COMP 424 Final Project Game: *Colosseum Survival!*

AI Gameplay Agent

Justin Novick - 260958505
Jacob Greenwald - 260974924

A report describing the architecture of
a winning agent

COMP424: Artifical Intelligence
McGill University
Montreal, Canada
December 1, 2023

1. Introduction & Motivation

Upon first glance at the Colosseum Survival game, it becomes evident that there is not one uniquely dominant strategy to ensure victory, given the time constraint that a program can take to decide to move. This is so because much like chess, there is a large branching factor; particularly, this game's branching factor is upper bounded by the potential wall placements multiplied by the potential moves that an opponent can move to. It is known that if an agent is fixed at a position, it can choose 1 wall to place from a maximum of 4 possibilities. The domain of possibilities for a position will decrease if the fixed position is attached to already constructed walls or the natural boundary of the board. Before picking a wall, an agent can choose to move from its original position. The Manhattan distance that an agent can move on an MxM board is determined by the "maximum steps" variable $\lfloor \frac{M+1}{2} \rfloor$, and furthermore, the length and width of the board (M). So, if we are considering the upper bound of the branching factor as being 4 times the maximum steps, we can clearly see that the exponential runtime of calculating every possible move will be increasingly difficult on larger boards. There is not a unique and perfectly efficient way to win this game, however, agents can be designed to adopt a general strategy and follow the utility brought about by heuristics and current game states to make moves that are close to optimal.

The main goal of the agent submitted was to utilize the minimax strategy to maximize utility relative to a set of functions; at each point in the game, this strategy recursively calculates the opimtal move to make, under the assumption that the adversary would subsequently be choosing the move that would minimize your utility. This implementation was constructed without the necessity of recursing to terminal game states, as doing so would be too computationally complex. Empirically, it was found that this approach, coupled with a meaningful utility function, preprocessing, and pruning was better than other implementations we made, including Montecarlo tree searching algorithms, and traditional minimaxing strategies.

2. Design

Overall the minimax strategy is quite robust, however, it does suffer from two major pitfalls. The first is the time complexity, as mentioned earlier. The second is that its performance relies on having a great utility function to accurately portray the quality of mvoves. This project's agent's overall architecture works to augment minimax so that it mitigates these two faults. Additionally, further considerations were made based purely on the way the game should be played from an external point of view.

In order to stop the exponential algorithm from running indefinitely, the agent's minimax function takes in a parameter "depth" which decreases by 1 with each recursive call. The recursion is stopped by 1 of 3 base cases. The first, obvious, base case is the depth hitting zero. The second base case evaluates if a timer that is initialized at the beginning of the turn approaches the 2-second mark. The 3rd base case is a winning move is found and the game would end. In the event that a base case is hit, a call to evaluate utility is made. Ideally,

the program would like to recurse through more depth, however, this becomes increasingly difficult with a larger board. The time-limited base case allows it to recurse as much as possible while respecting the turn's time limit. Allowing the algorithm to investigate moves relative to a time frame also opens up the possibility that it will perform even better on virtual machines with larger computer power. Another architecture decision to respect run time was the implementation of a depth-limited breadth first search for generating successors function, which was used in both minimax and the preprocessing stage. This searching, which keeps track of already visited positions in its call, has a branching facor of 3. This is so because every position has a maximum of 4 possible places that it can move to with one unit (assuming no walls interfere), and one of those places is the square it came from. With such a branching factor, the time complexity of this operation becomes $O(3^{max\ steps})$, which is computationally efficient for the sake of the game. It is only necessary to search up to a manhattan distance equal to the max steps parameter because other possible successors are not valid moves. Another way this project was able to cut down on runtime was with the use of alpha-beta pruning in the minimax function; this augmentation gave minimax 2 more variable: alpha initialized to negative infinity, and beta initialized to infinity. Alpha is the value representing the minimum score that the maximizing player is assured of, while beta represents the maximum score that the minimizing player is assured of. By keeping track of these variables, the algorithm can find instances where alpha is greater or equal to beta, and thus proceed to prune the branch; this approach significantly reduces the number of recursive calls that minimax has to consider, ultimately allowing the algorithm to focus on more fruitful branches with the time it's allotted.

Again, the validity of minimax is only as good as the utility that it calculates when encountering the base case. The first term of the utility function used in this implementation evaluates to 1.2 times the remaining number of moves that the agent has left, minus the number of moves the opposing agent has for that given move. The utility attached to the agent's remaining moves is scaled by 1.2 to make it slightly more conservative, as this seemed to help based on empirical evidence. The second and third terms of the utility function subtract 3^x and add 3^y, respectively, such that x is the number of walls surrounding the agent and y is the number of walls surrounding the adversary. This part of the utility function emphasizes avoiding sticky situations and also brings in an additional measure as to how trapped the adversary and agent are. The utility function also assigns an extremely large positive value (+10000 utils) to game states with positions that are in striking range of a victory, while assigning an extremely large negative value (-10000 utils) to game states in range of a terminally losing play. The values for this utility function were found to be the best through testing the agent manually. Ultimately, this utility function does a good job of rewarding plays that humans generally think are indicative of winning, while penalizing those indicative of losing.

One of the largest decisions with respect to how the minimax function should operate came down to how the successive states should be generated and passed as input to recursive calls. Minimax attempts to recursively find the best utility-yielding move by considering all positions, but the definition of the term "position" has an affect on this; positions can refer to just the x and y coordinates of the agents, or alternatively, it could refer to the entire game state, including an actual placement of the wall. In theory, computing the latter exponentially increases the runtime it would take to find all possible positions, as

it increases the branching factor by up to 4. Undoubtedly, this has a major effect on how much depth the minimax algorithm can achieve in its allotted time. If minimax were to only consider x and y coordinates as its positions, it could achieve much more depth, however, the wall placement passed into the next recursive call would have to assume some sort of strategy, similar to the way MCTS algorithms run simulations on promising nodes with a rule. The debate of more minimax depth with a sacrifice of the tree structure versus less minimax depth with a truthful tree structure was put to the test with experiments. One agent was designed to simulate a random wall inside the body of minimax, while the other had no simulation as each position was already a combination of the coordinates and wall placement as one. We also sought to test whether or not the type of simulation executed for the wall placement in minimax affected the outcome. To test this, a new agent was constructed with almost entirely the same architecture, except the simulated walls in minimax were designed to aim in the approximate direction of the opponent; if a wall were to already be in that position, the placed wall would go to the first available position.

The final major architectural change was the implementation of preprocessing. Traditionally the successful positions that minimax is recursed with don't conform to much of a pattern other than the order they are traversed in. In this project's implementation, it was found to be extremely useful to sort the order of nodes returned by the successor generating function, by their expected utility. This way the recursive calls for minimax would be called on the more 'interesting' nodes first, leading to decisions that are much more meaningful. Although doing so increased the time complexity of the successive state generating function from $O(n)$ to $O(n^2 + n * log(n))$, [$O(n^2)$ with sorting done in place], and effectively diminished the amount of depth the minimax tree could reach, the nodes that were reached were much more telling. This approach with less depth and more preprocessing before the recursive calls with the successive states outperformed the implementation with more depth and no preprocessing, approximately 80% of the time. Another way this implementation worked to preprocess was by expanding out what would have been the first maximizing call to minimax, as its own code block in the step function. It was constructed in a way that wouldn't change how the minimax behaves as it loops over the successive states and calls them as minimizing nodes. Disjointing this part from the recursion the implementation of terminal state checkers, on just the immediately possible positions the agent could move to to end the game. This step makes sure to avoid terminally losing branches, and immediately move to terminally wining branches, without having to worry about calculating the utilities, potentially many times, in the minimax function.

3. Quantitative Performance

3.1 Depth

The minimax function is the main source of looking ahead. As discussed in the design section, it was found that sacrificing some depth in exchange for presorting recursive calls was advantageous. However, this did have a limit; for instance, the agent with a maximum depth of 1 and 2 (with presorting) seemed to underperform the agent with a maximum depth of 5 (without presorting). This difference was more profound on smaller boards, due to the fact that the agents with more allowed depth would naturally be stopped from recursing, because of the time limit. However, upon allowing depth to be even larger than

Board Size	Depth
3x3	3
4x4	3
5x5	3
6x6	3
7x7	3
8x8	3
9x9	3
10x10	2
11x11	2
12x12	2

Table 1: Depth for Boards

2 (i.e. 3), for small to medium size boards, the main presorting agent performed better than the agent minimaxing to a depth of 5 and 6; this was noticed because the presorting allowed the minimax of the agent to explore paths resulting in higher variances in utility. Overall, the presorting reduced the number of moves it could look ahead to, but, allowed it to achieve more depth on moves that would be critical to the game. The table of minimax depths achieved with the agent's presorting technique is listed in the table below. While these values are calculated on based on the first few moves, it is worthy acknowledging that far more depth is considered towards the end of the game, when the average branching factor diminishes. Note that the table below only starts 3x3 board, because of the natural limit of possible moves an agent can look ahead on, with a smaller board.

3.2 Breadth

The breadth achieved in the minimax is relative to the max steps variable, which in turn is relative to the board size. This is of course also dependent on the board state as the walls determine how many successors can be found in the depth-limited breadth-first search for successive positions. For the sake of calculating an upper bound, assume the agent is in the middle of the board with no walls surrounding it. The number of moves the agent can possibly move to is calculated as the number of moves less than or equal to a Manhattan distance of max steps away; this of course is upper bounded by a square perimeter around the current position, with a length and width equal to max steps. Suppose for the sake of the argument that this number is n: an upper bound to the maximum breadth that can be analyzed if the algorithm had no time limit. For board sizes ranging from 3x3 to 5x5, the breadth that could be analyzed ranged from almost all of n to 2n/3. For board sizes between 7x7 and 10x10, the breadth analyzed ranged between 2n/5 and n/3. For board sizes of 11x11 and 12x12, the breadth analyzed was slightly less than n/3. Although the small fraction of the breadth could seem alarming on larger board sizes, this was actually done by design. The presorting of the utilities makes sure that only the most important moves based on the magnitude of their utility will be analyzed further. Note that the fractions provided above are also a pessimistic outlook, given the fact n is an upper bound for possible moves, and there are likely walls that would prevent n from being as large as it

is assumed to be. The previously mentioned fractions also refer to the breadth analyzed on the first few sets of moves; as the board becomes more congested by walls, it can analyze all of the breadth of n. The degree of breadth that can be analyzed is approximately equal for the minimizing and maximizing calls.

3.3 Board Scaling

As mentioned in the design section, and elaborated on in the depth and breadth subsections, the call for generating successors increased from $O(n)$ to $O(n^2)$ once the preprocessing of the utility was implemented. This is so because now, for each of the possible successive positions, the algorithm considers each of said position's possible successive positions. Traditionally the time complexity of minimax is $O(b^d)$[1] such that b is the branching factor and d is the depth. However, given that the branching factor essentially doubles with the preprocessing, the overall time complexity becomes $O((n^2)^d) = O(n^{2d})$. Since n is a function of max steps, which is a function of the board size, it is evident that with larger boards, there will be a larger base, and time complexity will increase much more with the exponent. But, as discussed in the breadth and depth sections, the strategy doesn't plan to check each branch of the minimax tree. As seen in the above table, depth and breadth can be as little as 2 and 1n/3 respectively, for a board size as big as 12x12. All depth and breadth combinations respect the 2 second time, and recurse on the more interesting moves.

3.4 Impact of Heuristics

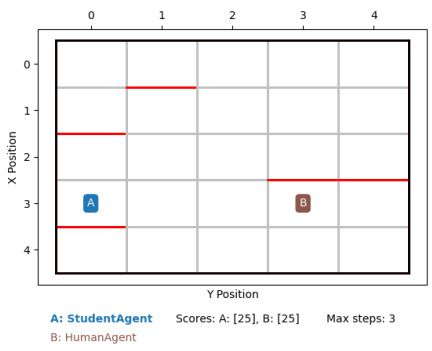

Fig 1: B's Turn

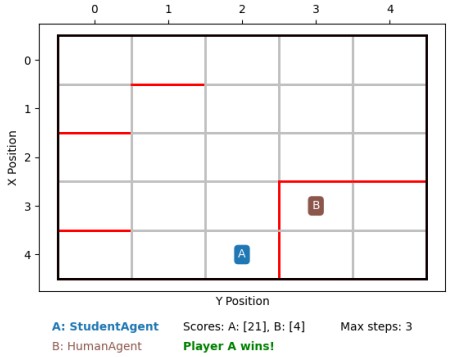

Fig 2: B to 3,3,l and A to 4,2,r

We chose to start our implementation of our heuristic by checking if the opponent has less than 4 moves available and if so we return a very large number to indicate that we should play aggressively and that is a likely winning move. In these figures that concept is displayed perfectly since in Figure 3 the human agent moves to 3,3 with a wall facing left. Our heuristic function than realizes that we have a way to trap the opponent so they only have 4 moves left by going to 4,2 with a wall at right and that would be the position with the highest heuristic value. When we begin minimax with our presorted moves by our heuristic we see that the move 4,2 will be the first suggestion and then we check if we can

1. https://cis.temple.edu/ vasilis/Courses/CIS603/Lectures/l7

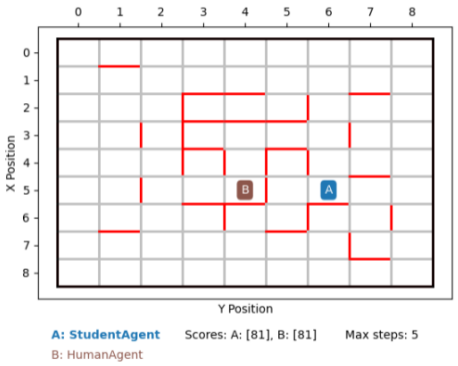

Fig 3: B's Turn

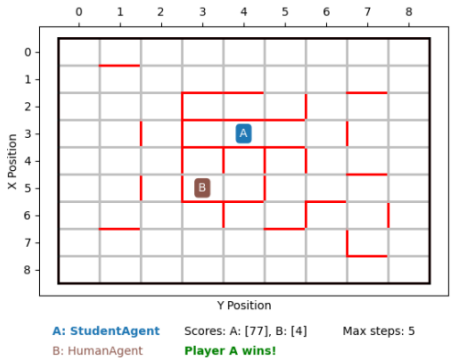

Fig 4: B to 5,3,l and A to 3,4,d

end the game in one move and we can so we return the move 4,2 with a wall at right and the game ends without even running minimax, and allows us to win right away. This same effect can be shown in figure 5 and 6. Otherwise our heuristic takes actions that are not too aggressive and plays to not get trapped and take some chances to attack the opponent at an opurtune time.

3.5 Tournament Predictions and Win Rates

We predict our agent to win 99.5% of its games against the random agent since that has been our result when running it to test our agent. We predict to win 25% of the games against Dave since humans are not limited by time complexity so they could see a fuller view of potential moves and their consequences. However, on larger boards, such as 11x11 and 12x12, we could see this win rate jumping to 50% as human's spatial perception diminishes at such sizes. We expect to win 75% of our games against our classmates' agents, and think our alpha-beta pruning minimax with a great heuristic will help us succeed.

4. Advantages and Disadvantages

The advantages of our approach are multi fold. We implemnted a is terminal function before running minimax, so we automatically return winning moves and skip losing moves right away which help give us a more robust agent. Another advantage to our approach is that our heursitic does a good job of looking at both our position and the adversary position and maximizes us having a lot of moves and not being surrounded by walls and the opponent having very few moves and being heavily surrounded. Finally we pre-process our moves, allowing us to explore the most promising nodes first and being able to look specifically for the best possible wall placement given our initial promising nodes, so we explore in depth what are the most promising nodes and every possible wall placement. The main disadvantage would be that our utility could use some fine-tuning on how to best show that the difference between having one move left and having 10 moves left is much worse than the difference between having 30 moves left and 40 moves left.

5. Other Approaches

As we mentioned in the design function we tested our agent against a minimax by just placing a random wall at each point and never looking for the best wall at each possible move, and while this agent was able to get more breadth, but because the moves were presorted based on the utility, finding the best wall for every move was a greater help than getting more depth for each move. We also tried our agent against an agent that does not use minimax and only uses a heuristic and again found it to not help create the best agent possible. We also tested our agent against an agent who ran minimax without pre-processing the next possible moves in any way and found that pre-processing by utility had a large effect in creating the best agent because we searched the most likely moves right away in the best order possible.

6. Future Improvements

In the future we would create an agent using MTCS and be able to simulate which method yielded better result. Another improvement we would look to make is placing non random walls in the code of the minimax. Right now we place random walls after our first placement of the wall where we traverse over every possible wall. We did this to save on time complexity, as running minimax for each possible wall would take up to 4 times as long. We can also use more efficient sorting by using a heap to allow for better time complexity. Finally, we can try using different utility functions and testing each of them for extensive periods of time and figuring out which one is best. Minimax is also not necessarily optimal if played against a non rational opponent.[2] So in the future, we would look into trying to have our tailor our minimax to each individual opponent and find the best way to exploit their suboptimal strategy.

7. Bibliography

References

[1] `https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7`

[2] Katarzyna Kosciuk, *Is Minimax Really An Optimal Strategy In Games?*, Faculty of Computer Science, Bialystok University of Technology, Białystok, Poland.

2. Is Minimax Really An Optimal Strategy In Games? By Katarzyna Kosciuk, Faculty of Computer Science, Bialystok University of Technology, Białystok, Poland