Game Programming II

Report

By: *Maksym Husak*

Södertörn University | School of Natural Science, Technology and Environmental Studies

Game Programming II

Media technology | autumn semester *2022*

Games programme

1 Development process

- **What problems did you encounter in developing your solution?**

In the beginning of the project I remember the biggest problem was just not knowing what to expect from it and not having a clear plan in place. Getting over this hurdle was a major turning point for the success of the project, as looking at blank assets in unity and task description was like staring at a blank wall. It felt overwhelming and discouraging but after going over all files that were in there and breaking tasks down into smaller problems helped ease the stress a little bit.

Key thing for me was definitely managing the game state effectively. The representation of the board and the pieces in memory needs to be efficient and easy to manipulate, especially since the Minimax algorithm requires creating and evaluating numerous game states. I would need to ensure that the game state is updated correctly after each move and that the win conditions are accurately checked and updated. Establishing classes and methods for this was incredibly complicated for my level. Understanding and implementing the Minimax algorithm for AI decision-making in a game like this was somewhat complex. This algorithm kind of involves good grasp of recursion and tree data structures, and it can be challenging to optimize, especially considering the high branching factor of the game.

By far the hardest and most time-consuming part was debugging, testing and fixing scripts based on feedback. Ensuring that all possible game scenarios are handled correctly, and that the AI behaves as expected, required a lot of testing testing. Debugging issues related to game logic or the Minimax algorithm are time-consuming, especially if the bugs result in subtle errors in gameplay or AI decision-making in the final moments of the game that require time to get to them. Thankfully, the graphical user interface (GUI) and user experience design didn't present many challenges since everything was already there and there was absolutely no demand on creating anything more complicated or special.

- **How did you approach the problems?**

I started by thoroughly understanding the Minimax algorithm through research and study. I learned that diving into implementation without a solid grasp could lead to confusion later on, so I started by learning what Chinese Checkers even is and how it plays by playing for 1-2 hours online and reading about it overall. I then tried to see if there are any digital versions with source code available. I thought that even if it was in different programming language or engine it would still offer valuable insights of how it can be done, but I didn't really find anything usefull. After understanding the game rules, blank project and task guidelines I started drawing out a rough plan for the program and discussing it with teachers and classmates.

Managing the game board and state was the focus, where I chose appropriate data structures. I initially underestimated the complexity of updating the game state, which led to bugs and inefficiencies that I had to address later. But at this point since I'm very familiar with the scripts it takes little time to find where is the issues are exactly in the code. Once I locate the issues, I analyze the code and make necessary modifications to address them. It's error identification during testing phase that's definitely is the weak link in my developing process chain.

- **How did you solve the problems?**

Constant testing after edits in code were important components of my development approach. To make sure the AI performed as planned and the game rules were correctly implemented, it was essential to conduct frequent testing in a variety of circumstances. I discovered how crucial comprehensive testing is, which I had before disregarded and so allowed serious flaws to remain in the game. I used an iterative development process with frequent feedback cycles from teachers and classmates for both the gameplay and the code, which helped identify problems and learn on fixing them. I also discovered how crucial it is to provide up-to-date, concise documentation and comments, particularly for intricate sections like the game logic and board state.

I faced challenges and made mistakes, but these were valuable learning opportunities. Through effective problem-solving, time management, and seeking help when necessary, I navigated these challenges. I thing the project was not just about coding a game but also about learning and applying several important computer science concepts.

- **Do you now see any improvements that you could have made?**

I definitely could have done more thorough testing by completing the game with as many possible outcomes as I could. The game itself most definitely could be further improved by adding some sound design, animations, perhaps even multiple camera angles, but core features are more important in this case. As for the code itself there are definitely some 'tips' that I've learned more specifically for each scripts, such as:

BoardModel

Game State Management The methods for managing the game state and checking win conditions are crucial. However, if not implemented efficiently, they could lead to performance issues, especially in a game with as many possible states as Chinese Checkers. The script can be considered **potentially redundant,** the use of separate methods for starting and restarting the game might have overlapping code. Consolidating common functionalities into shared methods could improve scripts efficiency but I couldn't spend more time on something that already works okay.

IMiniMax

Algorithm Complexity: Implementing the Minimax algorithm is complex and requires careful consideration of recursion and tree traversal. If not optimized (more on that in final part), the algorithm could be inefficient, leading to slow AI responses, but since the game is by modern computation power standards extremely light (one board state taking around 1mb of storage), it's okay. Poorly optimized 'Score' and 'Expand' methods in State.cs could severely affect performance of the mini max algorithm as well. **Lack of Flexibility**: The Minimax implementation seems tightly

coupled with the game's specific rules, which might limit its reusability for other scenarios or games but that's fine for this project.

Specific Analysis

In this section I would like to more generally describe how I understand each script and answer more specific questions about them as well.

The BoardController.cs script is a crucial part of the game's architecture, implementing the Controller component of the MVC (Model-View-Controller) pattern. It includes namespaces for handling collections and utilizing Unity's core functionalities.It defines aliases for Position and Coordinate to use Unity's Vector2Int and Vector2 types, which likely simplifies the handling of board positions and coordinates. It declares a BoardController class that inherits from MonoBehaviour and implements an IBoardListener interface, so it kind of it listens for board-related events. There are serialized fields, such as a GameObject for the board's visual representation, so that some variables can be set directly in the Unity Inspector for easier setup. It initializes an IBoardModel instance, making the separation between the game's logic (model) and its representation (controller/view). Basically This script contains methods for processing player inputs, updating the game state, and communicating with the BoardModel to reflect changes in the game's logic

The BoardModel.cs script serves as the backbone for the game's logic, aligning with the Model component of the MVC pattern. It uses collections and Unity's core functionalities, similar to BoardController.cs. It imports the Minimax namespace, so that the game logic includes or supports AI decision-making based on the MiniMax algorithm, which was the goal in this turn-based Chinese Checkers game. The script defines aliases for Position using Unity's Vector2Int, indicating a focus on grid-based logic for the game board, it also includes operations for serialization, to make game state saving/loading functionalities. I would describe this script as pretty essential for managing the game logic, including keeping track of player moves, the state of the board, and ensuring the game adheres to Chinese Checkers rules.

As the View component in the MVC paradigm, the BoardView.cs script is mostly concerned with the game's visual depiction. It incorporates standard and Unity-specific namespaces, including those for UI elements, it handles user interface components and ssimilarly to the other scripts, it defines aliases for Position and Coordinate using Unity's vector types for easy handling of spatial data. The class BoardView inherits from MonoBehaviour and implements an IBoardListener interface, so it can react to changes in the game board's state. The Unity Inspector has serialized fields for configuring the board and piece prefabs, making it simple to alter the visual components of the game. The player must be able to see the board's layout and the pieces' locations while playing, which is made possible by this script.

The GUIHandler script is responsible for managing the game's graphical user interface, particularly in handling user interactions through GUI elements. It inherits from MonoBehaviour, which is standard for scripts in Unity that require access to game object components and event handling. The script is described as a receiver for events from GUI elements, meaning it's used to trigger actions based on user inputs, such as button clicks or selections from dropdown menus. It maintains a default number of players (numPlayers) and uses a conversion array (menuConversion) to translate dropdown selections into the corresponding number of players. There's a reference to a board model instance (IBoardModel board), which implies the GUI interacts directly with the game's logic to apply user selections or settings.

Of course to calculate AI moves the IMiniMax.cs script defines interfaces related to the MiniMax algorithm, a common strategy for AI in games that involves simulating future moves to choose the best outcome. It defines the IState interface with methods like Expand(IPlayer player) to generate possible future states from the current state and Score(IPlayer player, IState state) to evaluate the desirability of a state for a player. This setup is typical for turn-based games where the AI needs to consider multiple future game states. The IPlayer interface is designed to represent plauers in the game (either human players or AI) within the MiniMax framework. This allows the algorithm to simulate moves for both the AI and its opponents.

The State script is intended to maintain the game's state, which is essential for putting game logic into practice, particularly when using the MiniMax algorithm in AI decision-making. The script makes use of Position through Unity's Vector2Int for managing board positions.With functions like Expand for generating possible moves and Score for evaluating the desirability of moves within the MiniMax algorithm this script is essential for the AI's ability to simulate and evaluate different game states, guiding the Ai's strategy by calculating the best possible moves.

The Utility .cs script offers general-purpose functions to support game functionality, including converting between board positions and pixel/screen coordinates. It is crucial for games where the logical representation of the game world is accurately mapped to the visual representation on the screen. The script also includes functions for board setup and piece placement, such as calculating distances and converting coordinates. These utilities are pretty much used across the game.

BoardModel

GetPiece makes a call to AllPossiblePosstions, why?

The GetPiece method implementation clears a list of possible moves (legalMoves.Clear()), then calls AllPossiblePositions(pos, true) to calculate all possible moves for the player from the given position before returning the piece at the specified position. This suggests that the method is used not only to retrieve a piece but also to evaluate and store possible moves from its position. The call to AllPossiblePositions within GetPiece kind of indicates a dual purpose: retrieving a piece and understanding its potential legal moves from the given position.

What is the intended difference between GetPiece and GetCurrentPiecePosition?

GetPiece(Position pos) retrieves the piece at the specified board position and also calculates all possible moves for that piece, storing them in legalMoves. This serves a dual purpose of identifying the piece and evaluating its mobility on the board.GetCurrentPiecePosition(Position position) directly returns the piece at the given position without any additional processing or move calculation. This method is straightforward in its purpose, focusing solely on retrieving the piece at a specific x y location. The difference between GetPiece and GetCurrentPiecePosition lies in their secondary functions: GetPiece is involved in move calculation, while GetCurrentPiecePosition simply retrieves the piece without any additional logic.

2 Board data structure

- **What data structure did you use for the board states?**

The data structure used for the board states, as inferred from the BoardModel.cs and Utility.cs scripts, is a two-dimensional array of *Piece* objects . This data structure is effective for board games because it mirrors the physical layout of a game board, allowing for straightforward implementation of game rules and logic, such as movement rules, checking for win conditions, and updating the board state(Dahlin, 2018). Here's a breakdown of how this is implemented:

- 1. Two-Dimensional Array:

The board is represented as a 2D array (`Piece[,] board`). Each element in this array corresponds to a position on the game board.
This structure allows for easy access to each position on the board using row and column indices, which is efficient for games where positions need to be frequently accessed and updated.

- 2. Piece Objects:

Each cell in the array holds a Piece object or a similar structure.
These objects would represent the individual pieces on the board, containing information about their state, such as their type or whether a position is taken by some piece already.

- 3. Dimensions from Utility.cs:

The dimensions of this array are defined by the xMin, yMin, xMax, and yMax values from the Utility.cs script, indicating a grid size of 17x17 in thisspecific implementation.

- 4. Board Initialization:

The board is initialized in the BoardModel.cs script, where the 2d array is created and populated according to the game's starting layout.

- **Compared to other potential solutions, why was this the best?**

Alternative data structures and their drawbacks:

Graphs:

Even while (especially in a non-rectangular arrangement) a graph data structure may more explicitly depict the links between positions, the implementation would become more complicated (Geeksforgeeks, 2024a). It could also be more difficult to access and update positions than with a 2D array.

Linear arrays:

It would also be possible to use a single-dimensional array, but this would complicate the logic for accessing adjacent positions and diagonal movements, making the code more difficult to read and update.

Custom Data structures:

For the size and complexity of a Chinese Checkers game, custom or more complicated data structures might not offer much of an advantage over a 2D array, even though they might provide a more customized solution in long term or if I wanted to create new/different variant of the game (which is a questinable idea for a game that is over a century old).

Choosing a two-dimensional array to represent the board state in a Chinese Checkers game, as compared to other potential solutions, came down to several key advantages specific to the requirements of board games and the nature of Chinese Checkers itself. Here's why I think this choice might be considered the best for this scenario:

1. **Efficient Access and updating the pieces on board**:

2D arrays allow for efficient access and updates to individual positions using row and column indices. This is crucial for a game like Chinese Checkers where pieces are frequently moved, and the state of each position needs to be checked regularly.

2. **Simplicity in implementation**:

Using a 2D array is straightforward and doesn't require complex data structures. This can make the code easier to understand and maintain, particularly important here since this is the hardest project that I've worked on yet.

3. **Compatibility with Game Mechanics**:

Chinese Checkers involves movement in multiple directions and capturing strategies. A 2d array can easily accommodate these mechanics, allowing for simple calculation of valid moves, jumps, and captures.

4. **Performance**:

For the scale of a typical board game like Chinese Checkers, a 2D array is efficient in terms of memory and performance (more on that later). More complex data structures might offer marginal benefits at the cost of increased complexity and overhead.

5. **Visualization and debugging**:

A 2D array directly mirrors the physical layout of a grid-based board game. Each cell in the array corresponds to a specific position on the game board, making it intuitive to translate game logic into code. It's relatively easy to visualize the game state and debug issues when using a 2D array, as you can print out the board's state in a format that closely resembles the actual game board

So the selection of a 2D array achieves a compromise between simplicity of use, effectiveness of performance, and easy mapping to the logical and physical needs of the game. It's a sensible decision that fits in nicely with the requirements shared by most grid-based board games.

2.1 Time complexity

What is the time complexity of

1. **Finding a piece on the board?**

Lets start by understanding what time complexity is, **"O"** stands for "Order of", which describes the upper bound of the time complexity. **n^2** indicates that the time taken grows quadratically with the increase in n. This means that as n increases, the time taken will grow quadratically. (Geeksforgeeks, 2024b)

Given that the board is represented as a two-dimensional array, the depends on the approach used to search for the piece.
When we know the exact coordinates (row and column) of the piece we're looking for, accessing it is a constant-time operation, denoted as O(1). This is one of the key advantages of using a 2D array; we can directly access any element if you know its index.
For example, if we want to find a piece at position (x, y), we can directly access it using board[x][y] (assuming zero-based indexing) (Stringfestanalytics, 2021).
Also : In case we don't know the direct coordinates and need to find a piece based on some criteria (like the first piece of a certain color), we would have to search through the array. This scenario is more complex and involves iterating over each element in the 2D array until you find the piece.
For a 2D array of size n×n:

- In the worst case, the program might have to look at every single cell.
- This results in a time complexity of $O(n^2)$, where n is the dimension of the board (e.g., for this 17x17 board, n = 17).

In this case, it's performing a linear search over a 2D structure, which inherently takes more time as the size of the structure (the game board, in this case) increases.

- Direct Access by Coordinates: O(1) - Constant time.
- Searching Without Known Coordinates: O(n^2) - Quadratic time, where n is the size of one dimension of the board.

These complexities are typical for operations on 2D arrays and reflect the trade-offs in choosing this data structure. Direct access is highly efficient, but searching can be less so, especially IF the size of the array increases.

2. **Finding all the pieces of a given colour on the board?**

This operation involves scanning each position on the board to check if it contains a piece of the specified color. This process is akin to a linear search across all elements of a 2D array.

Here's how the time complexity for this operation breaks down:

- The board is a 2D array with dimensions n×n (where n is the size of one dimension of the board).
- To find all pieces of a given color, the program needs to inspect each cell of this array once.
- This means it will perform n×n inspections, where n×n represents the total number of cells on the board.

Therefore, the time complexity of finding all pieces of a given color is O(n^2), where n is the dimension of the board.

Also, if the board were to double in size, it would take four times longer to locate every piece of a particular hue. This quadratic time complexity is typical of operations where every component in a two-dimensional structure needs to be examined. It's a simple method and usually fine for board games such as Chinese Checkers, when the board size (n) is not too big.

3. **Moving a piece on the board?**

The time complexity of moving a piece on a board in a game like Chinese Checkers, where the board is represented as a two-dimensional array, is O(1), which is constant time. This efficiency kind of stems from the fact that moving a piece involves a few direct access operations to the array:

1. Accessing the Piece's current Position: Directly retrieving the piece from its known coordinates on the board (x, y).
2. Placing the Piece at the new Position: Directly updating the new position with the piece's information.
3. Updating the old position: Marking the piece's previous position as empty or updating it accordingly.

Because they each involve immediately accessing or altering a single element in the array rather than having to loop over or process several elements, these steps are all constant-time operations. Because of this, moving a piece is generally very efficient, regardless of the size of the board.

2.2 Space complexity

Size Calculation:

This is handled by the BoardModel.cs script, which acts as the "model" in the Model-View-Controller (MVC) architecture. In this architecture, the model is in charge of the application's rules and data, which, in the instance of a board game, comprise the logic and representation of the game board.

To assess the space complexity for storing one board state we need to consider all components that contribute to the memory usage of a single state. This includes not only the board itself but also any additional data that represents the complete state of the game. Here's my breakdown:

1. Board Representation:

Board Size: As established, the board is a 17x17 grid.

Storage per cell: Each cell holds a Piece object or a reference. If we assume a Piece reference or object takes n bytes, then the board's total storage is 17 * 17 * n bytes.

2. Game State Information:

Player Data: Information about the players, like their current pieces, scores, or turn status.

Game Status: This might include flags or variables indicating the state of the game (e.g., in progress, finished), which typically require minimal space.

Space Complexity Calculation:

If we assume:

- Each Piece reference or object is n bytes.
- Player data and game status collectively take x bytes.
- Overhead for the array and objects is y bytes. Object overhead as I understand it is hidden memory cost that involves metadata about arrays and objects themselves and it's memory cost differs from programming languages and / or system architecture (32bit/64bit) (Hunder, 2009)

The total space complexity for one board state should be approximately 17 * 17 * n + x + y bytes.

Example

If a Piece reference is 4 bytes, player data and game status are 100 bytes, and overheads are 50 bytes, the calculation would be:17 * 17 * 4 + 100 + 50 = 1,156 + 150 = 1,306 bytes.

This is a simplified estimate, and the actual space required could vary based on the the data structures used for player data, and the programming language's memory management. The space complexity for storing one board state in this game is the sum of the memory required for the board, the additional game state information, and any overheads.

3 Scoring function

- **What is the underlying principle of your scoring function?**

It takes a thorough understanding of multiple scripts, including BoardModel.cs, State.cs, and IMiniMax.cs, to examine the scoring function in this Chinese Checkers game project. Each of these scripts has an impact on how the game determines the score and assesses the current state of play.

The win conditions of the game are clearly the main emphasis of BoardModel.cs. In order to ascertain whether a player has attained a winning state and where the winning positions are on the board, the script incorporates methods like winCondition and WinningPositions. These techniques are essential to comprehending how the game assesses the placement of the pieces in relation to winning. This script focuses on win conditions and determining winning positions.

`Line 325: winCondition(board[endPos.x, endPos.y])`: Indicates a method that checks if a win condition is met at a certain board position.

`Line 355: if (winCondition(player.Value()))`: Shows the game checks for a win condition based on the player's move.

`Line: 487 public bool winCondition(Piece playerColourPiece)`: Suggests the method evaluates win conditions based on a player's pieces.

State.cs, a crucial script for the Minimax algorithm that the game uses. It has a Score system to evaluate the possible moves. This scoring method takes into account the pieces' strategic positions, taking into account their locations in relation to the spawn and winning positions as well as the distances between the pieces and their goal places. The strategic depth of the game is reflected in the adjustments made to the scoring based on the advantage of a position or move.

Central to the scoring in the context of the Minimax algorithm:

`Line 72:  public int Score(IPlayer player, IState state)` - This line defines a method in the State class for calculating a score for a given state and player. It's essential for evaluating the desirability of a game state within the Minimax algorithm

IMiniMax.cs which describes the scoring system that the Minimax algorithm employs to make decisions. With the goal of optimizing the player's position, the algorithm uses the Score method to assess the desirability of different game states. This entails lowering the opponent's score while increasing the player's. Using a recursive process, the algorithm chooses moves by considering the estimated advantages and the board's present condition (more on that later).

`Line 12: int Score(IPlayer player, IState state);` - This line declares the scoring function used by the Minimax algorithm to evaluate game states.

`Line 36: if (depth == 0 || state.Score(otherPlayer, state) == Int32.MaxValue || state.Score(otherPlayer, state) == Int32.MinValue)` - This conditional statement within the Minimax algorithm uses the scoring function to decide on game moves, showing its importance in AI decision-making.

When these scripts are combined, they show that the fundamental idea behind the scoring function in this project is the strategic movement of pieces to the other side of the board and the assessment of the board's current state to identify favorable positions. This is where the Minimax algorithm comes into play. It makes judgments based on the scores, evaluating both offensive and defensive plays in an effort to increase the player's chances of winning. The objectives and strategies common to Chinese Checkers are well-aligned with this integration of strategic board evaluation and AI decision-making, which allows the AI to make move selections based on a quantitative appraisal of the game state.

- **What is the time complexity of computing the score for one board state?**

The 'Score' method in State.cs is key here and it kind of starts by creating a list to store the positions of the current player's pieces on the board. It then casts the state parameter to the State class for accessing specific properties and methods. The score is initially set to zero. The method implements a double loop that iterates through each cell of the game board, which is a 17x17 grid, resulting in 289 iterations. Within each iteration, it checks if the cell contains a piece that belongs to the current player. If it does, the position of that piece is added to the list of current positions.

- The method starts by creating a list to store positions of the current player's pieces on the board `(List<Position> currentPos = new List<Position>();)`.
- It casts the state parameter to the State class to access its properties and methods `(State nextState = (State)state;)`.
- The score is initialized to `0 (int score = 0;)`.
- The method then loops through each cell of the game board (`for (int x = Utility.xMin; x <= Utility.xMax; x++) and a nested loop for y`).
- Inside the loop, it checks if a cell contains a piece belonging to the current player `(if (nextState.takeBoard[x, y] == player.Value()))`.
- If so, the position of that piece is added to the '`currentPos`' list.

The time complexity for this method is $O(n^2)$, where n is the dimension of the board, primarily because it needs to examine each position on the board. The operations within each iteration, like checking a condition and adding positions to a list are kind of constant-time operations.

4 Minimax computation

First I would like to go over generally what the IMiniMax script does and how it's constructed. Its goal is to provide an in-depth view of the Minimax algorithm used in the Chinese Checkers game project.

Namespace and Interfaces: The script starts by defining a namespace Minimax and includes interfaces like IState and IPlayer. The IState interface has methods like Expand, which generates a list of reachable states from the current state, and Score, which calculates the heuristic score for a player in the current state

MiniMax Algorithm implementation: The core of the script is the MiniMax class, which implements the Minimax algorithm's logic to select the best move. The Select method within this class is a recursive function that evaluates potential game states and decides on the move that maximizes the AI player's score while minimizing the opponent's score. It considers different game states at a specified depth and returns the state with the best move based on the evaluation function Score for the given player.

Decision making process: The algorithm involves checking if the search depth is reached or if the state is a terminal state (either a win or a loss). It then expands the current state to find all possible moves the player can make. The Select method iterates through these child states, comparing their scores to find the state with the optimal move. The script incorporates a logic that kind of switches between maximizing and minimizing states according to the player, which is in line with the Minimax strategy of optimizing the AI's position while taking the best possible moves of the opponent into account. Now let’s discuss specifics.

- **What is the branching factor of the minimax computation?**

If the initial value of the difficulty parameter is set to 2 (in BoardModel line 54) in this Chinese Checkers game project, and this value is used as the depth for the Minimax algorithm's computation, then the depth for AI moves computation is 2. This means the algorithm will look two moves ahead in the game.
Here's how the depth works in the context of the Minimax algorithm:

1. Depth Level 0: This is the current game state, where no additional moves have been made.

2. Depth Level 1: This includes all possible states that can be reached from the current state with one move by the AI.
3. Depth Level 2: This encompasses states that can be reached from those in depth level 1 with one additional move by the opponent.

When the depth is set to 2, the AI will assess every move it can make (first level of depth) and every possible reaction the opponent can make (second level of depth) to each of those actions. This gives the AI the ability to think ahead and plan its moves in addition to the opponent's possible answers, making games more strategic and planned.

- **To what depth does it make sense for the minimax function to recurse? Why?**

A crucial factor in figuring out the ideal depth for the Minimax function in a game such as Chinese Checkers is striking a balance between the AI's intricacy and feasible computing. An algorithm with deeper recursion makes the AI more strategic by enabling it to plan further ahead. This does, however, come with higher processing requirements. Speaking from playing extensively against these bots, this range offers a demanding AI without unduly straining processing power. The speed of the game and the intended degree of difficulty can also affect the precise depth; some games dynamically change the depth to accommodate various game conditions or user preferences.

Practically speaking, this degree of complexity makes the AI somewhat difficult since it allows it to anticipate its next move and take into account both its own and its opponent's possible countermoves. This degree of depth strikes a compromise between strategic depth and computing complexity, enabling the AI to make competitive decisions without becoming unduly complicated.

- **What are the storage requirements of the minimax computation?**

The storage requirements for the Minimax computation in a game, particularly in a complex board game like Chinese Checkers, are primarily influenced by the size of

the game tree that the algorithm generates. This tree expands exponentially based on the branching factor (the average number of moves possible from any given board state) and the depth of the recursion. To narrow it down a bit lets set some boundaries and assume the game is played with initial depth set to 2 and that player is playing against two AI bots. It entails knowing how the game tree grows and how much storage each game state requires. The Minimax algorithm first assesses every conceivable move for the initial AI player at a depth of two. Then, it takes into account every conceivable reaction from the second AI player for each of these moves.

The average number of moves possible from any given game state is known as the branching factor, and it is the crucial aspect in this case (Pearl, 2022). Because each piece in Chinese Checkers has several ways to move, this number can go rather high. The number of game states stored increases exponentially with depth. Initially, the AI's movements are represented by the branching factor 'n' at level 1. At the second level, there are 'b' reactions from the second AI for every one of those 'n' movements, resulting in 'n^2' game states at this layer. The total number of states stored by the algorithm at a depth of 2 would be 'n + n^2'. The way the board and pieces are represented determines how much storage is needed for each state. The storage for each state in a simplified model, with a 17x17 grid for the board, would be determined by the information kept in each cell, such as piece existence and who owns it (which color it is).

To estimate the total storage requirement, if each state needs a 's' amount of storage, then the total storage for depth 2 Minimax computation would be approximately '(n + n^2) * s' bytes. The actual numbers for 'n' and 's' rely on how the game is implemented specifically and on the move rules. This estimate explains why the Minimax algorithm can be memory-intensive, particularly in games with high branching factors like Chinese Checkers, and emphasizes the value of optimization strategies like alpha-beta pruning in real-world applications. The use of alpha-beta pruning can greatly improve the performance of the Minimax algorithm (Aradhya, 2024)

References

Stingfestanalytics, 2021. *Zero-based indexing: What it is and when you've seen it before.* [online], Available at: <https://stringfestanalytics.com/seen-zero-based-indexing/ > [Accessed 2 Januari 2024].

Pearl, J., 2022. The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality. Probabilistic and Causal Inference: The Works of Judea Pearl (1st ed.). Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/3501714.3501724

Aradhya, A., L., 2024. *Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning).* [online], Available at: < https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/> [Accessed 4 Januari 2024].

Hunter. A., 2009. *Object Overhead: The Hidden .NET Memory Allocation Cost.* [online], Available at: < https://www.red-gate.com/simple-talk/development/dotnet-development/object-overhead-the-hidden-net-memory-allocation-cost/> [Accessed 18 December 2023].

Geegsforgeeks, 2024a. *Graph Data Structure And Algorithms* [online], Available at:< https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/> [Accessed 29 December 2023].

Geegsforgeeks, 2024b. *Graph Data Structure And Algorithms.* [online], Available at:< https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/> [Accessed 21 November 2022].

Dahlin. A., 2018. *Two-dimensional array + canvas = board games!* [online], Available at:< https://medium.com/@axeldahlin/two-dimensional-array-canvas-board-games-fd417e92953>[Accessed 14 December 2022].