Nguyen Viet Hoang

BOARD GAME AI AND MINIMAX ALGORITHM

Bachelor's thesis

Information Technology

Bachelor of Engineering

2022

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Nguyen Viet Hoang
Thesis title	Board game AI and minimax algorithm
Commissioned by	-
Time	May 2022
Pages	42 pages
Supervisor	Timo Mynttinen

ABSTRACT

The objective of this thesis was to develop a turn-based two-player board game called Dobutsu Shogi (Let's catch the lion) and implement a simple AI using minimax algorithm with alpha-beta pruning optimization techinique. The game was a single-page web application built using web development technologies: ReactJS and Redux with Typescript. It provided two play modes: two users against each other and a user against a computer.

In the theoretical part, some of the most basic concepts of game theory were introduced and followed by the explanation of minimax algorithm with alpha-beta pruning variation. Furthermore, in the implementation part, the process of making the game and programming its AI was described in detail.

In the end, the goal to make Dobutsu Shogi board game and develop its simple AI was successful. Minimax algorithm implementation worked well, and its optimization technique alpha-beta pruning showed its effectiveness in terms of performance. However, the AI was using a quite poor static evaluation function, and as a result, users can still defeat it if they have a certain amount of experiece about the game.

Keywords: board game, algorithm, minimax, alpha-beta pruning, react-redux

CONTENTS

1 INTRODUCTION

In this computer era, the use of AI to improve product quality as well as people's daily lives has become increasingly popular. As a result, its applications can be seen in almost every field, including game development. It is undeniable that this field has been a promising area in the application of AI since the very beginning. While video games are gradually becoming the first choice for presenting and testing different algorithms recently, board games have been the central topic for AI researchers from the start (Yannakakis & Togelius 2018, Preface ix).

One of the earliest applications of gaming AI was being opponents in some common board games like Tic-tac-toe, Chess, Checkers, or Go. These games have been researched deeply and as a result, not only the quality of AI used for them is improved, but also programs that capable of defeating world's top players have been created one after another. In order to implement AI for board games, different algorithms and optimization ways have been applied. Until now, minimax algorithm along with alpha-beta pruning was one of the most popular techniques used in this area.

The goal of my thesis is to develop a single-page web application as a simulated version of a turn-based two-player board game called Dobutsu Shogi (Let's catch the lion) and applying minimax algorithm with alpha-beta pruning optimization technique to implement a simple AI for the game. The game will be built using modern web development technologies such as ReactJS, Typescript, and Redux.

Throughout my thesis, readers will be introduced with the most basic concepts of game theory, minimax algorithm, alpha-beta pruning variation, and the process of programming a board game with its simple AI. In theoretical part, there will be illustrative figures with pseudo-code that help explaining the algorithm. In implementation part, beside explanation texts, not only screenshots of used code but also logic diagrams for different actions will be provided in order to describe the process better. Additionally, I also applied and tested different variations of the algorithm to show the improvement in term of effectiveness. In the end, I

hope that readers will be able to create an AI system for a board game themselves using introduced knowledge.

2 THEORETICAL BACKGROUND

This section covers theoretical background related to using minimax algorithm to program an AI system for a board game. The topics include basic concepts of game theory, minimax algorithm, and alpha-beta pruning.

2.1 Game theory

Game theory is a mathematical framework for strategic interactions between participants where each of them is assumed to always aim for the optimal result based on given rules and information. In gaming industry, while its applications are rarely found in real-time games, game theory is considered as the source for the terminology that is used in turn-based games (Millington & Funge 2009, 668). This subsection does not introduce all concepts about game theory but only enough to understand how it can be used to implement an AI system in a turn-based game.

2.1.1 Types of games

Based on different characteristics of the game, game theory classifies it into different forms: zero-sum and non-zero-sum game based on the game objective, perfect and imperfect information game based on the information about the game that each player has. The board game project that will be implemented later is considered as a two-player zero-sum game with perfect information.

Zero-sum and non-zero-sum game

In a zero-sum game, the sum of all losses by a player or group of players is equal to the sum of all gains for every possible outcome of that. Games such as Chess, Checkers, or Tic-tac-toe are simple examples of two-player zero-sum games since in the end, there is one winner and one loser. If you score 1 point for winning, then it would be equivalent to score -1 for losing. It is also possible for

ties in this type of game, because then neither player gains nor loses, and the sum is still equal zero. Another good example for zero-sum game is Poker. After all, the combined amount of winning money is balance to the combined amount of losing money. (Non-Zero-Sum Games vs. Zero Sum Games… 2010.)

In contrast to zero-sum games, there are games where all participants could all win or lose. These games are referred as non-zero-sum games. A classic example for this type of game is the Prisoners' Dilemma. In this example, two prisoners are held in different cells and cannot communicate with each other. They are both offered a bargain. If both confess, they will be convicted for three years each. If both stay silent, they will only serve one year in prison. However, if one confesses while the other does not, the one who confessed will be free, while the other will be convicted for five years. It is important to note that the prisoners do not know if the other confessed or not. It is clear that the option in which both prisoners remain silent will lead to the optimal payoff, however, it is not a rational option since they all behave in their self-interest. Therefore, the most rational decision here for both is to confess, resulting in their losses.

Perfect information and imperfect information game

Perfect information games refer to games where every player have all information about the game state at any time. They are aware of every possible action and its effect. Examples for this type of game are Chess, Go, or Tic-tac-toe. Although you may not know what move the other will play, but you will be able to determine all moves that your opponent could possibly make and its result at any given time. Meanwhile, in card games like Poker, there is information that you do not know, such as your opponents' cards. This type of game is called imperfect information game.

It is often easier to work with perfect information games since all game state data is visible all the time. Different algorithms and techniques can be adapted for imperfect information games too. However, because of more elements need to

be considered, they often lead to a worse performance. (Millington & Funge 2009, 669.)

2.1.2 The game tree

A game tree is a graph where its nodes represent game states, and its edges represent player actions to get from one game state to another. The number of tree branches from a node is the number of possible actions that player can make at that given game state, and the root of the tree is the current game state when the tree is started to be built. (Yannakakis & Togelius 2018, 39.)

Figure 1 shows a part of a Tic-tac-toe game tree.

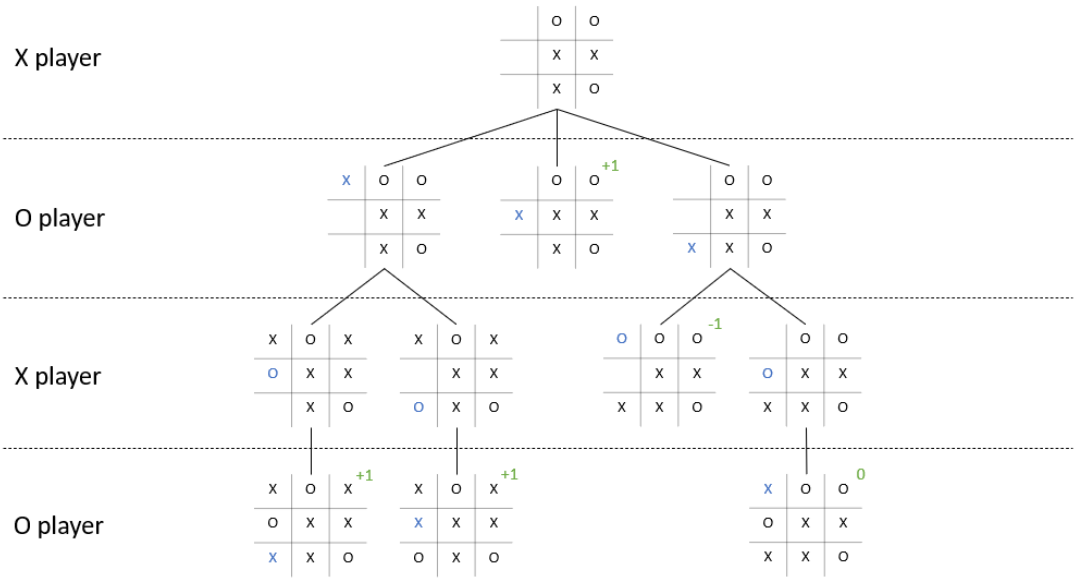


Figure 1. Tic-tac-toe game tree

In a game tree, each node describes a different game board position, and from that board position, each branch represents a player's possible move. Since this is a turn-based game, each player takes turn to make a move. After a move is made, the board position is changed. A game tree can be continued until the board position cannot provide any other possible moves. These positions are referred to terminal positions, in which the game is over. In terminal positions, a

point is assigned to each player. For Tic-tac-toe game, it is quite simple as +1 for the winner, -1 for the loser, and 0 if it is a tie.

A game tree's depth is the maximum number of turns in the game. Game tree in different games has different size of branches and depth. In Tic-tac-toe, it is possible to construct a complete game tree that include every possible outcome since the board only contains nine squares, equal to a maximum of nine turns. However, in Chess, there can be an infinite number of turns (the 50-move rule can limit this). Consequently, the depth can be infinity and it is impossible to build the whole game tree, although the numbers of branches is relatively small. (Millington & Funge 2009, 670.) For this type of game, the program often examines only a small part of the whole tree and assigns points based on the game state.

2.2 Minimax algorithm

In adversarial games like Chess, Checker, Go, and Tic-tac-toe, a basic adversarial search algorithm called minimax is used to find the optimal action for the player while assuming the opponent also makes the optimal decision. This algorithm has been applied successfully in different two-player perfect-information zero-sum games.

Minimax algorithm is a recursive algorithm that uses recursion to go through the game tree. In this algorithm, there will be two players participating in the game, one is called maximizing player, and the other is called minimizing player. Each player will always take action that leads to the best outcome for them. Since this is applied in zero-sum game where a player's loss is the other's gain, maximizing player will select action with maximum value, while minimizing player will do completely opposite, select action with minimum value.

As mentioned before, the complete game tree is not always able to be built. It is possible to be constructed in Tic-tac-toe since there is only a maximum of nine turns, but impossible in games like Chess or Go. Therefore, in most of the minimax applications, the search is stopped after looking ahead a certain number

of moves to only construct a part of the game tree. When the search is stopped, terminal nodes are also formed. The algorithm will first proceed down to these terminal positions and use static evaluation function to evaluate them. In Chess, a simple evaluation function can be the sum of total value of black pieces currently on board subtracted by the sum of total value of white pieces currently on board. The higher this result is, the better situation is for the white side. In order to construct high quality AI, much more complex functions are needed to evaluate the board state. After getting the evaluation for terminal nodes, the algorithm traverses up the game tree, determines the value that current player (minimizing or maximizing) at that given state would select assuming they play optimally. (Yannakakis & Togelius 2018, 43.) In order to describe the algorithm better in programming, Figure 2 uses pseudo-code to show a way to implement minimax algorithm.

```
function minimax(node, depth, isMaximize)
    if depth == 0 or game over // check for terminal nodes
        return static evaluation of node

    if isMaximize // for maximizing player
        maxEval = -infinity
        for each child of node
            currentEval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, currentEval) // check for updating max value
        return maxEval

    else // for minimizing player
        minEval = +infinity
        for each child of node
            currentEval = minimax(child, depth - 1, true)
            minEval = min(minEval, currentEval) // check for updating min value
        return minEval

// initial call
minimax(currentNode, 3, true)
```

Figure 2. Pseudo-code for minimax algorithm

In Figure 2, node is the current examining board state, depth is how many moves ahead that going to be searched, and isMaximize is a Boolean variable represents current player. First, the function checks if the current node is a terminal node to return its evaluation. Otherwise, current player will be determined. If it is maximizing player, all new game states that can be formed from current one will be explored to find the highest evaluation that can be obtained (maxEval). In order to return the highest evaluation between all child nodes, a recursive call to the minimax function is made, using a child node,

current depth minus one, and the new current player (the opponent – minimizing player) as the arguments. If current player is minimizing player, instead of finding the maximum evaluation, the function will search for the lowest evaluation of all child positions (minEval). Additionally, when looking at a new game state from minimizing player's perspective, it is important to remember to call recursive function passing isMaximize as true to represent the new current player (the opponent – maximizing player).

Figure 3 illustrates how minimax algorithm works in a specific game tree with scores assigned to terminal nodes as the results of static evaluation function. The red path describes the best moves for each player in each position.

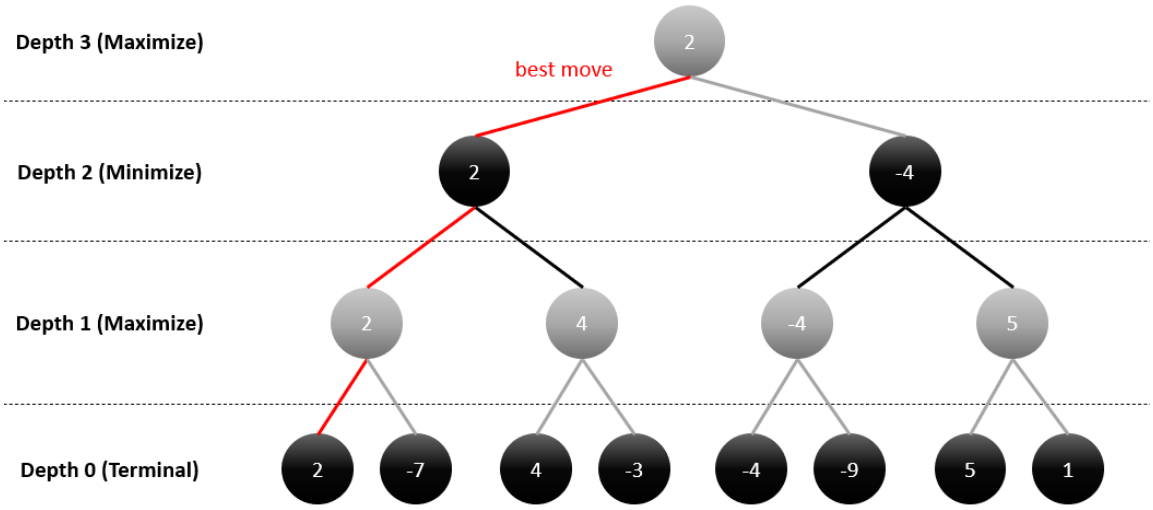


Figure 3. How minimax algorithm works

The higher the depth is used in the initial call, the better the returned move is. However, the number of tree node that need to be explored are exponential in depth of the tree. As a result, in order to finish running the algorithm in a certain amount of time, depth number cannot be too high, and the computing power also need to be taken into account.

2.3 Alpha-beta pruning

Alpha-beta pruning is a modified version of minimax algorithm, which can also be seen as an optimization technique. By using this variation, some tree leaves, or

even entire branches are able to be removed without affecting the result. In other words, without checking every game state, the algorithm can still find the optimal move that the original minimax will return with less time needed.

In order to find leaves and branches that are not affected the result for pruning, two threshold parameters called alpha and beta are used; hence, the name alpha-beta pruning. Alpha represents the highest value that the algorithm has explored so far at any point along the path of maximizing player, while beta represents the lowest value has been found so far at any point along the path of minimizing player. When beta is equal or lower than alpha, the prune will occur. (Alpha-Beta Pruning, 2022.) Figure 4 uses pseudo-code to show a way to implement this variation of the algorithm.

```
function minimax(node, depth, alpha, beta, isMaximize)
    if depth == 0 or game over // check for terminal nodes
        return static evaluation of node

    if isMaximize // for maximizing player
        maxEval = -infinity
        for each child of node
            currentEval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, currentEval) // check for updating max value
            alpha = max(alpha, currentEval) // check for updating alpha value
            if beta <= alpha // check for pruning
                break
        return maxEval

    else // for minimizing player
        minEval = +infinity
        for each child of node
            currentEval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, currentEval) // check for updating min value
            beta = min(beta, currentEval) // check for updating beta value
            if beta <= alpha // check for pruning
                break
        return minEval

// initial call
minimax(currentNode, 3, -infinity, +infinity, true)
```

Figure 4. Pseudo-code for alpha-beta pruning

Compared to the codes in Figure 2, in Figure 4, the minimax function required two new parameters: alpha and beta. They are checked to update after the evaluation of a child node is finished and followed by another check for pruning. The value for alpha can only be updated if current player is maximizing player, while the value for beta can only be updated if current player is minimizing player. It is also important to note that alpha and beta values are not passed to upper

nodes while backtracking the game tree, but only passed to the child nodes. Since the aim of these values is to keep track of the highest and lowest explored value along the path of maximizing and minimizing player, in the initial call, their initial values are minus infinity and infinity accordingly.

Figure 5 describes the same game tree used in Figure 3 but with alpha-beta pruning variation applied. Branches with double red crosses represent pruned branches, and nodes without value on them represent have not explored nodes. It can be observed that the returned optimal move path is still the same even without exploring all game states.

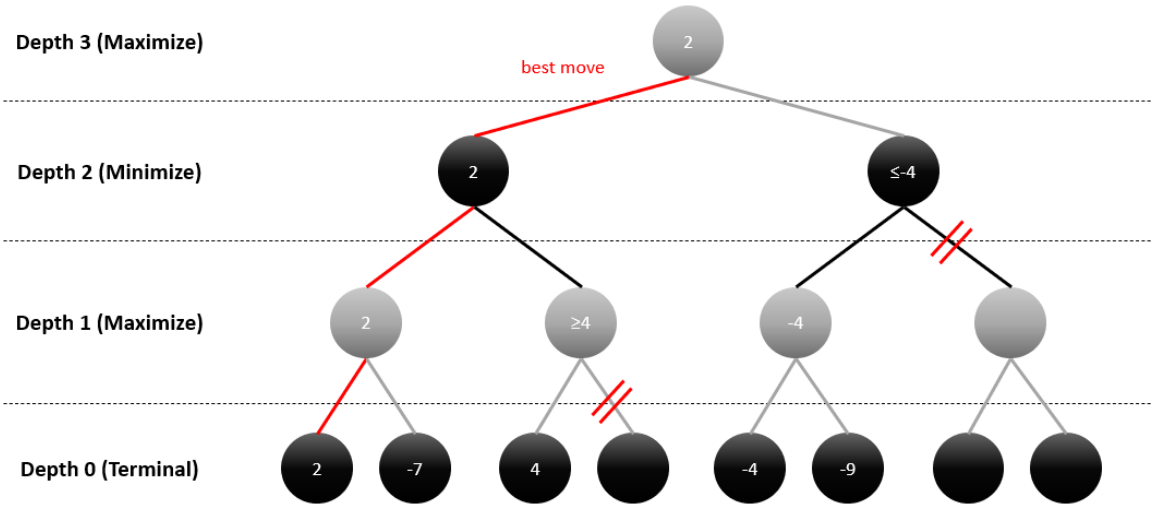


Figure 5. How alpha-beta pruning works

It is important to note that in alpha-beta pruning, the prune is not always guaranteed to occur. Figure 6 describes the game tree in Figure 2 with some terminal positions are swapped. This game tree is also applied with alpha-beta pruning, however, no node or branch is eliminated.

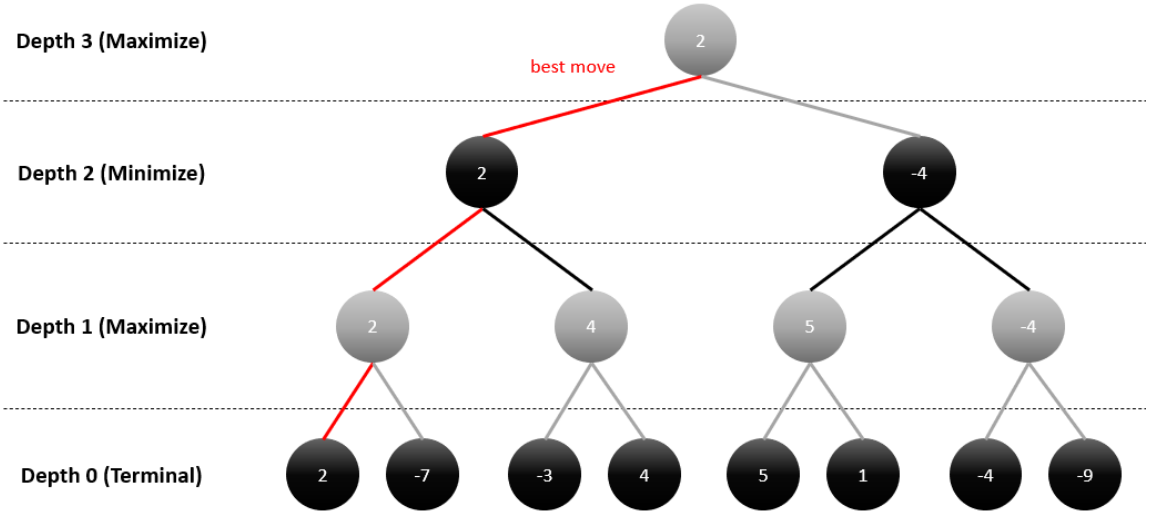


Figure 6. Alpha-beta pruning applied to game tree with bad move order

Move order is an essential factor affecting alpha-beta pruning's performance. The numbers of branch that will be pruned from the tree is highly dependent on the order of the moves. In the worst case, no branch is eliminated, and the algorithm works exactly like in minimax algorithm. It is even worse since the function also takes time to check for pruning, which will not happen. In an ideal case, the moves are ordered from best to worst for the current player at that depth. Domain knowledge is usually used to determine which order is likely to be good, for example, in Chess, captured moves are often examined first. (Alpha-Beta Pruning, 2022.)

3 PROJECT IMPLEMENTATION

In this section, the process of making a two-player zero-sum board game with perfect information called Dobutsu Shogi followed by the implementation of its simple AI is showed in detail. The AI programing process illustrates how minimax algorithm with its alpha-beta pruning variation is applied. This thesis project includes two main play modes: two users against each other to describe the basic functions of the game, and a user against a computer to test the AI system.

3.1 Dobutsu Shogi game

Dobutsu Shogi (“Let’s catch the lion!”, “Animal Chess”) is a small variant of Japanese Shogi. The game was invented by a female professional Shogi player Madoka Kitao. It is played on a 3x4 board with the general rules of standard Shogi, but with a few exceptions. (Dobutsu Shogi, 2022.) Figure 7 shows Dobutsu Shogi board with all pieces at the start of the game. The red dots in each piece illustrated different directions that piece can move, but only one square per move.

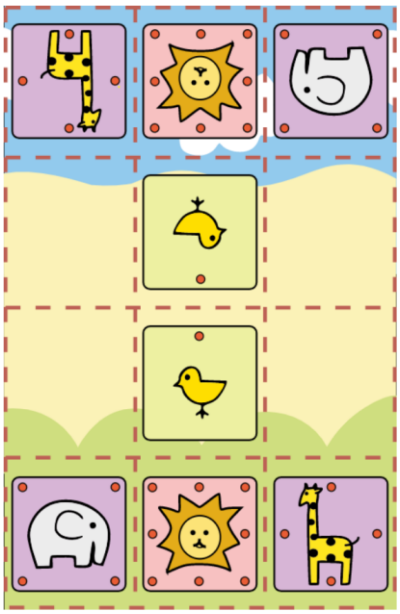


Figure 7. Dobutsu Shogi game starting position

The game rules are simple. There are two players (Sky and Forest), and each player starts the game with four pieces as in Figure 7:

- Lion is the same as King in Chess, can move one square in any direction
- Giraffe is the same as Rook in Chess, can move horizontally or vertically, but only one square
- Elephant is the same as Bishop in Chess, can move diagonally, but only one square
- Chick is the same as Pawn in Chess after its first move, can only move one square forward

Like in Shogi, captured pieces change side and are sent to the hand of player who capture it. When it is your turn, you can choose between moving a piece on board and dropping a piece in hand back to the board. If a Chick reaches the

furthest rank from its side, it is promoted to a Hen (Figure 8), and if it is captured, it is demoted back to a Chick. A Hen can move one square in any direction, except diagonally backwards. Unlike in Shogi, there are no restrictions about how a player can drop a Chick. Player can drop two Chicks in the same file, drop a Chick to checkmate, or drop a Chick in the final rank (however, by doing this, the Chick will not be promoted to a Hen and consequently, unable to move). (Dobutsu Shogi, 2022.)

Figure 8. Hen piece

In order to win the game, players need to capture their opponent's Lion, or move their own Lion to the furthest rank without being captured in the next move (stalemate is also a win). If two players play the same move three times in a row, the game will end in a draw. (Dobutsu Shogi, 2022.)

The game that is built as the thesis project does not use the original assets as in above figures in this subsection. Since the aim of the project is to illustrates the basic gameplay and its simple AI, a simple board will be used with two sides, white and black (Forest and Sky side respectively), and instead of the cartoon figures of the relevant animals, texts are used to represent different pieces.

3.2 Development technologies and tools

In order to build this project, different web technologies such as React for performing frontend operations, Redux and Redux toolkit for state management were used, along with Typescript for types check and proper documentation. For styling in the application, I found it easier for managing styles with SASS.

About tools, Visual Studio Code was used as the source code editor, while Microsoft Edge as the web browser. Furthermore, React Developer Tools and

Redux DevTools are two extensions that were used mainly for the debugging process. React Developer Tools is a powerful extension for inspecting components with their current props and states inside React tree, while Redux DevTools makes tracking the application workflow and managing state changes much easier.

3.3 Folder structure

Planning application folder structure is one of the key steps that have a huge impact on the scalability and maintainability of the project as it grows, however, it is sometimes skipped. By organizing all files in a logical order, when a project becomes bigger, its components, styles, states, and utility functions are still easy to keep track and manage. Figure 9 describes how different files and folders are arranged in this application.

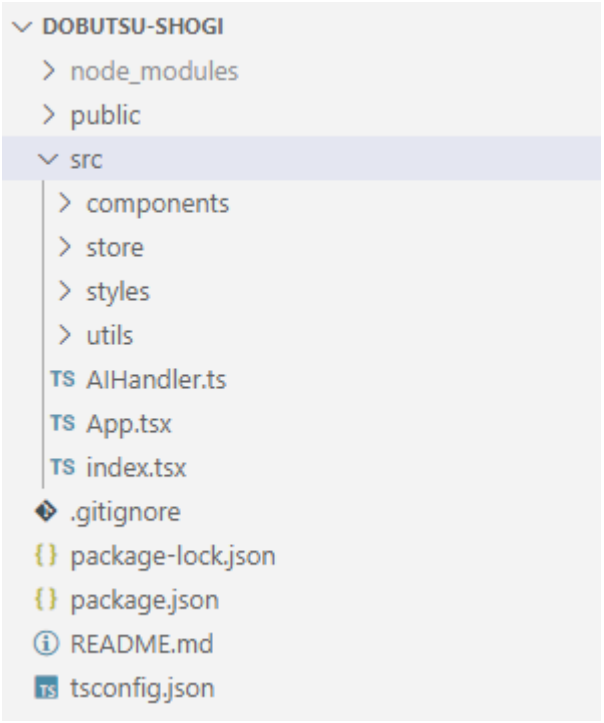


Figure 9. Project folder structure

All files outside of src folder are generated from create-react-app with Typescript template. Inside of it, components folder includes all UI components that are used in the application, while store folder contains all redux files for state management. utils folder keeps utility files for different game functions, constants, and types.

Furthermore, AIHandler.ts file is responsible for the logic implementation of the AI system, while App.tsx and index.tsx files are the main component of the React application and the root file for React accordingly. For styling, styles folder is used to keep all global styling, typography, animations, mixins, style functions and variables. To style each component, a separate SCSS file is used and kept beside the component file. The thesis project is built based on React, Typescript, and Redux. As a result, the folder structure is designed to fit this type of project.

3.4 Application layout

With React as the main development technology, the application UI is divided into smaller components containing their own properties and states. This part discusses not only how the project layout is divided, but also different features that the application provides. Figure 10 shows the inside of components folder.

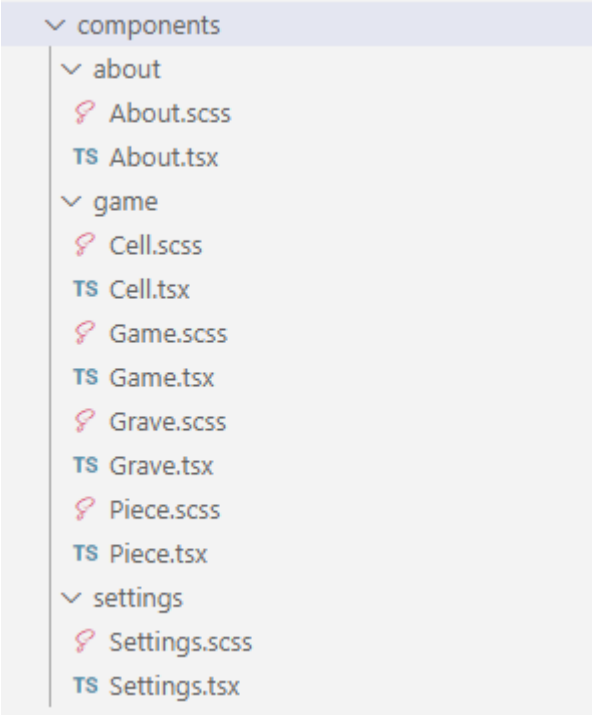


Figure 10. components folder

There are three main components: About, Game, and Settings. It can be seen from Figure 10 that Setting and About components are not divided into smaller ones to keep the application simple and since there are not too many reusable

parts here. Figure 11 describes how the application looks like with different components being rendered on a web browser.

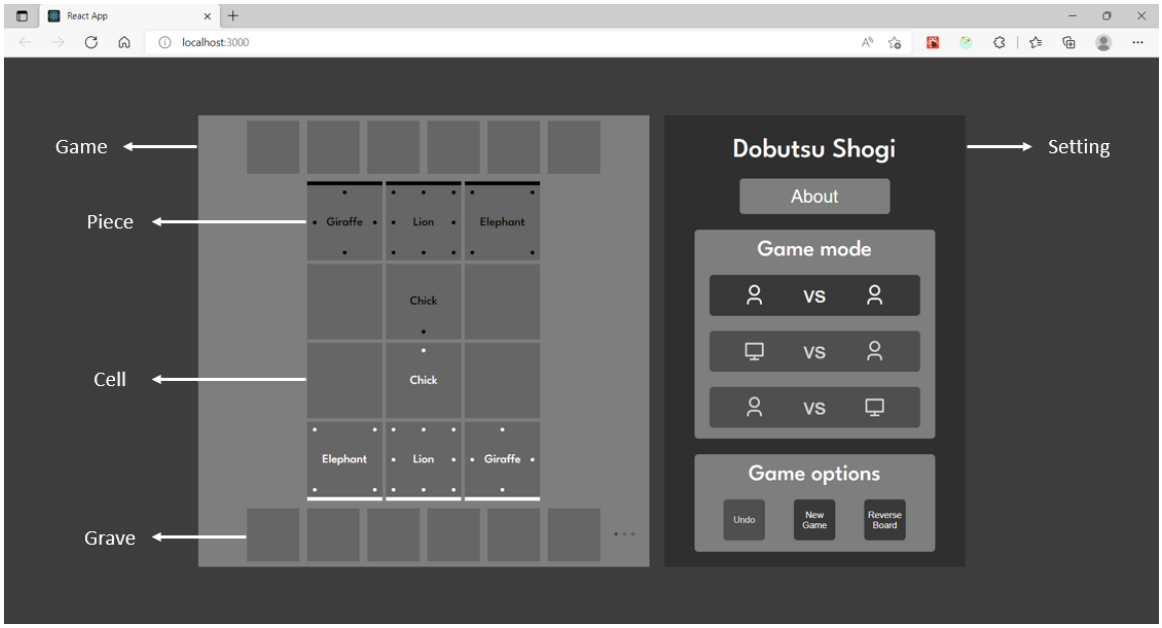


Figure 11. Application layout

Game component includes three other types of smaller components: Cell, Piece, and Grave. Each Cell contains a Piece that can only be rendered when there is a piece in that cell. Beside a 3x4 board containing 12 Cell and eight Piece components, there are also two rows of grave representing hand of two players for placing captured pieces, each row contains six Grave components.

As you can see from the setting panel, users can change between different modes. Two game modes that the application provides are two-player and computer-vs-player (an AI play as white/black against a user play as black/white side). Additionally, there are also different game options that can be selected during the game to improve users' experience. Players can undo the latest move if at least one move has been made, start a new game with the selected game mode immediately, or change to the opposite board view.

Figure 12 describes About component after being rendered. It contains the project information and basic game rules.

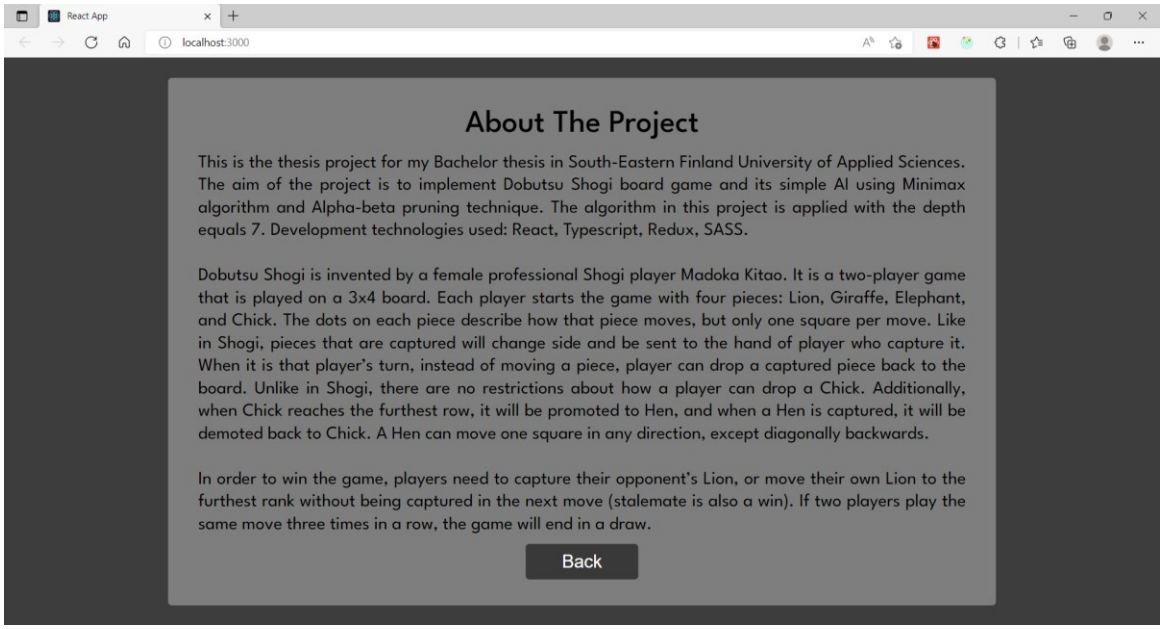


Figure 12. About component

This panel is not rendered by default. However, when About button from setting panel is clicked, it will be displayed on top of the current layout and will return to the hidden state when Back button is clicked.

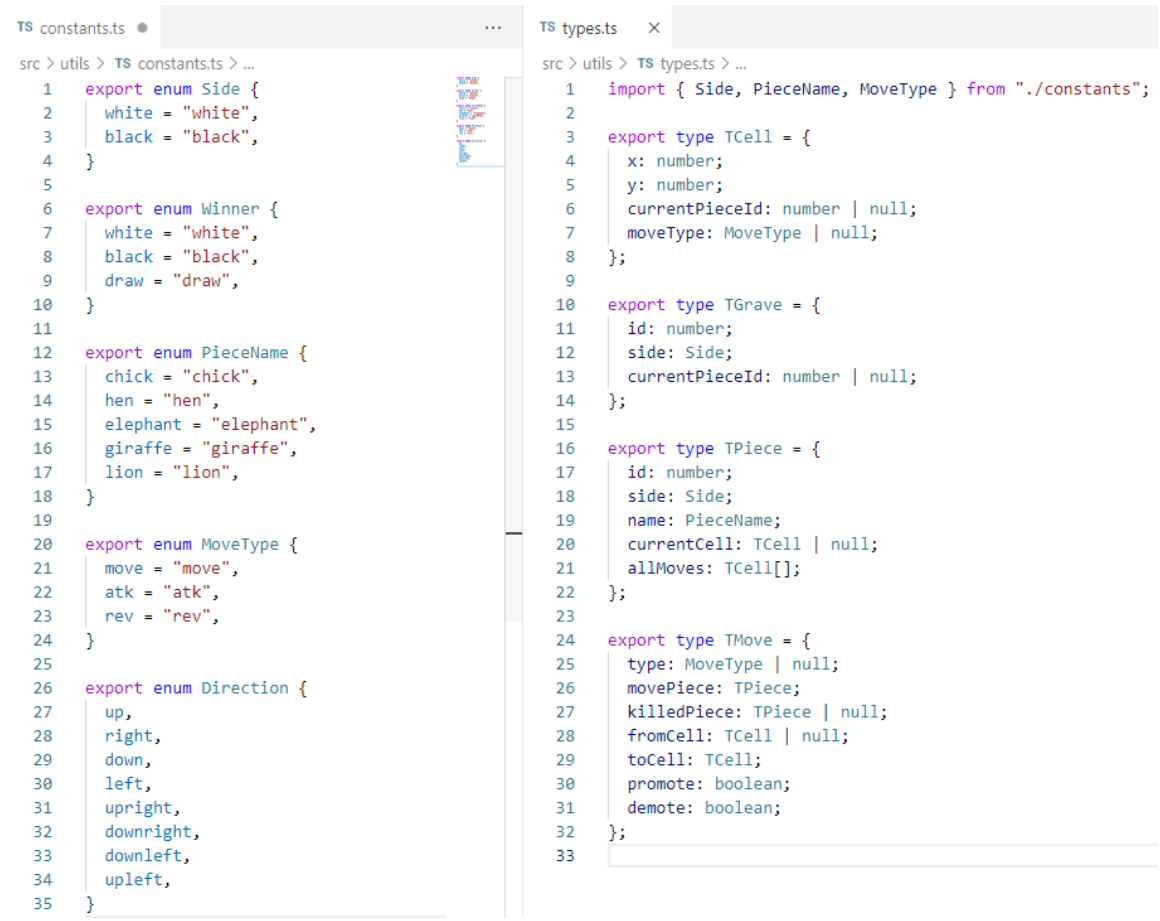
3.5 State management

In React applications, states can be managed inside components using useState hook and passed around through props. However, this way of handling state only works effectively in a small scale, not when the number of states travelling between components is huge. To simplify the process of managing and updating global states, in this application, Redux Toolkit is used to implement Redux store and logic, while React-Redux is used to connect Redux store and React components together.

Figure 13 describes the content of store folder. index.ts file creates the Redux store instance using configurationStore function from Redux Toolkit, while the other two Redux slice files are responsible for Redux logic of different features. game.ts exports gameReducer function for gameplay logics, and setting.ts exports settingReducer function for setting logics.

Figure 13. store folder

Figures 14 shows all enums (sets of named constants) in constants.ts and all types that are declared using Typescript in types.ts file.



constants.ts

```ts
export enum Side {
  white = "white",
  black = "black",
}

export enum Winner {
  white = "white",
  black = "black",
  draw = "draw",
}

export enum PieceName {
  chick = "chick",
  hen = "hen",
  elephant = "elephant",
  giraffe = "giraffe",
  lion = "lion",
}

export enum MoveType {
  move = "move",
  atk = "atk",
  rev = "rev",
}

export enum Direction {
  up,
  right,
  down,
  left,
  upright,
  downright,
  downleft,
  upleft,
}
```

types.ts

```ts
import { Side, PieceName, MoveType } from "./constants";

export type TCell = {
  x: number;
  y: number;
  currentPieceId: number | null;
  moveType: MoveType | null;
};

export type TGrave = {
  id: number;
  side: Side;
  currentPieceId: number | null;
};

export type TPiece = {
  id: number;
  side: Side;
  name: PieceName;
  currentCell: TCell | null;
  allMoves: TCell[];
};

export type TMove = {
  type: MoveType | null;
  movePiece: TPiece;
  killedPiece: TPiece | null;
  fromCell: TCell | null;
  toCell: TCell;
  promote: boolean;
  demote: boolean;
};
```

Figure 14. constants.ts and types.ts files

A TCell object represents a cell on board. This object contains its position on board, type, which is only set when this cell is one of the generated moves, and the id of current piece. The position is determined by two numbers as the coordinate of the cell on board where the origin (0, 0) is the lower left corner cell in the initial view (white side is under black side on screen). A TGrave object represents a grave slot that stores all related information such as this slot's id, side, or the id of current piece. Next, a game piece is represented by a TPiece

object which contains all information of this piece: id, side, name, the current cell (if not in hand), and a list of all reachable cells from current position. Lastly, a move that is made during the game is represented by a TMove object. This object includes its type, move and captured pieces, origin and destination cells, and two Booleans showing if this is a promote-to-Hen move or a demote-to-Chick move, or both. The type of a move will be set to null when is declared for the first time and can be modified later.

The game state in game slice keeps track of different states of the game. All states that are used to form a game state are listed below:

- All cells' current states
- All graves' current states
- All pieces' current states
- Current turn (white or black)
- Winning side (null if the game continues)
- Id of current active piece (null if there is no active piece now)
- List of all moves that are made since the game started

While game slice is responsible for handling game state, setting slice is responsible for managing current game settings. All states that are used to form a setting state are listed below:

- Current game mode index (0 is human versus human, 1 is AI as white side versus human as black side, and 2 is human as white side versus AI as black side)
- Current board view is reversed or not
- About panel is showing or not

Figures 15 describes initial game and setting states from Redux DevTools. It can be seen that allCells, allGraves, and allPieces arrays are empty currently, but later when the game starts, there will be an initialized function to generate all cells, graves, and pieces.

Figure 15. Initial game and setting states

Furthermore, by using useSelector and useDispatch hooks from react-redux, any piece of states from Redux store can be read and modified easily from any component.

3.6 Basic game functions

Before diving into developing an AI system for this game, all basic game features are implemented. This helps not only creating functions that operate a game match, but also later will be used while applying the algorithm. Figures 16 and 17 show all reducer functions that are used to update setting and game states. To implement different features, actions are dispatched from components.

```
13  const settingSlice = createSlice({
14    name: "setting",
15    initialState: initialSettingState,
16    reducers: {
17      changeGameMode(state, actions) {
18        state.gameModeIndex = actions.payload;
19      },
20      reverseBoard(state) {
21        state.reversed = !state.reversed;
22      },
23      showAboutHandler(state) {
24        state.showAbout = !state.showAbout;
25      },
26    },
27  });
```

Figure 16. settingSlice reducers

```
42  const gameSlice = createSlice({
43    name: 'game',
44    initialState: initialGameState,
45    reducers: {
46  >   initGame(state) {···
54      },
55  >   pieceOnClick(state, actions) {···
70      },
71  >   cellOnClick(state, actions) {···
134     },
135 >   undoHandler(state, actions) {···
175     },
176 >   aiHandler(state, actions) {···
245     },
246   },
247 });
```

Figure 17. gameSlice reducers

Since settingSlice reducers, which are responsible for updating setting state, are quite simple, they will not be discussed further. Additionally, this part concentrates only on the implementation of basic game functions, as a result, aiHandler reducer will be skipped for now.

3.6.1 Initializing game

In order to initialize a new game, initGame action is dispatched from React components. Figure 18 describes the reducer function for this game action. initCells, initGraves, and initPieces are three functions imported from GameUtils.ts file that are used to first empty all cells, pieces, and graves lists and then set them to their starting value.

```
46    initGame(state) {
47      initCells(state.allCells);
48      initGraves(state.allGraves);
49      initPieces(state.allPieces, state.allCells);
50      state.currentSide = Side.white;
51      state.winningSide = null;
52      state.activePieceId = null;
53      state.moveHistory = [];
54    },
```

Figure 18. Code for initGame reducer

Figure 19 illustrates the game state after initGame action is dispatched for the first time when the application run using useEffect hook with an empty dependency array inside Game component.

Figure 19. Game state after initGame is dispatched

After a new game is initialized, allCells contains 12 TCell objects, allGraves contains 12 TGrave objects, six from each side, and allPieces contains eight TPiece objects, four from each side.

3.6.2 Generating possible moves

When a user clicks on a piece, if this piece is not currently active, its possible moves need to be generated. Otherwise, all generated moves need to be deleted. This feature is implemented by dispatching pieceOnClick action from the clicked piece component after checking if the piece is currently clickable.

For the checking process, the current turn state that is extracted from the game state and the side of the clicked piece will be compared. If the side of the piece is the same as the current turn, then this piece is clickable. The current game mode id information that is extracted from the setting state is also considered in this process. For example, if the computer plays as white side, white pieces are not clickable for the whole game. Additionally, pieces are only clickable if the winner state is still null, implying the game continues.

After making sure the clicked piece is clickable, pieceOnClick action will be dispatched. Figures 20 and 21 show the logic workflow inside the reducer function and a way to implement it.

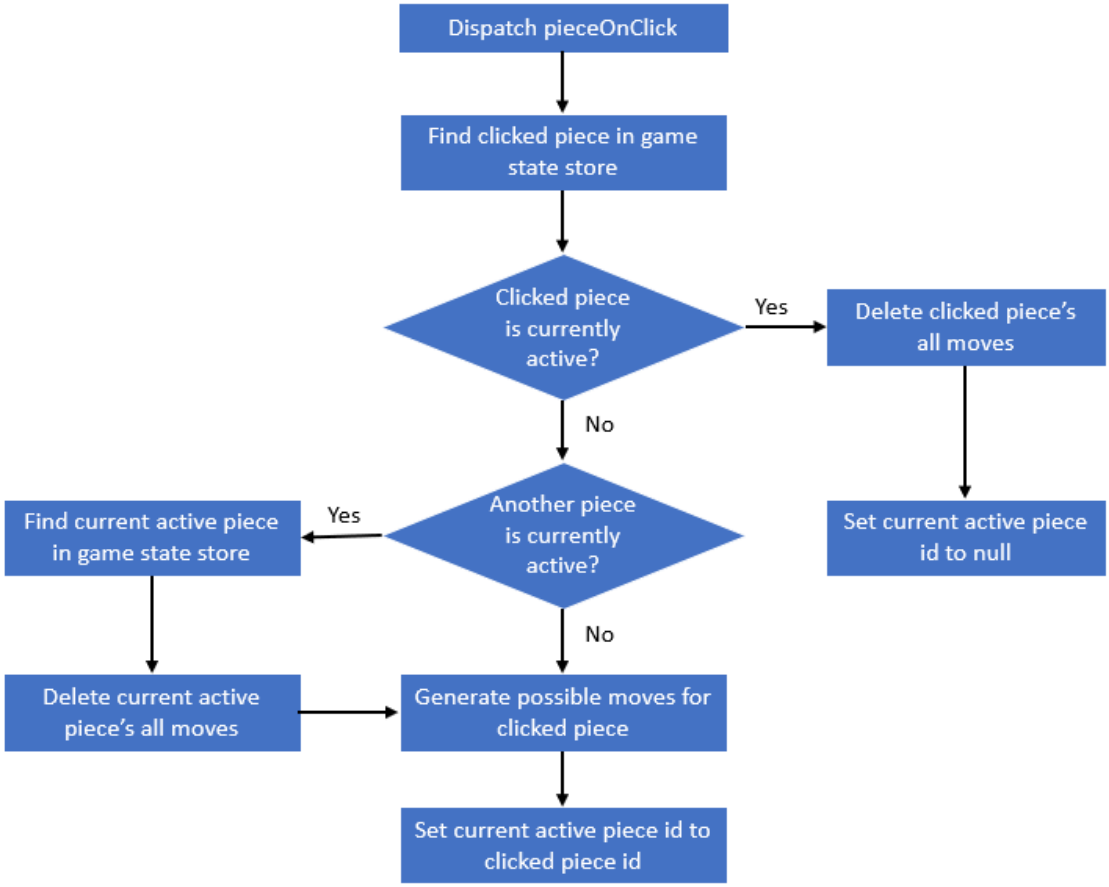


Figure 20. Logic workflow inside pieceOnClick reducer

```
55    pieceOnClick(state, actions) {
56      let piece = getPieceById(actions.payload.id, state.allPieces);
57      // if current activePiece is this piece
58      if (state.activePieceId !== null && state.activePieceId === piece.id) {
59        deleteMoves(piece, state.allCells);
60        state.activePieceId = null;
61      } else {
62        // if current activePiece is another piece
63        if (state.activePieceId !== null) {
64          const currentActivePiece = getPieceById(state.activePieceId, state.allPieces);
65          deleteMoves(currentActivePiece, state.allCells);
66        }
67        generateMoves(piece, state.allCells, state.allPieces);
68        state.activePieceId = piece.id;
69      }
70    },
```

Figure 21. Code for pieceOnClick reducer

In order to find a clicked piece object in the store, findPieceById function imported from GameUtils.ts file is used. This function takes two arguments which are a piece id and the list of all pieces and returns a piece with this id from the list.

If there is already an active piece when a piece is clicked, all generated moves need to be deleted. To do this, deleteMoves function is used. This function takes two arguments, the clicked piece object and the list of all cells. It first empties the list of all moves for the clicked piece, and then sets the moveType property of all cells back to null.

For the purpose of generating a piece's all possible moves from its current position, generateMoves function is imported from GameUtils.ts file and used. It takes the clicked piece, the list of all cells, and the list of all pieces as arguments. This function first finds all possible cells that this piece can move to base on its name, side, and current position, and then changes these cells' type according to the type of the move. Furthermore, all these reachable cells are stored in the clicked piece object's allMoves after the function finished. Figures 22 and 23 describe what happens when different pieces in different places are clicked.

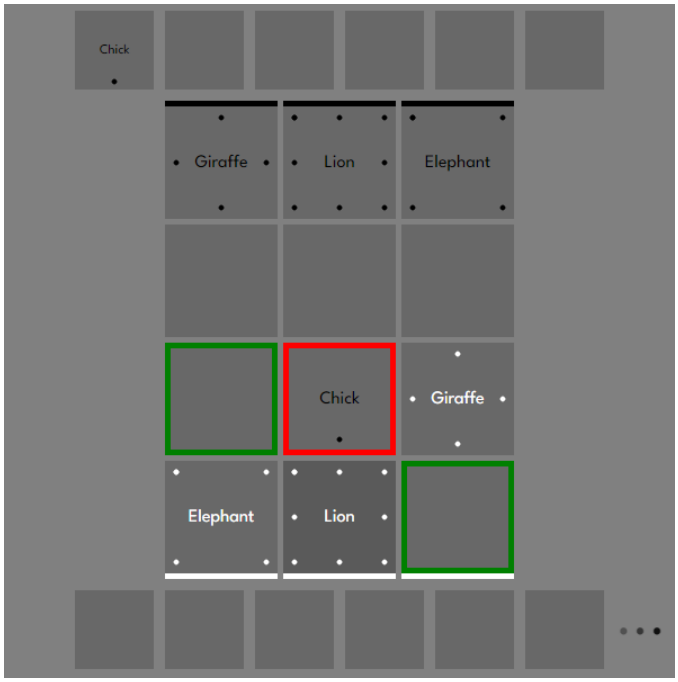


Figure 22. Possible moves generated when white Lion is clicked

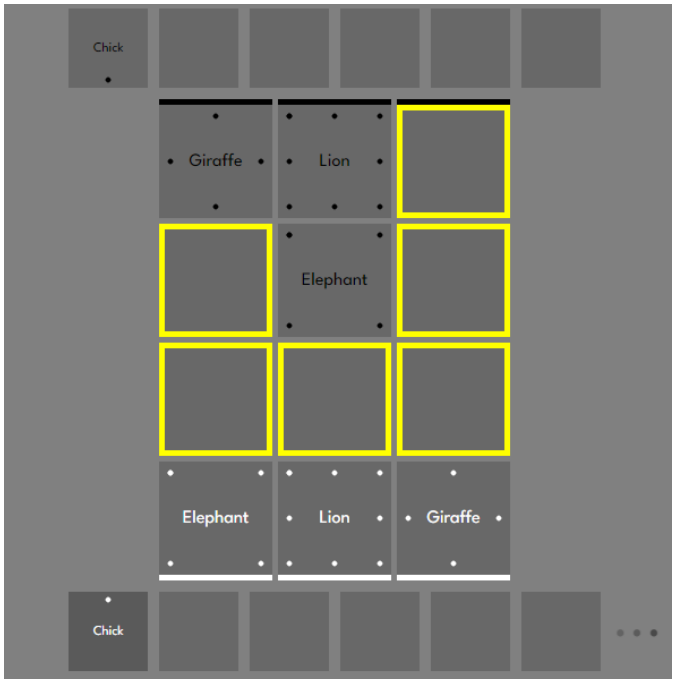


Figure 23. Possible moves generated when white Chick is clicked

Cells are rendered differently based on its moveType, so users can see all possible moves clearly on screen. In Figure 22 the red cell represents an attack to the black Chick, while green cells show that they are just normal moves between cells. In Figure 23, yellow cells represent all positions that the white Chick can be dropped back to the board.

3.6.3 Handling a move

After generating all possible cells that a piece can reach from its current position, users will be able to make a move by simply clicking on one of these cells. This feature is implemented by dispatching cellOnClick action after checking if the clicked cell is reachable for current active piece.

The checking process is quite simple. It is mentioned before that when finding all reachable cells for a piece, the type of them is changed according to the type of move. Therefore, if the moveType property of the clicked cell is one of three MoveType states (move, atk, or rev), that cell is one of the destination cells, otherwise, it is not.

If the clicked cell is reachable, cellOnClick action is dispatched. Figure 24 illustrates the logic workflow inside the reducer function.

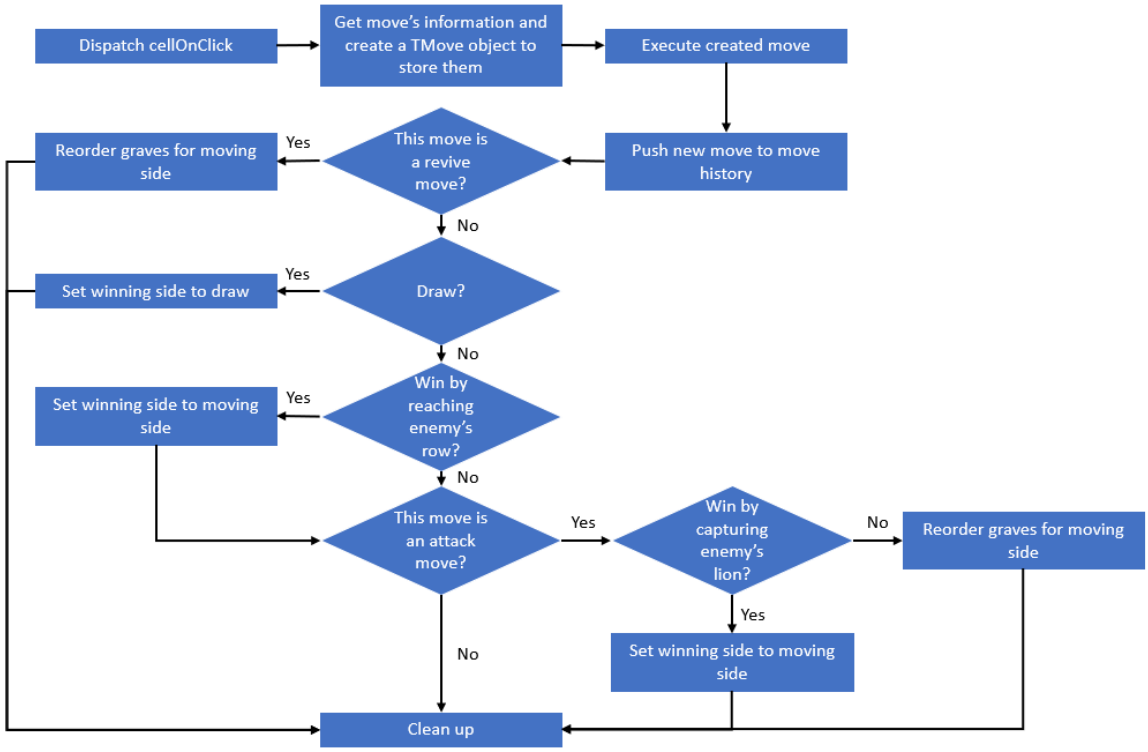


Figure 24. Logic workflow inside cellOnClick reducer

Figure 25 shows the code from cellOnClick reducer that is responsible for getting all needed information for a move and storing them in an object.

```
74        const moveType: MoveType = actions.payload.moveType;
75        const movePiece = getPieceById(state.activePieceId, state.allPieces);
76        let killedPiece: TPiece | null = null;
77        if (actions.payload.currentPieceId !== null) {
78          killedPiece = getPieceById(actions.payload.currentPieceId, state.allPieces);
79        }
80        let fromCell: TCell | null = null;
81        if (movePiece.currentCell !== null) {
82          fromCell = getCellByPos(movePiece.currentCell.x, movePiece.currentCell.y, state.allCells);
83        }
84        const toCell = getCellByPos(actions.payload.x, actions.payload.y, state.allCells);
85
86        // define move (promote and demote is not set yet)
87        let newMove: TMove = {
88          type: moveType,
89          movePiece: movePiece,
90          killedPiece: killedPiece,
91          fromCell: fromCell,
92          toCell: toCell,
93          promote: false,
94          demote: false,
95        };
```

Figure 25. Code for getting all information of a move and storing them in an object

In Figure 25, moveType represents the type of this move, while movePiece and killedPiece represent moving piece and captured piece (if this is an attack move). Different pieces' object is searched from the store using getPieceById function. Next, fromCell and toCell are origin and destination cells for this move. If the clicked piece is currently in hand, fromCell is known as null. These cells' object is gotten from the store using getCellByPos function, which is imported from GameUtils.ts file. The function takes the position of a cell and the list of all cells as arguments and then returns a cell with the same position from the list. At this point, promote and demote properties are set to false by default and can be modified if needed when the move is executed.

Figure 26 describes the code from cellOnClick reducer that is responsible for executing the move and updating the move history.

```
97          // move execute
98          moveExecute(newMove);
99          // push to moveHistory
100         state.moveHistory.push(newMove);
```

Figure 26. Code for executing move and updating move history

moveExecute function is imported from GameUtils.ts file. This function is used to update current state of different properties of the move object which is passed as argument. Additionally, it also checks if this move is a promotion move, demotion move, or both and changes the move's properties accordingly. After the new move is executed, it will be pushed to the moveHistory state from the store.

Figure 27 shows code that is responsible for checking game over state and reordering pieces in hand, which is the rest of the logic diagram except cleanup part.

```
102         if (moveType === MoveType.rev) {
103           reorderPieceInGraves(movePiece.side, state.allGraves, state.allPieces);
104         } else {
105           if (drawCheck(state.moveHistory) === true) {
106             state.winningSide = Winner.draw;
107           } else {
108             if (gameOverByReachingCheck(newMove, state.allCells, state.allPieces) === true) {
109               if (movePiece.side === Side.white) {
110                 state.winningSide = Winner.white;
111               } else {
112                 state.winningSide = Winner.black;
113               }
114             }
115             if (moveType === MoveType.atk && killedPiece !== null) {
116               if (killedPiece.name !== PieceName.lion) {
117                 reorderPieceInGraves(movePiece.side, state.allGraves, state.allPieces);
118               } else {
119                 if (movePiece.side === Side.white) {
120                   state.winningSide = Winner.white;
121                 } else {
122                   state.winningSide = Winner.black;
123                 }
124               }
125             }
126           }
127         }
```

Figure 27. Code for checking game over state and reordering pieces in hand

To improve users' experience, captured pieces will always be placed from left to right, in the following order: Chick, Elephant, and then Giraffe. In order to achieve that, a function called reorderPieceInGraves, which is imported from GameUtils.ts file, is used. It takes the side where the pieces' order needs to be considered, the list of all graves, and the list of all pieces as arguments. First, this function filters the array of all pieces to get only pieces in hand (current cell is null) and divides them into three smaller arrays according to their type: chick, elephant, and giraffe. Then, concat method is used on these arrays to form a new array of pieces that applies the new order. The array of all graves is filtered to select only graves of the chosen side. Before placing in new pieces with the right order and updating grave's information, each grave in this list has its currentPieceId property set back to null.

When two players play the same move three times in a row, the game will end in a draw. In order to check for this result, drawCheck function is used. The function investigates current move history and compares nine last moves from it. If the move has just been executed forms three repeated moves in a row, the winning side state is set to draw.

As introduced in Dobutsu Shogi rules, the game will also over when a Lion reaches the furthest rank without being captured in the next move, or when a Lion is captured. It is quite simple to check for the second situation. If the nearest move is an attack move and the captured piece is a Lion, then the winning side is set to the moving side. However, to check for the first condition, a more complicated function which is gameOverByReachingCheck is used. This function takes the recent move, the list of all cells, and the list of all pieces as arguments and returns a Boolean representing if the game ended or not. If the moving piece in this move is a Lion and the rank of the destination cell is the furthest, an array contains only opponent's pieces which are on board is filtered from the list of all pieces. After calling generateMoves function on all of them, allMoves property of each piece is checked. If there is no attack move with a Lion as captured piece, Boolean true will be returned, and the winning side is set to the moving side.

Finally, there is a cleanup part from the logic diagram that is responsible for deleting generated moves and updating current active piece as well as current side. Figure 28 describes how this part is implemented.

```
129        // clean up
130        deleteMoves(movePiece, state.allCells);
131        state.activePieceId = null;
132        state.currentSide = changeSide(state.currentSide);
```

Figure 28. Code for cleaning up after making a move

After deleting all moves to get all cells back to their normal state, the id of current active piece is set back to null and changeSide function is called to update the current side. Now, the board is back to its normal state with no active piece or generated moves and ready for the next move.

3.6.4 Undoing a move

At first, undo a move seems like an extra feature for this project since it not really affects the gameplay or the AI implementation. However, the way to implement the minimax algorithm later needs the ability to undo moves. As a result, some extra lines of code are added beside undo function to make undo a separate game feature that helps improving user's experience.

When undoHandler action is dispatched from a component, based on the current game mode and game state, different numbers of move are undone. For example, while two-player mode is selected, only one move is undone each time the button is clicked. However, while one-player mode is selected, if the computer already made its move, two moves need to be undone, otherwise, only one move is required. The number of moves then will be passed to the action as the payload object to decide how many times undo logic block will be looped. Figure 29 describes the logic diagram for each loop.

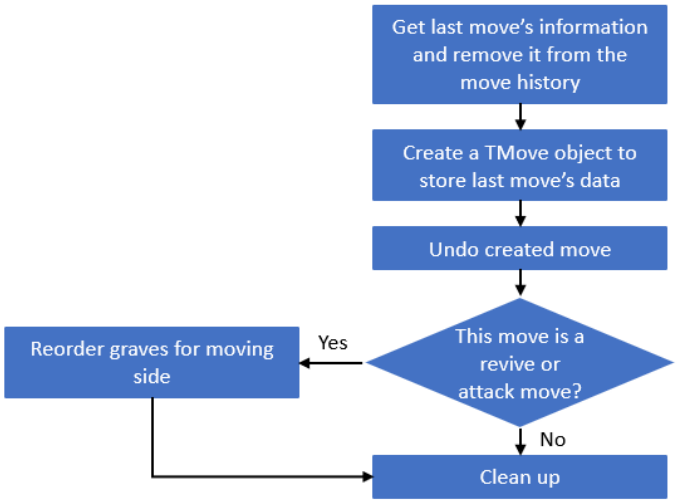


Figure 29. Logic workflow for undoing a move

Firstly, to get the last move's data and remove it from the move history, pop method is used. The process of creating an object to get and store a move's related information from the game state is the same as in cellOnClick reducer. However, this time the created object contained correct promote and demote properties already. Figure 30 shows the code from undoHandler reducer that is responsible for getting all information of last move from the game state and store them in an object.

```
// get all move info
const movePiece = getPieceById(lastMove.movePiece.id, state.allPieces);
let killedPiece: TPiece | null = null;
if (lastMove.killedPiece !== null) {
  killedPiece = getPieceById(lastMove.killedPiece.id, state.allPieces);
}
let fromCell: TCell | null = null;
if (lastMove.fromCell !== null) {
  fromCell = getCellByPos(lastMove.fromCell.x, lastMove.fromCell.y, state.allCells);
}
const toCell = getCellByPos(lastMove.toCell.x, lastMove.toCell.y, state.allCells);

const lastMoveCopy: TMove = {
  type: lastMove.type,
  movePiece: movePiece,
  killedPiece: killedPiece,
  fromCell: fromCell,
  toCell: toCell,
  promote: lastMove.promote,
  demote: lastMove.demote,
};
```

Figure 30. Code for getting information of last move and storing them in an object

Figure 31 describes code that is responsible for the rest part of the logic diagram in Figure 29, including undo the move, check for pieces in hand reordering, and clean up.

```
moveUndo(lastMoveCopy);

// update grave info if this is atk or rev move
if (lastMove.type === MoveType.atk || lastMove.type === MoveType.rev) {
  reorderPieceInGraves(movePiece.side, state.allGraves, state.allPieces);
}

// clean up
deleteMoves(movePiece, state.allCells);
state.activePieceId = null;
state.currentSide = changeSide(state.currentSide);
```

Figure 31. Code for undoing last move, checking for in-hand pieces' order, and cleaning up

In order to undo a move, a function imported from GameUtils.ts file called moveUndo is used. This function is basically the same as moveExecute function, which is used when implementing move handling feature. It takes the newly created object of last move as the argument and updates current state of its properties. After the state of related cells and pieces are updated, pieces that are currently in hand of the moving side are reordered if the type of undo move is attack or revive. Finally, the same cleanup code as the one used in cellOnClick reducer is applied to make the board get ready for the next move.

3.7 Game AI implementation

In this part, the process of implementing a simple AI for this game is showed in detail. In order to look ahead the next couple of moves and return the best one, an AI object is created using class. Instead of passing the board state as an argument for minimax function as showed in Figures 2 and 4, the state of the game will be stored and updated continuously through class's properties and methods. After the best move is calculated, the whole process of handling a move is applied in the reducer to make that AI move on the board and get ready for the next user's move. Figures 32 and 33 describe the AIHandler class and how an AI object is created inside aiHandler reducer using a depth of seven.

```
5   class AIHandler {
6     // algorithm needed info
7     nodeTraversed: number = 0; // only needed for debugging
8     maxDepth: number;
9     bestMove: TMove;
10    bestMoveScore: number = -10000000;
11
12    // current game state
13    allCells: TCell[];
14    allPieces: TPiece[];
15    aiSide: Side;
16    playerSide: Side;
17    moveHistory: TMove[];
18
19    // deep clone to store game state objects, declare bestMove object
20    constructor(maxDepth: number, allCells: TCell[], allPieces: TPiece[], aiSide: Side, moveHistory: TMove[]) {…
71    }
72
73    getBestMove() {…
78    }
79
80    minimax(depth: number, alpha: number, beta: number, maximize: boolean) {…
129   }
130
131   getAllSideMoves(side: Side) {…
182   }
183
184   gameOver() {…
215   }
216
217   evalutate(winner: Winner | null, depth: number) {…
282   }
283 }
```

Figure 32. Code for AIHandler class

```
177     let ai = new AIHandler(7, state.allCells, state.allPieces, actions.payload, state.moveHistory);
```

Figure 33. Code for creating an AI object inside aiHandler reducer

Most properties are initialized using constructor method, except nodeTraversed and bestMoveScore. nodeTraversed represents the number of game states the algorithm has searched and is used for better illustrating the effectiveness of

different algorithm variations. bestMoveScore represents the score of the best move and is initialized with a very low value so it will be updated no matter how bad the found move can be. Inside the constructor, current game states are cloned from game state store, while depth number and best move object are initialized. All objects representing current game state are deep cloned. Therefore, in case there are some unexpected behaviors happened while running the algorithm, the original state remains unchanged. For aiSide and playerSide properties, they are set using payload object when aiHandler action is dispatched.

3.7.1 Minimax with alpha-beta pruning

As described in the pseudo-code for alpha-beta pruning in Figure 4, the minimax function will first check if current examining node is a terminal node to run static evaluation function. If it is not, then based on whether current player is maximizing or minimizing, the function performs different actions to determine highest and lowest score from the child nodes accordingly. In this part, only the code for maximizing and minimizing player is described. The whole process of checking for terminal nodes to assigning evaluation score will be discussed later.

Figure 34 illustrates minimax method (alpha-beta pruning variation) with hidden code blocks. In line 81, nodeTraversed property is updated every time the method is called to represent the number of searched board states.

```
80   minimax(depth: number, alpha: number, beta: number, maximize: boolean) {
81     this.nodeTraversed = this.nodeTraversed + 1;
82
83     // check for terminal positions
84   >  if (depth === 0 || this.gameOver() !== null) {···
86     }
87
88   >  if (maximize) {···
111  >  } else {···
128    }
129  }
```

Figure 34. Code for minimax method

Figure 35 shows the logic code that is responsible for finding maximum score for maximizing player.

```
88  if (maximize) {
89    let maxEval = -1000000000;
90    let allMoves: TMove[] = this.getAllSideMoves(this.aiSide);
91    allMoves.every((move) => {
92      this.moveHistory.push(move);
93      moveExecute(move);
94      let currentEval = this.minimax(depth - 1, alpha, beta, false);
95      moveUndo(move);
96      this.moveHistory.pop();
97      maxEval = Math.max(maxEval, currentEval);
98      if (depth === this.maxDepth) {
99        if (currentEval > this.bestMoveScore) {
100         this.bestMoveScore = currentEval;
101         this.bestMove = move;
102       }
103     }
104     alpha = Math.max(alpha, currentEval);
105     if (beta <= alpha) {
106       return false;
107     }
108     return true;
109   });
110   return maxEval;
111 } else {···
```

Figure 35. Code for finding best score for maximizing player inside minimax method

This implementation is almost the same as showed in the pseudo-code; thus, it contains some changes. The major modification is that instead of passing the child board positions down to examine them, all current possible moves will be executed, one by one. By executing a move before calling minimax again, the new call will apply to the new board state that is formed after the move is made. Furthermore, the previous board positions can be set back using created undo function. While doing and undoing these fake moves, the move history is also updated along using pop and push methods. Additionally, an extra code block is added from line 97 to 102 to check if the current depth is the max depth to update the best move object and its score right after a better move is found.

For finding all possible moves, getAllSideMoves method is implemented. The method calls generateMoves function on each piece of the current side, adds generated moves to an array, and before returning this array as the result, it will reorder all moves based on how good they seem to be. The order from best to worst, which is based on my experience, is the captured moves, revive moves, and then normal moves between cells.

For finding lowest score for minimizing player, the same logic is applied with some modifications on the values. Figure 36 describes the code block that is

responsible for finding the worst score for minimizing player inside minimax method.

```
111     } else {
112       let minEval = 1000000000;
113       let allMoves: TMove[] = this.getAllSideMoves(this.playerSide);
114       allMoves.every((move) => {
115         this.moveHistory.push(move);
116         moveExecute(move);
117         let currentEval = this.minimax(depth - 1, alpha, beta, true);
118         moveUndo(move);
119         this.moveHistory.pop();
120         minEval = Math.min(minEval, currentEval);
121         beta = Math.min(beta, currentEval);
122         if (beta <= alpha) {
123           return false;
124         }
125         return true;
126       });
127       return minEval;
128     }
```

Figure 36. Code for minimizing player inside minimax method

Figure 37 shows getBestMove method where the minimax algorithm is called for the first time. After the algorithm finished running, the method will log the number of traversed nodes with the best move score and return a TMove object as the best move.

```
73    getBestMove() {
74      this.minimax(this.maxDepth, -1000000000, 1000000000, true);
75      console.log('number of searched node: ' + this.nodeTraversed);
76      console.log('best score for side ' + this.aiSide + ': ' + this.bestMoveScore);
77      return this.bestMove;
78    }
```

Figure 37. Code for getBestMove method

Figure 38 shows how getBestMove method is called from a created AI object inside the reducer.

```
179         let aiMove = ai.getBestMove();
```

Figure 38. Code for calling getBestMove from AI object inside aiHandler reducer

In aiHandler reducer, after the best move object is determined, it will be handled the same as described in handling a move feature.

3.7.2 Static evaluation

As explained before, when minimax algorithm proceeds down to the terminal nodes of the game tree, it will evaluate these positions and there will be no more recursions. A board position that is only considered as a terminal node when the current depth is zero or when the game is over. Figure 39 describes this process of checking for terminal positions to return evaluation value inside minimax method.

```
83    // check for terminal positions
84    if (depth === 0 || this.gameOver() !== null) {
85      return this.evalutate(this.gameOver(), depth);
86    }
```

Figure 39. Code for checking and evaluating terminal nodes inside minimax method

Figure 40 shows gameOver method which will look at current game state and return null if the game continues, or a Winner enum (white, black, or draw) if the game ended.

```
184   gameOver() {
185     const lastMove = this.moveHistory[this.moveHistory.length - 1];
186
187     // draw check
188     if (drawCheck(this.moveHistory)) {
189       return Winner.draw;
190     }
191
192     if (lastMove !== undefined) {
193       // check if lastMove was a killedLion move
194       if (lastMove.type === MoveType.atk && lastMove.killedPiece !== null) {
195         if (lastMove.killedPiece.name === PieceName.lion) {
196           if (lastMove.movePiece.side === Side.white) {
197             return Winner.white;
198           } else {
199             return Winner.black;
200           }
201         }
202       }
203
204       // check if lastMove was a winning move by reaching territory
205       if (gameOverByReachingCheck(lastMove, this.allCells, this.allPieces) === true) {
206         if (lastMove.movePiece.side === Side.white) {
207           return Winner.white;
208         } else {
209           return Winner.black;
210         }
211       }
212     }
213
214     return null;
215   }
```

Figure 40. Code for gameOver method

After knowing that a node is a terminal one, its score is calculated. Figure 41 illustrates the code for evaluation method when the game is over.

```
217   evalutate(winner: Winner | null, depth: number) {
218     let aiScore = 0;
219     let playerScore = 0;
220
221     if (winner !== null) {
222       if (winner === Winner.white) {
223         if (this.aiSide === Side.white) {
224           aiScore += 10000 + depth;
225         } else {
226           playerScore += 10000 + depth;
227         }
228       } else if (winner === Winner.black) {
229         if (this.aiSide === Side.black) {
230           aiScore += 10000 + depth;
231         } else {
232           playerScore += 10000 + depth;
233         }
234       }
235 >     } else {···
280     }
281     return aiScore - playerScore;
282   }
283 }
```

Figure 41. Code for calculating score difference when the game ended inside evaluate method

This method takes two arguments which are the gameOver method result and the current depth. The method first calculates the score for each side and then returns the difference as the result. If the game ended, a relatively high score with the current depth will be added to the winning side. In case the game is draw, the score difference stays zero. The reason behind adding current depth to the score is to make sure the AI will always try to win as soon as possible or delay its defeat. For example, if the next two move and the next three move for AI side are both winning moves, the nearer one will always be chosen. If the next two move and the next three move all lead to a loss, the AI should go with the further one.

If the game is not over yet, the score for each side is determined based on their current pieces, both on board and in hand. It is important to note that if a piece is currently on board, its position is not considered. As mentioned before, the higher quality of the evaluation function, the higher quality of the implemented AI will be. However, this project only uses a simple evaluation way to assess different game states since the main objective here is to create a simple AI system. Table 1

describes how each piece is evaluated based on my experience after playing this game for a long time.

Table 1. Score evaluation for different pieces in different places

Places	Pieces				
	Chick	Elephant	Giraffe	Hen	Lion
In hand	1	3	3	X	X
On board	2	6	6	12	10000

There is no score assigned to Hen and Lion piece when they are in hand since they will never be. If a Hen is captured, it is demoted back to Chick, and if a Lion is captured, the game is over.

3.7.3 Outcome

At this point, the game application has been finished and run fine in all game modes. The implemented AI system worked as expected. For better illustrating the effectiveness of different algorithm's versions (pure minimax, alpha-beta pruning, alpha-beta pruning with good move order), I have implemented and tested a separate AI system using each variation. Table 2 shows number of positions that each variation needs to search to make the optimal decision with different depth numbers when playing as white from initial board state.

Table 2. Number of nodes each algorithm version needs to search with different depth

Depth	Minimax	Alpha-beta pruning	Alpha-beta pruning with good move order
3	145	95	48
5	9,243	1,258	601
7	734,353	23,484	8,200
9	62,520,789	356,065	84,720

It is undeniable that the number of nodes that need to be searched has been decreased greatly when new techniques are applied. As a result, the speed of the

algorithm is improved significantly. The AI created in this project applies with only the depth of seven since my computing power and time to make a move in a real game is limited.

4 CONCLUSION

After having general knowledge about game theory's different types of games and the game tree, minimax algorithm basic concepts and how it is applied to develop a turn-based game AI that makes the optimal decision become easy to understand. Combining it with the use of alpha-beta pruning variation, the algorithm performance can be significantly improved.

The goal of this thesis was to develop a simulated version of a turn-based two-player board game named Dobutsu Shogi (Let's catch the lion) using different web development technologies and create a simple AI system based on minimax algorithm with alpha-beta pruning. This goal was achieved successfully. Different game features as well as the AI worked as expected. Furthermore, minimax solution was found to be easy to apply when implementing the AI system. After using different optimization ways that were introduced in the theoretical part, it can be observed from the result that the performance of the created AI was improved dramatically. It is interesting to note that this AI implementation way can easily be adapted and applied to other zero-sum games with perfect information to develop their own AI system.

It is undeniable that there is still plenty of room for further development for the AI system in this game. By applying a concept called transposition tables that prevents calculating the same board positions multiple times, the computation time can be greatly decreased. Furthermore, enhancing the current static evaluation function can also be considered as one way to increase the AI's quality since currently, it is built only based on my experience.

REFERENCES

Alpha-Beta Pruning. 2022. JavaTpoint. WWW document. Available at: https://www.javatpoint.com/ai-alpha-beta-pruning [Accessed 27 March 2022].

Dobutsu Shogi. 2022. PyChess. WWW document. Available at: https://www.pychess.org/variants/dobutsu [Accessed 29 March 2022].

Millington, I & Funge, J. 2009. Artificial Intelligence for Games. 2nd ed. Burlington: Elsevier.

Non-Zero-Sum Games vs. Zero Sum Games: Examples and Definitions. 2010. Bright Hub PM. WWW document. Available at: https://www.brighthubpm.com/risk-management/61459-comparing-zero-sum-and-non-zero-sum-games/#zero-sum-games [Accessed 23 March 2022].

Yannakakis, G & Togelius, J. 2018. Artificial Intelligence and Games. Cham: Springer.