Two Second Time Constraint Survival!

Kara Best - 260894988 (Solo project)
Due Date: Wednesday Dec. 6th, 2023, 8:59PM EST

1. Motivation

This report aims to outline the design and performance of an agent developed to play ***Colosseum Survival!***, a 2-player turn-based strategy game, where each turn is constrained by a 2 second time limit. The approach taken employs a minimax search with various strategies and heuristics including: iterative deepening, alpha-beta pruning, move-ordering, and an evaluation function.
Strategies and heuristics were critical for the algorithm to effectively select the best move within the time-constraint. The evaluation function focuses on minimizing the distance between players for aggressive play and, depending on the depth reached within the time constraint, maximizing the number of moves the agent is able to make in its following turn or minimizing the number of moves the other agent is able to make. The moves are ordered at three points in the algorithm. The first occurs when child states are computed: they are ordered by descending proximity between players. The second occurs when the depth limit is incremented: the nodes at level 1 are ordered by descending utility computed in the previous iteration. The third applies the 'killer heuristic' and occurs when branches are pruned: the child node causing the pruning is placed at start of the list since it is likely it will cause pruning again in following iterations.
The agent consistently chooses the optimal move given the heuristics and the time-constraint and has a win-rate of 1.0 against the random agent (and beats me more often than I'd like to admit). While some improvements could be made, the agent maintains a good balance of offensive and defensive, which typically indicates high quality of play.

2. Agent Design

This section contains an in depth description of the data structures and functions used to implement the agent.

2.1 Data Structures

2.1.1 StudentAgent

The agent class includes the following attributes:

- *dir_map, moves, opposites:* Various attributes for the agent's movement logic (copied from *world.py*)
- *chess_board:* The current state of the game board
- *depth:* The depth limit of the minimax algorithm
- *max_step:* The maximum number of steps the agent can make in a turn

- *root:* The current state of the agent player
- *board_size:* The length of the board
- *turn:* The turn number
- *timer:* The start time of the current turn. Used to ensure the turn does not go over 2 seconds.

2.1.2 Node

The agent makes use of a game tree structure made of Node objects. Each node represents a state in the game and includes the following attributes:

- *pos:* The position of the player. At min nodes, this is the agent's position, and at max nodes, this is the position of the adversary.
- *boundary:* The direction of the boundary set by the player in position **pos**
- *children:* A list of the node's children, each a Node object, representing the possible future states from this state
- *level:* The depth of the node in the game tree. Even levels indicate it is a max node, odd levels indicate it is a min node.
- *end:* A flag indicating if the game ends at this node.
- *utility:* If 'end' if true, this is equal to either positive or negative the board size squared. Otherwise, it is determined by the evaluation function.
- *remove:* This is a flag indicating this node should not be considered in **minimax_decision**. It is set on endgame nodes that result in a loss and is at either level 1 or 2.

2.2 Functions

2.2.1 Main Logic

- **step:** This function begins by resetting the agent's timer to the current time in order to be able to check the time elapsed during the turn and ensure the time constraint of 2 seconds per turn is met. The turn counter is then incremented.
  If it is the first turn, the board size and max step attributes are set accordingly and the root node representing the current state of the game is initialized using the player's starting position.
  If, during the previous turn, the depth reached was larger than 2, the children for current game state will have already been generated. Thus, to avoid recomputing these children, the node representing the current game state is found in the game tree. The root will have been set in **minimax_decision** to the node containing the last move made by the agent at the end of the previous turn. The last move made by the opponent is then determined by comparing the current game board and the game board from the previous turn and using the adversary's current position. With this,

the node representing the current game state is searched for in the root's children and is then set as the root node.
Otherwise, the root node is simply created using the player's current position.
minimax_decision is then called to determine the best move which is then returned.

- **get_children:** This function generates all possible child nodes from a given node. Its parameters include the node whose children are being determine, game board configuration at node, the position being explored for moves, the adversary's position, the maximum number of steps left, the previous move's direction to prevent backtracking, and a list of visited positions (empty on the first call). The function iterates over all four possible directions in the current position. For each direction, if no boundary exists and the node has not previously been visited, a child node representing the placement of a barrier is created and appended to the node's list of children. Then, if the maximum number of steps has not been reached, the function checks if movement to the adjacent position in the current direction is possible, by ensuring it does not lead back to the previous position, there is no barrier preventing movement, and the opponent is not in that position. If the move is possible, the function recursively calls itself with the new position and updated parameters. After exploring all directions in a position, the position is added to the set of visited nodes to prevent duplicate children in subsequent recursive calls. Once all children have been determined, they are sorted by descending distance between players before the function returns.

- **minimax_decision:** This function initiates the minimax algorithm and determines the best move the agent can make. Its only parameter is the position of the adversary. The function first calls **get_children** to generate the children of the root node, if they have not been previously determined. Alpha is initialized to -infinity. It then iterates through each child for which it makes the hypothetical move represented by this child by copying the board and setting the barrier given in the child in **get_copy_board**, and calls **minimax_value** to determine its utility. Once it has retrieved the utility of the child, if the win flag is set meaning the move will result in an automatic win, it returns the move immediately, otherwise, it compares the returned value to the alpha value which is the max utility observed thus far, and, if it is larger, assigns the child as the current best move. At this point, if more than 1.97 seconds have elapsed during the turn, the current best move is immediately returned.
If there are no timing issues, once all children have been evaluated, if there is enough time to increment the depth limit determined by (time elapsed so far)≤ 0.5 if depth is currently 1 or $\leq 0.4 \cdot (depth)$ (to increase depth to 2: no more than 0.5 seconds can have passed, for 3: no more than 0.8 seconds, for 4: no more than 1.2, etc.), the function orders the roots children by descending utility and recursively calls itself.
When there is no longer enough time to increase the depth, the move resulting in the highest utility is returned.
Regardless of whether the algorithm terminated early due to timing or not, if the depth level reached exceeds 2, the children of the state the game will be in at the start of the next turn will have been generated, so, before the move is returned, the root node is updated to the selected child node and the agent's board attribute is updated to reflect the state after the move has been made by calling **update_board_root**. On

the following turn, the updated root and board will be used in **step** to find the game state during that turn in its children.

- **minimax_value:** This function contains most of the main logic for the minimax algorithm and alpha-beta pruning and is designed to compute the utility of a node. Its parameters include the node in question, the board configuration at the game state represented by the node, the current position of the min or max player, the position of the other player, the alpha value, and the beta value.
  It first checks if the current state of the game is an endgame scenario by calling **check_endgame**, unless it is for a node at level 1, it is the first turn of the game, and the game board is larger than 6. The node's *end* attribute is then updated to reflect if the game is over at this node or not. If it is the end of the game and my player has a higher score, the utility is set to the game board size squared. If my player has a lower score than the opponent, the utility is set to -(the size of the board squared) to ensure it will be lower than any other possible utility and will never be chosen. If the move results in an immediate win for my agent, i.e. the node is at level one and results in a win, the win flag is set to true, which, when returned, will result in **minimax_decision** to immediately return that move as its next step. On the other hand, if the move results in a loss either immediately, or at a level 2 node, meaning my player's move gave the opponent an opening to win, the node's *remove* flag is set to indicate that this node should not be considered in the future. The utility and the win flag are then returned.
  At this point, if more than 1.95 seconds have elapsed during the turn, the function immediately returns the node's utility which is 0 if the node has yet to be assigned a value or is equal to its utility determined in the previous turn.
  If there are no timing issues and the game does not end at this node but the depth level is reached, **evaluation** is called to compute the node's utility which is then returned.
  If the node isn't an endgame and hasn't reached the depth limit, the function generates all child nodes then recursively calls **minimax_value** on each child. After a recursive call returns, if the node's level is 1 and the child's *remove* flag has been set, the node's *remove* flag is set, no further children are evaluated, and the function returns.
  α-β pruning is applied during this process. If it is a max node, each value returned by the recursive call is compared to the current α value and replaces it if it is larger. If it is a min node, each value is compared to the current β value and replaces it if it is smaller. Following this, if there is an inconsistency, $\alpha \geq \beta$, the remaining branches are 'pruned' and the algorithm does not evaluate any more children for this node. In addition, the killer heuristic is implemented here to maximize pruning in future iterations by placing the node causing the pruning at the start of the list.
  After the utility has been determined for every child or the remaining children have been pruned, the algorithm returns β if it is a min node, and α if it is a max node.

- **evaluation:** This function determines the utility of a node by applying an evaluation function. The evaluation function considers several features of the board. The first feature considers the proximity between players which is computed by subtracting the distance between players from the maximum possible distance between them so that smaller distances have a higher utility : feature 1 = 2(board length) − distance b/w players. This is weighted at 2 so that it remains significant even if the number of children is large. The second feature is the number of moves available to the min or max player. The feature is weighted by -1/(max step) for min nodes to minimize the adversary's movement, and is weighted 1/(max step) for max nodes to maximize my player's movement. They are divided by the max step to ensure that they never outweigh the utilities of other nodes resulting in the end of the game. This feature is equal to the number of children of the node. While this significantly increases the computation time of this function, the children generated are used in the following iteration of **minimax_decision** when the depth limit is incremented.

2.2.2 Helper Functions

- **check_valid_step** and **check_endgame:** These functions have been copied from world.py.
- **get_copy_board:** This function produces a copy of the board including a given barrier.
- **update_board_root:** This function updates the agent's board and root node to the one's given.

3. Performance Analysis

3.1 Depth

$$\text{Depth on a size 12 board} = \begin{cases} 1 & \text{if } turn \leq 5 \\ 2-3 & \text{if } 5 < turn \leq 30 \\ 4 & \text{if } turn \geq 30 \end{cases}$$

On a size 12 board, the agent is able to reach a depth of 1 for the first 3-5 turns, then during the middle game, depending on positioning and possible moves, it is able to reach a depth of 2 and occasionally 3. If the game exceeds around 30 turns, it is able to reach a depth of 4 since there are few moves to consider. Since the agent uses an iterative deepening algorithm, the depth explored is the same for all branches unless a node is an endgame scenario.

3.2 Breadth

Given b = branching factor, d = depth limit

$$\text{Breadth on a size 12 board} = \begin{cases} b & \text{if } d = 1 \\ b/2 & \text{if } d > 1 \end{cases}$$

On a size 12 board, the agent will consider every single possible move at the first level, then, in subsequent iterations after the depth limit is increased, moves at the first level resulting in an immediate loss or an opening for the opponent to win on their turn are no longer considered. If all moves prior to the last move at level 1 fall in this category, the last move is considered regardless since a loss is guaranteed. At levels 2 and below, since alpha-beta pruning and move-ordering is applied, the breadth is roughly reduced to b/2.

3.3 Complexity

The maximum possible number of moves from a given state is: $4^2 \cdot \frac{max\_step \cdot (max\_step+1)}{2}$ where max_step is equal to $\lfloor \frac{n+1}{2} \rfloor$. Assuming the branching factor,b, is proportional to the max number of moves, it is proportional to n^2. Considering the big-O notation for minimax with alpha-beta pruning is $O(b^d)$, where d is the depth, we get:

$$O((n^2)^d)$$

3.4 Heuristics

Let b be the average branching factor and d be the depth limit.

- Evaluation function and iterative deepening: With the evaluation function and iterative deepening, the depth explored is d and the breadth is b. Without this, the algorithm would operate in a depth-first manner, and attempt to explore the entire game tree, exceeding the time limit. The evaluation function provides a way to estimate the value of a state that is not an endgame scenario.
- Alpha-Beta pruning and move-ordering: With alpha-beta pruning and move-ordering, which allows more effective pruning, it can be estimated that the breadth is reduced to $b/2$, which gives the algorithm more time to explore deeper.

3.5 Win-rate Predictions

- Random agent: 1.0
  Against the random agent, I predict my agent will have a win-rate of 1.0 which is the win-rate achieved during testing.
- Dave: 0.7
  Against the average human player, I think my agent would have a win-rate of 0.7 based off of the games I have played against it.
- Classmates: 0.6
  Against my classmates' agents, I predict my agent will have a win-rate of around 0.6.

4. Advantages and Disadvantages

Some advantages of my approach include:

- Game state saving: If the depth limit exceeded 2 in the previous turn, the first level of the game tree has already been computed, thus, making the turn more efficient.
- Move-ordering: The move ordering has proven to be very effective in increasing the amount of pruning. The sorting by previously computed utility when the depth limit

is increased, also ensures that, when a time-out occurs, the moves that are able to be considered within the time limit have the highest estimated utility.

- Timing check when increasing the depth limit: The algorithm is able to somewhat accurately estimate whether it will have the time to re-execute **minimax_decision** with incremented depth and behave accordingly. This has resulted in a significantly reduced number of time-outs allowing the algorithm to select the optimal move while considering all paths.

Some disadvantages of my approach include:

- Inefficient evaluation function: While computing the following level in the evaluation function may save time when the depth level is incremented and on following turns, it may be hindering the algorithms ability to explore deeper.

- Time-outs: When the branching factor is large, the occasional time-out occurs, causing the agent to not fully consider all moves. This has lead to sub-optimal behavior during testing.

- Generates all children before evaluation: Every child of a node is computed before they are individually evaluated. This causes time to be allocated towards computing children that may end up getting pruned.

5. Other Approaches

Other approaches were explored for the overall algorithm, evaluation function, and move ordering but they did not perform as well as the final design. The first approach taken applied a depth-limited minimax search instead of iterative deepening, resulting in very poor performance on board sizes larger than 6, since the time constraint caused the algorithm to terminate early on every turn. This caused the first several branches to be explored to the depth-limit while the subsequent branches would not be considered at all. In the current design, however, every branch is considered at each level, unless they are pruned, and the risk of timing out is reduced by considering how much time has elapsed before incrementing the depth.

Regarding the evaluation function, the approach I initially took was considering both the number of moves my player could make from the given state and the number of moves the adversary could make as features. At the set depth of 2, this involved generating all children of the node which represent the moves I can make, then creating a copy of the node and determining the children of this node if it were the adversaries move again. This made the evaluation function very time-consuming with little benefit since the children generated to determine the possible moves the adversary could make were not legal moves because they ”skipped” a turn and thus, could not be saved to speed up future turns. This greatly reduced the performance of the agent since, due to the time constraint, the turn would terminate too early to evaluate enough moves to effectively select the best one.

6. Improvements

As stated previously, a key drawback of the algorithm is its approach of generating all children of a node first and then evaluating each one, since this results in time lost generating children that may end up being pruned. A significant improvement would be to assess each child as soon as it's generated, so that if the following children are pruned, no time is lost determining further children.