COMP 424 Final Project

Melek Deniz Colak — Joseph Zeidan

December 2023

1 Overview of Approach

Executive Summary

On a high-level basis note, our game algorithm approach sheds light on two complementary yet powerful concepts seen in class: Alpha-Beta pruning and Iterative Deepening Search (IDS).

By running the alpha-beta algorithm and storing the best move in between each iteration, our agent ensures optimal decision-making within time constraints that are set to 2 seconds. Within each iteration of alpha-beta, our agent will first tactically explore the moves with the most potential using move ordering, increasing the likelihood of pruning and the overall efficiency of our program.

However, how are we able to sort moves based on their potential of winning? This is where heuristic design comes into the picture. Maintaining strong and flexible heuristics contributes to an effective decision-making process that adapts to different stages of the game. In addition, we consider actual win-loss data within our alpha-beta by checking for whether the board state is an endgame state which adds to the predictive power of our algorithm. Hence, the achieved play quality of the agent reflects a competitive agent that should be able to win consistently against the Random Agent, win most of the time against the Human Agent, and compete well with other agents.

Detailed Description

Now, let us dive into a more theoretical basis of our approach with a detailed explanation of our agent design and a blueprint of how we manipulated different functions to achieve strategic accuracy and efficiency. The work is concentrated inside the `step` function. We start with checking whether there is an obvious winning move available with the `get_winning_move` function. If the opponent already has three borders around it, we check if the position to trap it is reachable and simply make the move to block the opponent and win the game instantly. However, if an obvious win move is not possible, we move into our main AI algorithm which will be elaborated below.

Allowed Moves:

The initial challenge for this project was getting the valid complete moves (row, column, direction for wall) that our agent could make given a board state. Because we can make more than one step in a single turn, we decided to implement a backtracking algorithm to achieve this goal. We return a dictionary called `allowed_moves` in the format of key = Tuples indicating the row and column of an available tile and value = List of all the valid directions (0: up, 1: right, 2: down, 3: left) to place a wall.

In the recursive `get_successors` function, we use the `allowed_dirs` (from the `random_agent`) to consider both the walls we can place and the tiles we can move into toward the available direction in the recursive call. Every recursive call moves us one step and we make sure we don't consider moves more than *max_step* away, we keep track of how many steps we have moved to come to the tile we're considering with the counter variable i. Knowing the walls we can place given the tile we're on (in the backtracking) gives us a valid move we can add to *allowed_moves*.

To guarantee no duplicate moves, we made sure that we don't move back during backtracking by utilizing a *prev_dir* variable and make sure we don't consider the opposite of that direction for the next move and checked if the move already exists in the dictionary before adding it.

The most important part about the `get_successors` function is that it checks for the obvious losing move which is a move that puts us in a board state where our agent has 3 walls around it. We don't add this move to *allowed_moves* unless no other move is available.

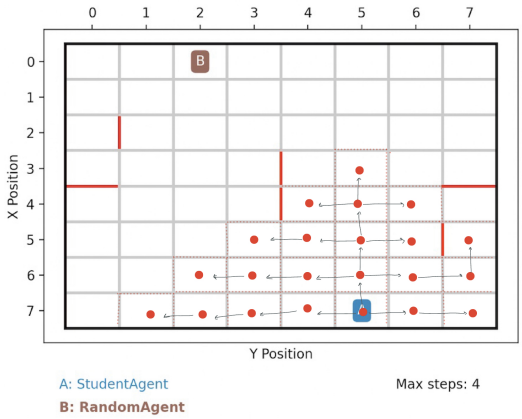

Figure 1: Key-Value Pairs: [((7, 5), [0, 1, 3]), ((6, 5), [0, 1, 2, 3]), ((5, 5), [0, 1, 2, 3]), ((4, 5), [0, 1, 2, 3]), ((3, 5), [0, 1, 2, 3]), ((4, 6), [0, 1, 2, 3]), ((4, 4), [0, 1, 2]), ((5, 6), [0, 2, 3]), ((6, 6), [0, 1, 2, 3]), ((5, 4), [0, 1, 2, 3]), ((6, 4), [0, 1, 2, 3]), ((5, 3), [0, 1, 2, 3]), ((6, 7), [0, 2, 3]), ((5, 7), [0, 2]), ((7, 7), [0, 3]), ((7, 6), [0, 1, 3]), ((7, 4), [0, 1, 3]), ((7, 3), [0, 1, 3]), ((6, 3), [0, 1, 2, 3]), ((6, 2), [0, 1, 2, 3]), ((7, 2), [0, 1, 3]), ((7, 1), [0, 1, 3])]

Heuristic Evaluation Function:
The agent's decision-making process is guided by a heuristic evaluation function. This function assigns a numerical value to a given game state, reflecting the desirability of that state for the player. As you can see inside the `eval_function`, we have three key heuristics:

- *available_move_number*: `get_number_of_reachable_tiles` function returns the number of tiles reachable by the agent in the current board state. We want our agent to prioritize moving to positions that will give it more options to be able to get out of being trapped. In addition, a high number of tiles reachable indicates having control over more of the board.

- *aggression_heuristic*: Computed by calling the `is_aggressive_move` that checks whether a given move can be considered aggressive based on whether the move passed into is a move placing a wall next to the opponent. If any of the conditions are true, the function returns the aggression heuristic value (Initially set to 2). Otherwise, it returns 0.

- *winning_heuristic*: Computed by calling the function `winning_heuristic` that checks if the adversary has three sides blocked with walls and if the `my_move` passed into it traps the opponent and returns a very high value to incentivise alpha beta to pick this move since it guarantees a win.

Keep in mind that in this game each player will try to maximize their overall score, so we calculate the overall heuristic of each move as *the sum of our heuristics - the sum of the opponent's heuristics*. In the opponent's heuristics, we don't include the `winning_heuristic` because if we find the winning heuristic the game is over therefore adversary's chances of winning in the next move doesn't matter when we already won.

It's important to state that the way that we implemented alpha beta makes it so that when `eval_function` is called from the max it takes our agent to be the main player and when it is called from the min, it takes the opponent's agent to be the main player. Therefore when `eval_function` is called from the min, it is calculating sum of the opponent's heuristics - the sum of our heuristics (not including `winning_heuristic`)

Alpha-Beta Pruning:
It is safe to say that this part of the project took most of the time to finalize. The following 3 functions are executing this search algorithm: `alpha_beta`, `alpha_beta_max` (Max node in the tree), and `alpha_beta_min` (Min node in the tree). The second and third functions present big similarities in terms of implementations and logic.

Starting with the cutoff function that is employed to determine whether a cutoff condition is met based on the game state and depth. The `cutoff` function returns a boolean indicating whether we're at a cutoff, and a float indicating the end score or heuristic returned by the `eval_function`. In fact, a cutoff can happen in two different cases, either the game ended (this is checked using a

Union-Find algorithm in the `endgame` function that is taken from the world.py file), or the current depth of tree is equal to the maximum depth. However, if none of the above conditions are met, we should proceed with the exploration of the game tree.
This is also where we introduce the concept of move ordering. We get all possible successor moves, then we use the `dict_to_heap` function to get the top successors, limiting the breadth of the tree strategically. In this function we build a priority queue (using the built in *heapq.nlargest*) from the list of (heuristic, move) pairs based on the `eval_function`'s computation of each successor's heuristic.

Next, we loop through the top successors and call the `alpha_beta_min` function making sure to switch *curr_move* and *adv_move*. This switch allows us to keep the structure of `alpha_beta_min` almost the same with `alpha_beta_max` by simply switching players. In the `alpha_beta_min` function we do everything the same and switch players back when calling the `alpha_beta_max` function within it. This turn-taking continues until we reach a cutoff within the most left branch.

As the min and max functions return their chosen successor's heuristics, alpha and beta values get updated as they go up in the tree. As the values go up in the tree, other successors also get computed or pruned.

By providing a numeric value representing the desirability of each move, and returning this value from min or max calls with the move a successor represents and the the `alpha_beta_max` function loops through the successors list, updating the alpha value if needed and selecting the move with the highest heuristic evaluation and considering some unpromising values to prune from some of the branches.

The following explains the process of updating alpha and beta values and pruning according to them.

- `alpha_beta_max`: **(1)** Initialize the helper variable called *eval* representing the max value returned from the successors to negative infinity and the *new_alpha* variable to the input alpha parameter. **(2)** Update *eval* if the successor (min call) returns a bigger value than *eval*. **(3)** To be able to return the move that is associated with the heuristics in the root node, we keep track of the move that the successor node represents along with its value. To be able to do this we form a dictionary called *move_alpha_dict* mapping moves to their values returned from the min call. **(4)** Prune the siblings of the successor if *eval* (current max alpha value) is greater than or equal to the beta value. Pruning is done by returning *eval* and the move associated with it and thus stopping the loop from exploring the siblings. **(5)** Set *new_alpha* to *eval* if *eval* is bigger than *new_alpha* and pass this down as alpha for the next sibling by passing it to the min call. **(6)** Return the move associated with the max alpha value by utilizing the *move_alpha_dict*.

- `alpha_beta_min`: **(1)** Initializes the *eval* which represents the min value returned from the successors to positive infinity and the *new_beta* value

to the input beta parameter **(2)** Update the *new_beta* if the successor (max call) returns a smaller value than *eval*. **(3)** Prune the siblings of the successor if alpha is bigger than or equal to *eval* (current min beta value).

Time Check: The class *Timer* with its corresponding three functions `start`, `elapsed_time`, `check_timeout` makes sure we don't pass the time limit and contributes to the mechanism of the iteration as follows:

- At the beginning of the step function, we initialize a timer object and log the start time.
- We pass this timer object to the functions we call. This allows us to *check_time* multiple times throughout the runtime.
  Iterative Deepening:
  In this part of the project, we worked on improving the efficiency of our Student agent. Iterative deepening enhances the alpha-beta pruning algorithm by calling the alpha-beta function with increasing maximum depths until a specified time limit is reached. Using a while loop and a try-catch structure, The search algorithm proceeds until the maximum allowed time is reached. Here, iterations of the depth-limited search will stop when depth ≥ 10.
- If the time limit is reached during a particular depth iteration (limit= 1.9 seconds), the TimeoutException is raised, and the algorithm gracefully terminates the search while returning the last updated version of the best move before termination.

2 Quantitative Performance

1. The agent's lookahead maximum depth it's able to reach under two seconds on the size 12 board ranges from 2 to 6, influenced by the initial board state. Typically, the initial depth sets the tone for the rest of the game. For instance, if the depth achieved during the first step of the game is 3 or 6, then the rest of the game more likely fluctuates respectively in between 2-4 or 5-7. Although not all of the branches has the same depth because the game might be over (`endgame` function returns true) before the maximum depth for the iteration of alpha beta we're in is reached. To prevent excessive computation in rare cases, a maximum depth cap of 15 is implemented, avoiding extended periods spent in a single branch.
2. The agent's breadth, representing the best possible moves of a given state of the board, is controllable just like the depth. It is currently set to the board size + 3 (handled this in `dict_to_heap` as n in the *heapq.nlargest*). For a board of size 12, the breadth is equal to 15. This parameter is crucial, impacting the relation between depth and breadth. A higher breadth allows the exploration of fewer depths due to time constraints, while a lower breadth enables deeper tree exploration. To optimize performance,

experimentation with different values for both depth and breadth is necessary. The breadth is the same for both max and min player.

3. As the board size increases, the breadth of the search space also expands, accommodating more potential moves at each stage. However, the scale of depth is inversely affected by a larger board, as the increased number of available moves per stage limits the feasibility of exploring the game tree to deeper levels. Note that in the absence of pruning, the complexity of alpha beta is equivalent to minimax, both having an exponential time complexity of $O(b^d)$ (b: breadth, d: depth). However, with Alpha-Beta's pruning capabilities and with a good heuristic, it often significantly reduces the number of nodes explored, leading to a better time complexity of $O(b^{(d/2)})$, effectively halving the exploration in the best case.

4. The breadth and depth achievable by our iterative deepening function is completely dependent on how much time we can allocate to alpha beta. Move ordering based on a good heuristic function allows us to consider the best moves first and therefore makes pruning more likely to happen. This saves time per iteration of alpha-beta, allowing for more iterations to be completed within the time limit. Considering we have saved time, we can then decide to increase the number of successors we're considering thus increase the breadth. For example, before implementing mover ordering and capping breadth, we were able to explore depth of 3 within 40 seconds, now we're able to reliably explore depth 3 within 2 seconds implying that if we had 40 seconds to explore, we would reach a vastly larger depth. In this case, we have sacrificed the breadth for depth and performance since without move ordering we were considering all successors.

5. Prediction of win-rates in the evaluation against:

   - Random Agent: After multiple testings with different board size, our agent always guarantees to win over the Random Agent.
   - Human Agent: If the player has an average knowledge of this game, we expect our agent to win 80% of the time.
   - Classmates' Agents: Depending on how well the opponent agent has been implemented. For an average agent, we are hoping for a 70% win rate. The more sophisticated and advanced the opponent agent is, the less win rate we are expecting. Maybe

3 Advantages and Disadvantages of our approach:

We chose to implement alpha-beta over Monte Carlo Tree Search because of the deterministic nature of this games. Now, compared to the minimax algorithm, alpha-beta pruning seeks to reduce the number of nodes evaluated in the minimax tree with pruning saving time and computation. Our agent also limits the breadth time to more promising sub-trees using move-ordering, which enables

a deeper search. As an advantage, we can handle larger board sizes and diverse game scenarios within the time constraints. Our alpha-beta algorithm with an iterative deepening approach ensures that the agent finds equally good or better moves as time passes. Thus even though we don't go for the best move from the start, we avoid returning a bad move because our alpha-beta didn't complete in time as it would in a set max-depth alpha-beta case.

On another hand, our agent represents some disadvantages that limits its efficiency. The absence of memoization in iterative deepening search is one of them. Our approach lacks the ability to "memorize" computations at shallower depths for reuse in deeper iterations. This results in excessive recomputation, where information obtained at a certain depth is not retained or passed down to subsequent iterations of the alpha-beta algorithm.
Another drawback in our implementation is having some sort of unscaled heuristic. In other words, we are not able to put a limited range on all the possible outcomes of our heuristics. This can affect the accuracy of evaluations in certain contexts, demanding further refinement for nuanced decision-making.

4 Failed Approaches

The early version of the approach over simplified alpha-beta algorithm, as it didn't took into account the understanding for the two-player nature of the game and hence exhibited chaotic behavior and non-optimal decisions against the opponent. We realized that not updating the corresponding values for the *adv_pos* was an error. Therefor, we improved our strategy on how to take into account alternating players moves. This last version showed a substantial increase towards better strategic decisions.
The way the simulated games were implemented created another issue for us. Initially, we entered our simulated games using the identical chessboard object that we used in the real game. Because the chessboard was altered during the simulations, we would receive an error when we attempted to return a correct move. As a result, we were using our step function with erroneous movements. In order to address this issue, we made a deep duplicate of the chessboard object that would be utilized in the simulations.
When we implemented alpha-beta with turn taking and manipulating the board correctly, we came across another problem. Our search tree was going through every single successor possible and therefore our program took a significant amount of time. To fix this, we implemented a move ordering and limited the number of successors. By utilizing a priority queue to sort the successors with best heuristics and putting a cap on the number of successors we have for different board sizes in the `dict_to_heap` function, we managed to reduce our runtime significantly and it allowed us to be able to explore the search tree deeper than before.

5 Further Improvements

In order to enhance the player's performance, we can still work on implementing more heuristics. For instance, we could consider improving a wall continuation heuristic, which involves prioritizing moves that contribute to the continuation of walls or patterns rather than just focusing on free space. This can be achieved by introducing a more sophisticated evaluation function that takes into account the strategic importance of maintaining and expanding walls.

As we previously saw in heuristics, being near the opponent favorize the aggressive play to trap them, while staying far can be better for survival and defense mode. From this, having some sort of balanced center heuristic might be a great idea. By that, our agent can prioritize moves that control the center of the game board. The center is often a strategic location that provides flexibility and control over the overall game space.

Another improvement would be to have different strategies for different states of the game: start, middle and end as suggested by the project instructions document. We have started to implement this idea with the function `get_game_state` however decided against it due to time constraints.

To save time and computing power, we can add a *found_winning_move* boolean variable to the iterative deepening function where in every alpha beta call we check whether the search tree of that iteration's depth has found a winning move (to be decided based on the alpha-beta's return heuristic) and stop further iterations.

Overall, the most significant improvement would be to test and tweak our existing approach to find the best combination of values for the heuristics values we choose and the number of successors we allow alpha-beta to expand on. Scaling the heuristics between -1 and 1 so that we have a guaranteed win value of 1 and guaranteed lose value of -1 and every other heuristic in between would be another improvement to greatly benefit our performance.

To address the drawback memoization mentioned previously, we can use some memoization strategy (implementing a caching mechanism) by storing and reusing the previous computed result at a given depth. This would optimize the iterative deepening process and avoid redundant calculations overall efficiency.