COMP 424 Final Project Game: Colosseum Survival! Report

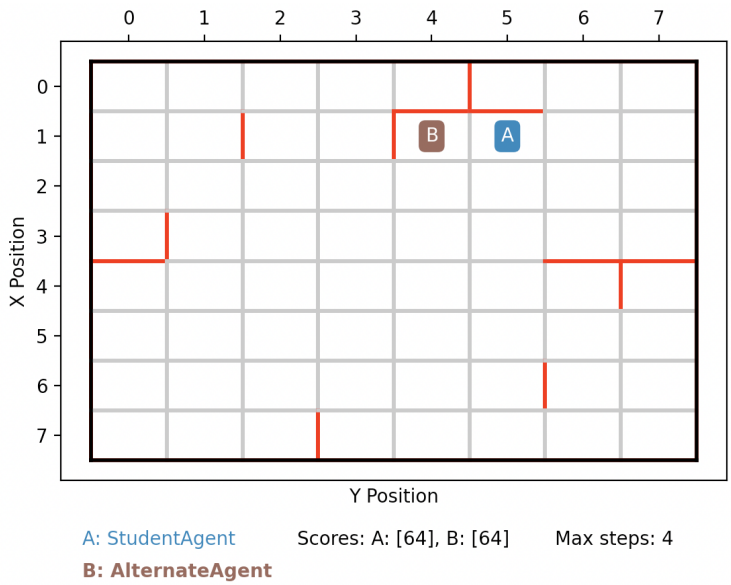
Michel Carroll: 260584901 Ayomide Ojo: 260967249

April 8th, 2022

Contents

1 Abstract

The goal of this paper is to explain the techniques and methodologies used when creating an AI agent for our final project for Computer Science 424, Artificial Intelligence, with rules and descriptions of the project defined in https://www.overleaf.com/read/gcpfjdpqpytp. Using a modified version of Minimax and various heuristic methods, we nearly achieved a 100% win rate against the random opponent.



Colosseum Survival game display

2 Introduction

With the exception of the initial configuration, Colosseum Survival is a deterministic game where every move has one possible outcome. It's also one where the adversaries have complete information. These two facts reduce the decision-making algorithm to searching the space of possible actions for the optimal move. However, both agents are bound to a strict time and a memory constraint which limits their ability to search the entire space – this fact heavily informs the choice of appropriate AI algorithm to solve this problem.

Assuming the worst-case scenario of a 12x12 game board ($M = 12$) and a maximum number of allowed moves of $K = 6$, we can estimate the upper-bound of the size of the search tree as follows: For simplicity, let's approximate the range of a player to a circle around him so it's able to reach $\pi 6^2 \approx 113$ tiles, and let's assume the player is in the middle of the board in which no wall has been put down yet. Since the player has 4 possible directions to place a wall after moving, this would mean a branching factor of $4 * 113 = 452$. The depth of the search tree can be approximated to $12^2 = 144$ – in other words, the game goes on until every possible square has walls. Thus, the size of the search tree is approximately 452^{144}. This is an immense game tree that any modern hardware would not be able to traverse effectively within a reasonable amount of time.

With this in mind, and the fact that Colosseum Survival is a zero-sum game, we decided to use a heavily modified Minimax algorithm as a basis for our AI, along with many handcrafted heuristics and optimizations to force it to stay within the problem's time and memory constraints. Using our knowledge of

the game, we derived the following intuition as a basis for the heuristics: it's generally advantageous for the player to move towards the enemy rather than away from it, and the most likely optimal move is along the shortest path towards the enemy.

3 Methods

3.1 Modified Minimax with Alpha-Beta pruning

We decided to use the Minimax algorithm, which is designed to find the optimal move to play at any given state in the game. This is usually done by expanding the search tree until all the terminal states have been reached. Due to the time constraints, the remaining time is checked to ensure the algorithm doesn't go over its 2 seconds of allocated turn time. If the time approaches the limit, the algorithm breaks early and returns the best move found so far.

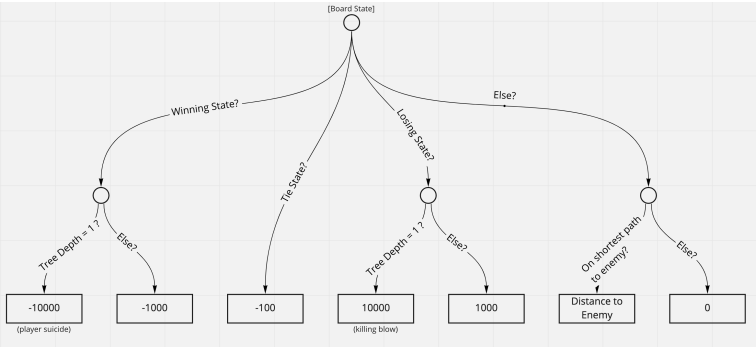
In additional, we made a modification to Minimax in regard to both its maximum depth and first-level branch ordering. At the start of every turn, the A* algorithm (see below) is applied between the player and the enemy to find the shortest path between them. Then, a procedure is used to order them for execution priority and maximum depth of traversal by using a domain-specific heuristic function: First, legal moves on the path to the enemy are processed in order from closest to the enemy to farthest, and with a maximum depth of 2. Second, legal moves not on the path to the enemy are processed in order from closest to the enemy (using the Manhattan Distance, as explained below) to farthest, with a maximum depth of 1. Finally, the moves not on the path to the enemy are re-processed with a depth of 2, taking inspiration from iterative deepening search. This is nicknamed the "overtime optimization" – its motivation is to allow the AI to take advantage of the rest of its allocated decision time to search its space more thoroughly.

Minimax has an exponential running time of $O(n^b)$ where b is the depth of the search tree. We effectively cut the running time in half using a technique known as alpha-beta pruning [3], which prunes branches of the tree that are worse than what is already computed. When applied to a standard Minimax tree, it requires less steps to run than the standard Minimax algorithm by pruning branches which do not play a role in influencing the final decision, without affecting the optimality of the solution. (Appendix A)

To efficiently traverse the game tree, we used a method inspired from *python-chess* [4], a popular Python Chess library [1]. The board is represented by both a snapshot of its current state and a stack of moves applied to it. When the Minimax algorithm moves down a edge, a move is pushed to the stack and the snapshot is updated. When the algorithm moves back up an edge, the stack is popped and the inverse move is applied to the snapshot. As a consequence, at the expense of a bit of extra memory to store the move stack, the game tree becomes very straight-forward to traverse while minimizing the amount of work for updating it.

3.2 Frontier Node Evaluation

Since the algorithm is often not able to traverse down to all leaf nodes (corresponding to an endgame states), an evaluation function had to be crafted to allow the algorithm to judge whether an internal tree node is more or less favorable to the player. The evaluated value of a node is decided using this simple decision tree:



Evaluation function decision tree.

A state located close to the enemy on the shortest path towards it is favored over one farther away, and even more favored over one not on the path. This rule is the basis for likely our most important insight: the most likely optimal move is along the shortest path towards the enemy. This has the effect of producing an aggressive agent that seeks to control as much terrain as possible compared to its opponent.

An immediate loss after the next move (suicide) is discouraged over a potential loss within the subsequent move to account for the fact that Minimax assumes a completely rational enemy. In other words, if the enemy makes a blunder and chooses sub-optimal move (especially true for the random agent), we want to take advantage of this to continue the game. Tie endgames are also discouraged slightly for a similar reason: a winning opportunity may present itself in the future. A win move (killing blow) is favored over simply a guaranteed winning branch to speed up the resolution of a win.

3.3 A* search algorithm

The agent uses A^* to calculate the shortest path from the agent to the enemy in order to prioritize the Minimax algorithm as well as evaluate the frontier node (see above). In addition, it uses A* as an optimization to check whether the board is in an endgame state or not; if the player and the enemy are unreachable from each other, this implies that the game is over. A^* is more effective in the cases where there's a clear path between the agent and the enemy, since Union-Find always needs to iterate through all the board tiles. If A^* reveals that

it indeed is over, the standard Union-Find algorithm is used to calculate the winner.

A^* search is effectively a combination of best first search and lowest cost first search, considering both cost of path so far and estimated path to goal in its selection of which node/ path to expand next. A^* search evaluates nodes by combining the cost incurred so far $g(n)$ and cost remaining $h(n)$ as $f(n) = g(n)+h(n)$. Because $g(n)$ is defined as the path cost from the starting node and $h(n)$ is defined to be an estimate of the cost of the cheapest path from said node n to the goal, we can deduce that $f(n)$ is an estimation of the shortest/cheapest solution path through the node n.

3.4 Manhattan distance

We used Manhattan distance to measuring all distances in the game because it best approximates the movement of an agent using the game rules and ignoring wall and enemy collisions. The distance between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is calculated as $|x_1 - x_2| + |y_1 - y_2|$[2].

4 Results

To evaluate the benefit of different parameter choices, in addition to pitting our agent against the random agent, we played our agent parameterized with different strategies against each-other. Here's a description of the different parameter combinations we used in our trials:

- **Random Agent**: Chooses action randomly from set of legal moves
- **Agent 1**: Minimax without move ordering strategy, and a simplified evaluation function (Lose=-1, Tie=0, Win=1, Other=0)
- **Agent 2**: Minimax with simplified evaluation function (Lose=-1, Tie=0, Win=1, Other=0)
- **Agent 3**: Minimax without "overtime optimization"
- **Agent 4**: Full Algorithm

A series of 500 games were simulated between them, and the results were as follows:

(n=500)	Agent 1	Agent 2	Agent 3	Agent 4
Random Agent	99.6%	99.6%	99.8%	99.8%
(n=500)	Agent 1	Agent 2	Agent 3	Agent 4
Agent 4	82.6%	58.6%	57.6%	—

Table 1. Each agent pitted against random agent, and the final algorithm was pitted against all the other agents.

These results demonstrate that the different strategies added to the agent each individually improve its outcome. Agent 4 outperforms all the other agents by a wide margin.

4.1 Pros/cons of chosen approach

One of the primary benefit of our algorithm choice is the simplicity and modularity of each component of the agent, relative to more complicated methods such as Monte Carlo Tree Search. Minimax is a standard approach to implementing agents for deterministic games with perfect information, and A* using Manhattan distance is a standard path-finding algorithm. Consequently, troubleshooting and expanding on it is also relatively simple.

Another benefit is the small number of parameters that need to be tuned, besides the maximum tree depths. Only the relative values of the evaluation function are important (e.g. a win is better than a tie, which is better than a loss). On the flip side, this also puts an upper bound on the algorithm's performance since parameter tuning can only bring it so far. To improve the algorithm, a better heuristic would need to be thought of.

One drawback to using the heuristic we chose is its inherit bias: an aggressive agent that chases after the opponent will perform best. This assumption may turn out to be wrong, in which case the agent is in fact not acting in an optimal manner.

5 Conclusion

5.1 Alternative techniques

During the development of the agent, we thought about implementing a memoizing mechanism to store the result of our endgame checks, with the current state of the board as cache key. However, this turned out to not be as useful as imagined, since game states don't repeat often themselves during the Minimax game tree traversal. In practice, our cache turned out to have a hit rate of around 1%, which doesn't justify the amount of memory it requires.

5.2 Future improvements

An important omission in the algorithm is not taking advantage of the 30 seconds offered to the agent during the first turn. The most obvious use of this time is to precompute and cache intermediate results to save computation time during upcoming turns. Based on particular future board configurations, the endgame statuses (e.g. win, loss, tie) or/and the result of the A* algorithm could be cached.

Based on the intuition that a player surrounded by many walls is at a potential disadvantage, an additional heuristic that could be leveraged is the difference in the number of walls next to the player relative to the enemy. This factor

could be added to the existing evaluation function to give the agent a bias towards putting the enemy next to walls and removing itself from claustrophobic situations.

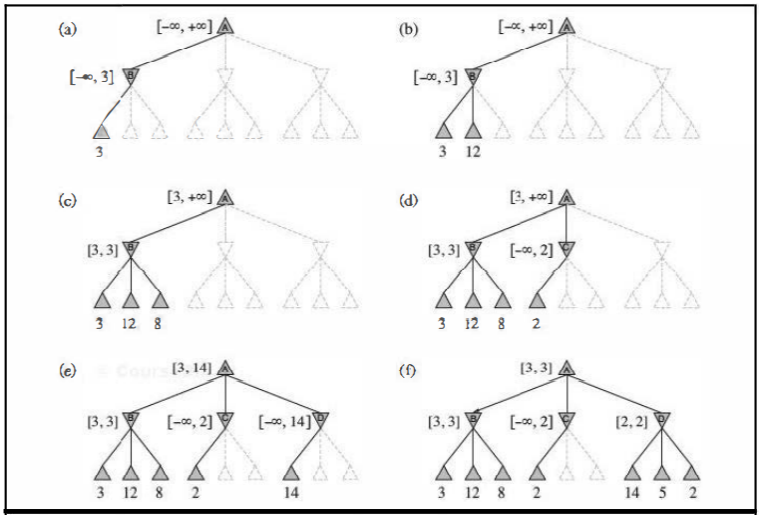
Another entirely different avenue the development could have gone down is implementing a Monte Carlo Tree Search algorithm. We decided against it with the rational that the game is relatively "memory-less"; in other words, moves beyond 2-3 turns in the past usually don't have a big impact on the outcome of either agent. Therefore, evaluating future games state beyond a depth of 2 is unlikely to benefit the agent. However, MCTS might have some merits due to it being an anytime algorithm, so is able to perform well in a highly time-constrained environment.

6 Appendix

6.1 Appendix A

Definition of Alpha-Beta pruning algorithm

(a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of alriWst 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha-beta pruning. (e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3 [3].



Alpha-Beta pruning algorithm [3].

References

[1] python-chess. https://python-chess.readthedocs.io/en/latest/.

[2] Paul E. Black. "manhattan distance", in dictionary of algorithms and data structures, Feb 2019.

[3] Russell Stuart and Norvig Peter. Artificial intelligence-a modern approach 3rd ed, 2016.