Artificial Intelligence: *Colosseum Survival!*

Jack Wei, Eamonn Lye

1. Introduction

Colosseum Survival! is a fully observable, two-player, turn-taking, zero-sum game. The adversarial and deterministic nature of the game naturally motivates search strategies such as Minimax search and Monte Carlo Tree Search (MCTS), which we explored throughout the project. By comparing different approaches, including a 1-step look-ahead greedy algorithm, MCTS and various flavours of minimax search. Ultimately, we have determined that the superior algorithm is the Iterative-Deepening Heuristic Alpha-Beta Tree Search (IDH-AlphaBeta). For performance optimization, we also implemented a history table for dynamic move ordering and to prevent repeated evaluations of previously seen game states.

2. Explanation of IDH-AlphaBeta

IDH-AlphaBeta is implemented by repeatedly invoking a depth-limited Minimax search algorithm with alpha-beta pruning. With the input game state as the root, the algorithm searches a partial game tree with a cut-off depth given as an input. We start the search with a cut-off depth of 1 and iteratively increase the search depth. An optimal solution/move is obtained for each iteration. With the idea that optimal moves from deeper searches are better than those found from shallower searches, our agent updates the current best move with the move found from the most recent iteration. As the agent used up the time allocated for deliberating a move (1.92 seconds), the current best move would be returned and executed for our ply.

Utilities of nodes within the constructed search tree are ultimately determined from terminal nodes, states that result in the game's termination, and nodes at the cut-off depth. Minimax search and, hence, alpha-beta tree search is depth-first, with each recursive call searching nodes one level deeper from the root. Therefore, we passed the current search depth from the root node (input game state) as an input in the recursive functions of the alpha-beta tree search. We then implemented a terminal test function that returns a utility if the input node is either a terminal node or a cutoff node.

We designed a heuristic function for the evaluation of cut-off nodes. We decided to use a function of our agent's number of reachable tiles minus that of the opponents.

For terminal nodes, the utility values are as follows: if the node is a win or a loss as determined by our area score (the number of blocks in our agent's enclosed area minus the opponent's area), multiply the area score by 1000. If the game is a draw, the utility is again the heuristic function. The evaluation function is as follows.

$$\text{Eval}(s) = \begin{cases} 1000 \text{ area\_score}(s), & \text{if } s \text{ is a win or lose} \\ f(\text{max\_player}) - f(\text{min\_player}), & \text{if } s \text{ is a draw or a cut-off node} \end{cases} \tag{1}$$

where s is a game state, and f is the function that returns the number of reachable tiles

The min and max function both receive a game state as input. A game state is represented by three variables, chess_board, a NumPy matrix, our_pos, the position of our agent and adv_pos, the position of the adversary. In each of these functions, we obtain the legal moves at the input game state by the function get_all_valid_steps and generate new game states by applying the moves to the input game state (the new board is a deep_copy of the board of the input). The generated game states are then passed to max functions in min and min functions in max.

Furthermore, we use Alpha-beta pruning combined with a history table to optimize the algorithm. The Alpha-beta pruning technique saves searches on the next level of the game tree when it is guaranteed that further searches and evaluations would cause no change in the final solution. As alpha-beta pruning can save searches in terminal nodes or subtrees, it significantly improves computation time. It, therefore, allows the iterative deepening minimax algorithm to reach a higher depth within the allocated time, leading to better moves. Inspired by section 6.2.4 of Norvik & Russell's AI textbook, we can maximize the potential of alpha-beta pruning by ordering the legal moves that generate the children nodes of the game state in consideration. We can achieve this by caching the evaluation of previously evaluated nodes in a history table. Iterative deepening search guarantees that evaluated nodes from the previous iteration are searched again in the following iteration. We can use the evaluation value of these nodes to reorder visits/legal moves in the Max and Min functions. When the best moves (higher evaluation) for the max player's ply are considered first, it is more likely that alpha pruning will occur more frequently in calls to min. Similarly, when the best moves (lower evaluation value) for the min player's ply are considered first, beta pruning may occur more frequently in calls to max. We order the moves accordingly based on the evaluation of the game state they generate. As the iterative deepening search terminates due to the time limit, the history table is cleared since the same game states can't occur again for future turns [1].

3. Motivation

3.1 CutOff

The idea for a heuristic cut-off of the alpha-beta tree search stems from the fact that the game tree for Colosseum Survival is theoretically massive. We let $M \times M$ be the board size and $K = \lfloor \frac{M+1}{2} \rfloor$ be the maximum number of steps according to the specification. Then the branching factor of the game b is $O(K^2)$. We let m be the maximum number of plys in a game, which is the maximum depth of the game tree, then the game complexity (size of the game tree) is $O(b^m)$. Hence, a full minimax search on the game tree is computationally intractable for a decently large M, especially under the 2-second search time limit. We address this by cutting off the game tree at a certain depth and evaluating the game state of the cut-off nodes with a heuristic evaluation function.

3.2 Iterative Deepening

Under the 2s time limit, the reachable depth of the alpha-beta search algorithm is highly variable for various game stages and boards of game stages. For example, at the start of

1. A placed barrier cannot be removed

the game, the alpha-beta search algorithm can reach depth 3 for 5×5 boards but only depth 1 for 10×10 boards. In the late game stages, where there is a significant reduction in the valid moves, the search can explore the remaining game tree for 5×5 boards and reach depth 3 for 10×10 boards. Therefore a search algorithm with a fixed cutoff depth is undesirable due to high variability in the maximum searchable depth. Motivated by this problem, we decided to discover the optimal search depth by computing multiple depths with an iterative deepening search. Another major benefit of this strategy is that solutions produced during each iteration are candidates for the final move. Our implementation of returning the most recently obtained move allows us to use up all the time resources given to use in our ply.

3.3 Heuristic Function

Based on our analysis of the game, a good design of the heuristic function should be 'similar' to the end-game score. Specifically, we want the heuristic function to order the terminal game states close to the same order as ordering using the actual score. With the well-designed heuristic, we can safely treat non-terminal games as if they are terminal games without misleading the agent into choosing bad moves. Based on this criterion, we immediately thought about the heuristic function in the second part of function 1. In the end game stage, when the blocks are numerous, the difference in the reachable blocks between our agent and the opponent reflects the final area score.

Alternatively, we can think about the heuristic function more intuitively. We want to maximize the number of reachable blocks as this value is a good indicator of the number of blocks in the agent's enclosed zone when the blocks are numerous. Moreover, more reachable blocks mean more legal moves to pick from, which is highly advantageous. Conversely, we want to minimize the opponent's reachable blocks to potentially reduce its area score and reduce the number of its legal move.

We now explain the motivation for multiplying the area score by 1000 for terminal nodes. This is because we want to force the max player to pick winning moves and the min player to pick losing moves without possible interference from the nodes evaluated from the heuristic function. Multiplying by 1000 essentially pushes the terminal nodes out of the range of the heuristic function and orders terminal nodes to the extreme ends.

4. Theoretical Basis

The alpha-beta tree search is an optimized version of the Minimax search. The minimax algorithm is a depth-first search that generates and searches the entire game tree using alternating recursive calls of the functions max and min, essentially simulating alternating plys of the max player and the min player. Our agent is the max player and the opponent is the min player.

The utility/evaluation values are all determined from the perspective of the max player. The max function returns the maximum utility/evaluation returned from its children which are either the min nodes or terminal nodes. In contrast, the min function returns the minimum utility/evaluation returned from its max nodes or terminal nodes. In regular minimax search, the utility values are initially determined at the terminal nodes. The internal nodes are updated as the recursion unrolls from the terminal nodes. In the depth-limited version of our minimax algorithm, nodes at the cut-off depth are treated as terminal nodes with their utilities determined from an evaluation function.

The optimal move for the max player can then be determined by calling the max function with the input game state and finding the corresponding move that generated the maximum utility. Under this scheme, the minimax algorithm guarantees a minimal loss of utility when the opponent plays optimally.

Lastly, as mentioned in the explanation, the alpha-beta pruning combined with the move-ordering technique can massively reduce calls to deeper levels of the game tree. Under perfect move ordering, alpha-beta only needs to search $O(b^m/2)$ nodes [2]. The dynamic move ordering technique implemented brings the algorithm closer to the theoretical limit.

5. Advantage and Disadvantage

One of the benefits of the IDH-AlphaBeta Tree Search algorithm is that it is highly robust to various game states and the opponent's strategy. This is due to the minimax algorithm guaranteeing a minimal loss in the worst-case scenario when the opponent maximizes their gain. Therefore, our agent's play can be characterized as extremely careful. Suppose our agent considers two depths, which corresponds to a single turn. In that case, the agent will never make a game-losing move unless, for every legal move, the opponent can make a decisive counter move that leads to a worse scenario for our player. Hence, against strong players, especially players that use similar heuristic functions for their evaluation, our agent play is near-optimal given the same search depth on both sides. Against weak players, our agent can win consistently, given that the minimal loss scenario corresponds to a win. This advantage is evident as our agent has a 100% win rate against a random agent and MCTS agent, which we will discuss in the following section.

On the other hand, this careful playstyle leads to the minor disadvantage of not exploiting weak enemies and not adjusting strategies to opponents' playstyle. For instance, an agent under a different playstyle can increase the final score when playing against a weak opponent. However, since the Colosseum Survival is a game where "the winner takes it all", this is not a problem.

2. Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010.

A critical disadvantage of any minimax agent is the high runtime due to attempting to explore the large game tree. The time complexity is $O(b^m)$ where m is the depth of the search. This downside is especially dangerous in games of large boards, i.e. 10*10, where the agent can only search for one ply. However, we have offset this disadvantage with the heuristic function, as it generally allows survival until the late game stages, in which the search can reach much deeper into the game tree. Moreover, alpha-beta pruning combined with move ordering significantly improves performance over ordinary minimax. For significantly larger boards such as 20×20, our agent fails to evaluate a single ply and therefore is equivalent to a random walk agent during the start game. We observe that our agent have trouble winning against our MCTS agent on boards larger than 16×16.

Another disadvantage of using a history table to cache previously evaluated moves is the space complexity of $O(b^m)$, as all nodes in the tree will be saved in the process of iterative deepening. However, since the move table is cleared after each move is taken and the limit of 2-second thinking time would prevent memory usage to exceed 500 Mb even in extremely large boards such as size 30×30.

6. Other Approaches

6.1 Look-Ahead Greedy

We started with a simple implementation of a look-ahead greedy algorithm that sorted our valid moves from best to worst. This agent returned any winning moves immediately upon finding one, and when it didn't, it sorted our valid moves into three categories: tie, no-end, and losing. For most of our algorithm, it maintains only a 1 or 2-step look-ahead, but it goes as far as 3-step look-ahead when it has to decide whether our no-end moves will result in a potential loss and that tying would be better. No-end moves (neither winning, tying, nor losing) are split up into two groups:

- (Priority) Moves that prevent an adversary from winning two moves later by forcing our agent to kill ourselves or maintain a vulnerable position
- Moves that prevented an adversary from winning in 1 move.

Along with a frequent time check to enforce our time limit, we also update our panic move with our safest moves when sorting our non-ending moves.

6.1.1 Motivation & Results

Our motivation for this approach was to start us off with an agent that could consistently beat a random agent and to pave the way for our understanding and implementation of a better approach.

It's a functionally similar version to a shallow minimax algorithm, where moves are ordered as follows: Victories, Safe Non-Ending Moves, Tie Moves, Unsafe Non-Ending Moves, and Losing Moves. Even though the look-ahead contingencies we implemented in our algorithm did not play as significant of a role against a random agent as it could have against a more rational agent, it still prevented the chance of the random agent picking a winning move in the next 2 steps. In the end, our greedy algorithm agent scored around a

99.7% win rate against a random_agent in 1000 runs which was a great start to the rest of our development. Against IDH-AlphaBeta, the win rate is 0%.

Table 1: Summary of advantages and disadvantages of Greedy Look-Ahead

Advantages	Disadvantages
• Prevents taking a losing or dangerous step, if other moves exist. • Guarantees taking a winning step, if there's one that exists. • Panic move guarantees a legal move is returned every time. • Never goes over the time limit (avg 0.25 seconds).	• Mostly defensive, unless there's a winning move. • Too simplistic; only ever looks ahead by 3 steps maximum.s • Easy to understand and beat after playing against a few rounds.

6.2 Simplified MCTS

After seeing our alpha-beta agent consistently beat our first approach (Greedy), we decided to see if we could take another completely different approach to challenge our new contender. Our Monte-Carlo approach was significantly less expensive but equally less thorough than the original Monte-Carlo tree search. It ran a single random simulation for every legal move, giving a 100 on a win, -25 on a tie, and -10000 on a loss, which guaranteed that moves that simulated in a loss would be very rarely chosen. We maintained frequent time checks between our simulations, and when we hit our own 1.97-second time limit, we fell back to our first greedy approach calculation.

6.2.1 Motivation & Results

Our motivation for a simplified MCTS was to use a completely different approach to challenge our leading agent, without building on the concepts and mechanics our previous agents already had. Without the use of an evaluation function and direct assumptions on an adversary's rationality, the choices our simplified MCTS agent made were from a completely different angle. This made it extremely riveting to see that it was able to take on a 44% win rate against the first iteration of our best agent, where our greedy agent was losing 100% of the time. The Monte-Carlo agent became a good judgement on the overall quality of our leading agent and was a good measure on the upgrades and changes to our alpha beta agent by seeing the results as it's pitted up against this MCTS agent. After modifying our best agent at that time, alpha beta agent, to an iterative deepening version, our MCTS agent saw a decrease to a 20% win rate in 500 runs. After optimizing our best agent with move ordering and modification to the evaluation function, we further saw a decrease to 0.02% win rate in 100 runs. Interestingly, even though our MCTS agent saw a huge improvement compared to our greedy agent when facing up against the 1st iteration of our

best agent, it performed mildly worse against a random agent, seeing a win rate drop from 99.7% (Greedy) to 97.8% (MCTS).

Table 2: Summary of advantages and disadvantages of Simplified MCTS

Advantages	Disadvantages
• Significantly less expensive than standard MCTS. • Does not need an evaluation function. • Does not have to make direct assumptions on the adversary's rationality.	• Significantly less thorough than standard MCTS. • Only effective when our list of valid moves is small relative to our board size. • Cannot be 100% certain on the quality of any move.

7. Future Improvements

We didn't take advantage of the 30 seconds for preprocessing data and changing the internal representation of the game. One improvement we can make is to convert the representation of the board from a NumPy array to a simpler bitboard format in which each block of the board is represented by one bit. We can use four bitboards to represent the direction of the barrier. If a block on the 'up' bit board is true, then that means the block has a barrier placed in the up direction. This enhancement will allow us to hash the game state more easily for caching purposes and for the reordering of moves.

Although our heuristic function is well-designed, we have not thoroughly tested other alternatives for the heuristic. For example, various other functions, such as the manhattan distance between our agent and the opponent, and adjacent barriers of our agent and the opponent, can be added to the heuristic function to form a weighted linear combination. The weights can then be determined through trial and error base on win rates when pitted against each other. Alternatively, we can train a neural network to learn the optimal evaluation of a non-terminal game state.