruning also implements the structure of depth-first search, which is easier to modify.

3 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?

Our custom agent will search the game tree using an iterative deepening minimax search with alpha-beta pruning. If time is not up, the agent will keep searching the tree until it finds the action that can lead winning. If time is up, the agent will select the action based on our heuristic function, which is the same as the one we implement in problem 3.

One problem of this heuristic function is that it is not always the best action to move based on the amount of comparably increased stones. For example, suppose the agent's row is [0 1 1] on the top whereas the opponent row is [2 0 0] on the bottom. Based on our heuristic function, we ought to move the stones at the second pit because no matter which pit we choose, the resultant evaluation values are the same, thus we ought to move at the smaller index, which is the second pit. However, the opponent can capture the agent's stones at the third pit. In order to avoid capturing, we have to move stones at the agent's third pit. We could fix such an error by checking Thus, it is unnecessary to move the stones at the first index. We could fix such an error by checking whether the opponent can reach the empty pit or not. Considering that we only have 1s to report the action, we choose to keep our heuristic function as simple as possible.

4 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?

When time limit is 5s: Minimax:(M=3,N=2),(M=2,N=7)
AlphaBeta:(M=3,N=5), (M=4,N=2),(M=2,N=30)
Custom:(M=2,N=10) (M=3,N=2)(M=4,N=1)
When time limit is 10s: Minimax: (M=2, N=8) (M=3,N=2)
AlphaBeta: (M=2,N=40) (M=3,N=8)(M=4,N=2)
Custom:(M=2,N=20)(M=3,N=2)(M=4,N=2)
When time limit is 20s: Minimax: (M=2, N=9) (M=3,N=2)
AlphaBeta: (M=2,N=50) (M=3,N=8)(M=4,N=2)
Custom:(M=2,N=35)(M=3,N=3)(M=4,N=2)
When time limit is 30s: Minimax: (M=2, N=10) (M=3,N=2)
AlphaBeta: (M=2,N=62) (M=3,N=8)(M=4,N=2)
Custom:(M=2,N=47)(M=3,N=3)(M=4,N=2)

For all of the algorithms, N tends to decreases exponentially as M increases.
When time limit is fixed, alphabeta performs the best, custom performs worse than alphabeta, and minimax performs the worst. It is because alphabeta skips over all the redundant nodes that the game tree may have, which saves a lot of time compared to minimax. Custom player

implements the iterative deepening algorithm, which takes extra time compared to alphabeta, but is still better than minimax.

5 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.

	CustomPlayer(0) (depth_limit)	CustomPlayer(1) (depth_limit)	Board size	Game result
1	5	10	M=2, N=2	draw
2	5	10	M=2, N=5	0 win
3	5	10	M=3, N=2	1 win
4	5	10	M=3, N=3	1 win
5	5	10	M=4, N=2	1 win
6	10	15	M=2, N=2	draw
7	10	15	M=2, N=5	0 win
8	10	15	M=3, N=2	draw
9	10	15	M=3, N=3	0 win
10	10	15	M=4, N=2	0 win
11	15	20	M=2,N=2	draw
12	15	20	M=2, N=5	0 win
13	15	20	M=3, N=2	draw
14	15	20	M=3, N=3	1 win
15	15	20	M=4, N=2	1 win

When board size is very small, our custom player can solve it under small depth limit, thus there is little difference between two players although they have different depth limit.
When board size is at the moderate level, the one with higher depth limit will win, because it tends to search to the leaf nodes of the game tree, whereas the other player can only react based on the heuristic function, which is less reliable.

When board size is very large, both custom players cannot finish scanning the whole game tree, thus they both react based on the heuristic function, which means the winning status is randomized.

6 A paragraph from each author stating what their contribution was and what they learned.

Jinye xu: I worked on pretty much every problem in this assignment, and learned how to code those algorithms.

Jiaying Hu: I worked on problem 2 and 4. I learned the algorithm of alphabeta pruning and transposition table and how to implement them in code and debug with pdb.

Zhongting Hu: I worked on problem 3 and reported all the approaches we use in this assignment. I learned how to improve minimax algorithm and apply it to a real game.