Contents

Chapter 1

User Documentation

In this chapter, all the necessary information for the users is presented in detail. The Toguz Korgool game rules will be given first. Then we explain how to install and run the client and server applications. Also, the game modes and navigation instructions will be given in detail.

1.1 Toguz Korgool Rules

Toguz Korgool [1] ("nine dung balls") is the Kyrgyz name of a Mancala game [2] also known as Toguz Kumalak ("nine pebbles") in Kazakh. It is a two-player game that is played in Central Asia and is considered a national sport in Kyrgyzstan and Kazakhstan.

Figure 1.1: Souvenir wood board for Toguz korgool game

1.1.1 Board

Toguz Korgool is played on a special board having two rows of 9 small holes and two big holes called Kazans. The player's side is the bottom row of the small holes and the player's kazan is the bottom one (close to the player's side).

At the beginning of the game 9 balls are placed in each small holes.

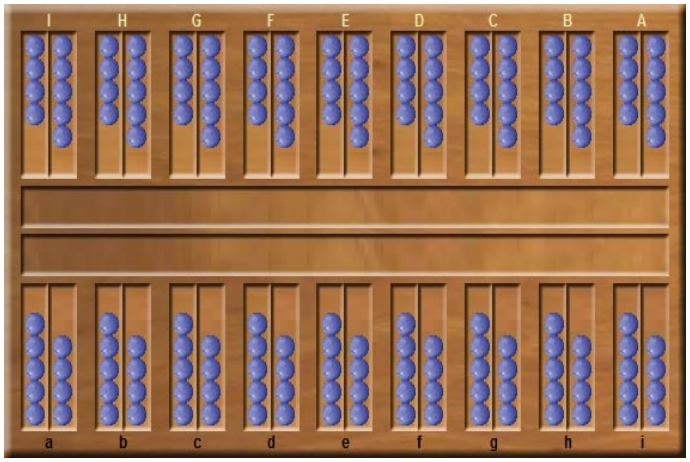

Figure 1.2: Board's initial position

1.1.2 Objective

A player wins a game if he earns more balls in his kazan than the opponent. If both players have earned the same number of balls, the game ends with a draw.

1.1.3 Play

Players take turns sowing their balls. The sowing is done in the following way:

- The player picks all balls up from one of the holes on her side.
- If the selected hole contained more than one ball, then the first picked up ball is dropped back to the starting hole. Then the player continues dropping the taken balls in a counter-clockwise direction, one ball in a hole.
- If the selected hole contained only one ball, the player drops it in the next hole in a counter-clockwise direction.

An example of sowing is given below:

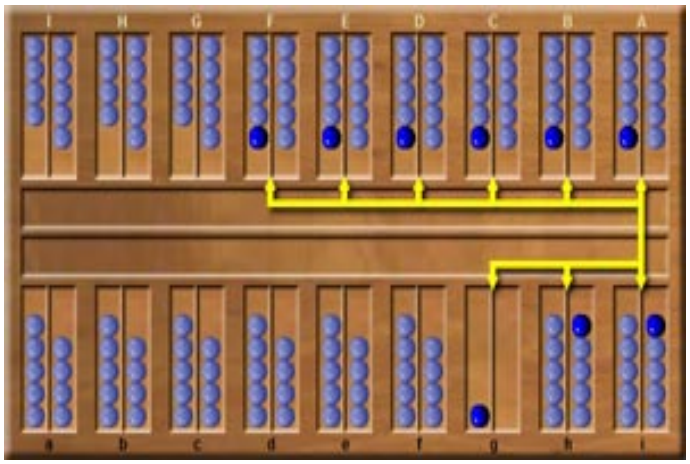

Figure 1.3: A player takes nine balls from her hole "g" and sows them counter-clockwise direction. Note that the first ball falls back to the hole "g."

- If the last sown ball lands in a hole on the opponent's side and brings the total number of balls in the hole to the even number, all balls from the hole are captured by the player and moved to the player's kazan.

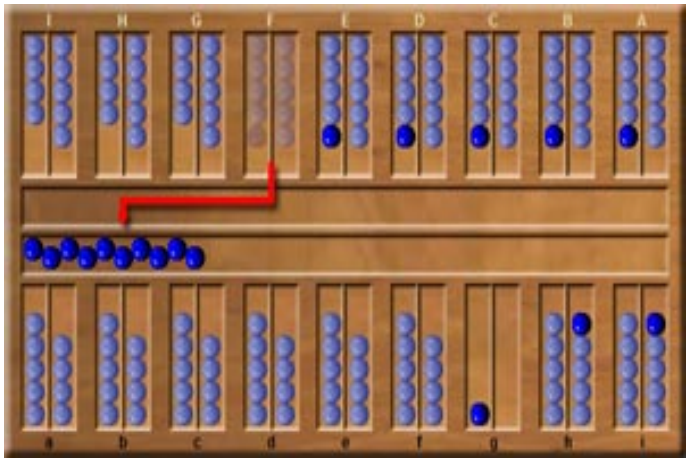

Figure 1.4: The last sown ball lands in a hole on the opponent's side and makes the total number of balls in the hole to the even number (10). All these 10 balls are moved to the player's kazan.

- If the last sown ball lands in a hole on the opponent's side and brings the total number of balls in the hole to three, then all balls from the hole are taken by the player and are moved to the player's kazan. The hole turns into the player's tuz ("salt" in Kyrgyz). There are several cases when tuz is not created:
  - If the player has created one tuz previously.

– The last hole of the opponent (his ninth or rightmost hole) cannot be turned into a tuz.

– A tuz cannot be created if it is symmetrical to the opponent's tuz (for example, if your second hole is already a tuz, you cannot turn the second hole of your opponent into one).

It is allowed to make such a move, but it would not create a tuz. All the balls that fall into a tuz during sowing are captured by the tuz's owner and are moved to his kazan.

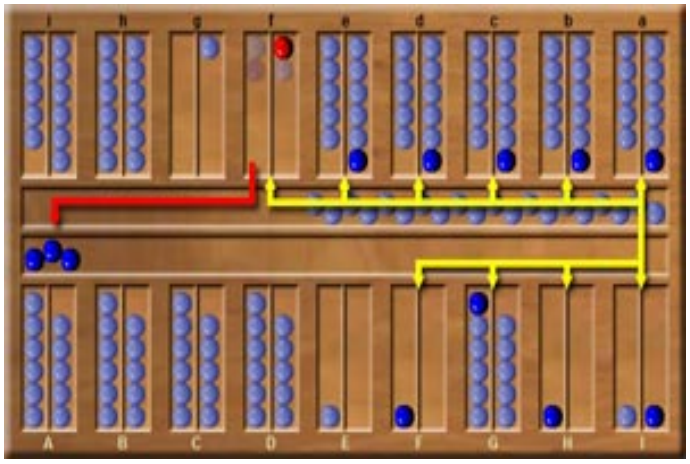

Figure 1.5: A player takes ten balls from her hole "F" and sows them counter-clockwise direction.The last sown ball lands in the hole "f" on the opponent's side and brings the total number of balls there to three. This hole is turned into the player's tuz (all three balls are moved to the player's kazan). Note that the red ball is used to label the tuz on the picture. It is not a ball; it is just a label.

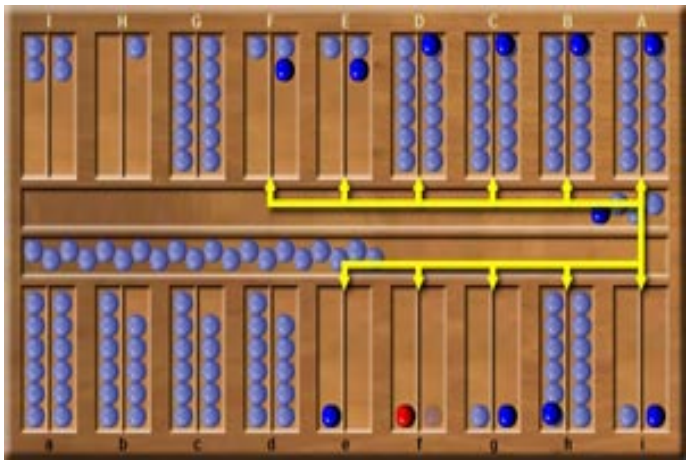

Figure 1.6: A player takes eleven balls from her hole "e" and sows them in a counter-clockwise direction. The last sown ball lands in the hole "F" on the opponent's side and brings the total number of balls there to three. This hole is NOT turned into the player's tuz because it is symmetrical to the opponent's tuz in hole "f." Note that during sowing, one of the player's balls felt into the opponent's tuz (hole "f") and is moved to the opponent's kazan.

- If a player does not have any balls on his side on his turn, then his opponent takes all remaining balls to his kazan, and the game ends.

1.2 Client application

The client application is an Android mobile application that allows users to play the Toguz Korgool game. Users can play the game with a human opponent over the internet in multiplayer mode or play it against a computer in a single-player mode. In this section, we will explain how to install and run the client application. We also present all the features available.

Android devices come in all sizes and shapes. Please note that this application was developed and tested on a mobile device emulator with a resolution of 1080 x 1920: 420dpi. If your phone's resolution is different, then some application pages will look slightly different, and some widgets will be visible partially. Making the application to support different screen sizes is a future improvement.

The application's user interface design was inspired by The Toguz Kumalak application made by Zhanat Nurbekova. You can find that application in the google play store using this link.

1.2.1 Installation

To install the application, download the apk file on an Android phone. Then, go to the location where the file is downloaded and run the file. Depending on the security setting of the device, you might be asked whether to trust this application or not. In such a case, you need to choose to trust the application. After that, the application is installed on your device.

1.2.2 Running

When you open the application, you will see the home page:

Figure 1.7: Home page

There will be three buttons shown. By tapping the SINGLE-PLAYER or MULTIPLAYER button, you will play the game in single-player or multiplayer modes, respectively. By tapping the SETTINGS button, you will be redirected to the settings page (Figure 1.8) for the game's single-player mode. On the settings page, you can choose the game's difficulty and whether you want to start the game.

Figure 1.8: Settings page

Tapping the SINGLE-PLAYER button redirects you to the page, where you can choose if you want to start a new game or continue a game from the state you left before. After you have made your choice, you will be redirected to the gameplay page. If you exit the application or gameplay page while the game did not end, then the application will save the game's state.

Tapping the MULTIPLAYER button redirects you to the page (Figure 1.9), where you can choose one from the three multiplayer mode types.

Figure 1.9: Multiplayer mode types page

Note that INTERNET types of multiplayer mode can be played only if the server application is up and running; otherwise, you will receive an "Unable to

connect to the server" error message. Information on installing and running the server application is given in Section 1.3.

- In case you have chosen the INTERNET RANDOM type, you will wait until your opponent will join the game or will start the game and be redirected to the gameplay page if there is an available opponent. Your opponent will be a random player from the internet who also chose to play the INTERNET RANDOM type. Before the game starts, you will be informed which player you are and whether the upper or lower row of holes is yours.
- In case you have chosen the INTERNET SPECIFIC type, you will be redirected to the page, where you will be asked if you want to create or join a game. If you chose to create a game, you will be given a unique id, which you should share with your friend you want to play with. If you chose to join a game, your need to ask for the id from your friend who created a game, then enter it in the input box (Figure 1.10) and tap the OK button. If the inserted id is correct, you will be redirected to the gameplay page; otherwise, you will receive an error message and be redirected to the home page. Before the game starts, you will be informed which player you are and whether the upper or lower row of holes is yours.

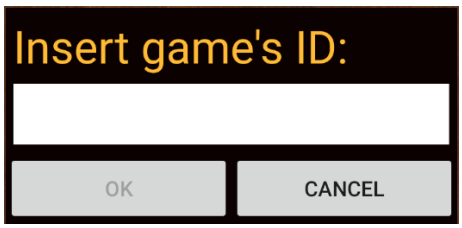

Figure 1.10: Input box to insert game id

- In case you have chosen the LOCAL type, you will be redirected to the page, where you will be asked if you want to start a new game or continue a game from the state you left before. After you have made your choice, you will be redirected to the gameplay page. This type is for people who are at the same location. The game is played by using one device for both players.

Gameplay page (Figure 1.11) is the place where the Toguz Korgool game is played. It consists of a game board, golden turn stick, animated sand clock, score

label, and sound button. The game board contains move holes and kazans. Each hole has one white and one orange label. The Orange label displays the hole number, while the white label displays the balls' count in the hole. The background of the tuz hole is different from simple holes. Hole number 1 of the upper row and hole number 7 in the lower row are tuz. Also, the last hole from which the move is made has yellow borders, as you can see the number 9 hole in the lower row. In a single-player mode of the game, the lower row of holes is yours, and the upper one is the computers. The lower row belongs to the first player in the game's multiplayer mode, and the upper one belongs to the second player. You will be informed which player you are before the game starts. If the turn stick is next to your holes, then it is your turn to make a move. The sand clock will be shown while your opponent is thinking. It will indicate that the application is not frozen. Score label shows the count of balls in each player's kazan. The sound button will enable/disable sound. After each move, a different sound will be played depending on the type of the move. The move can gain zero, less than 15, more than 15 balls, or tuz.

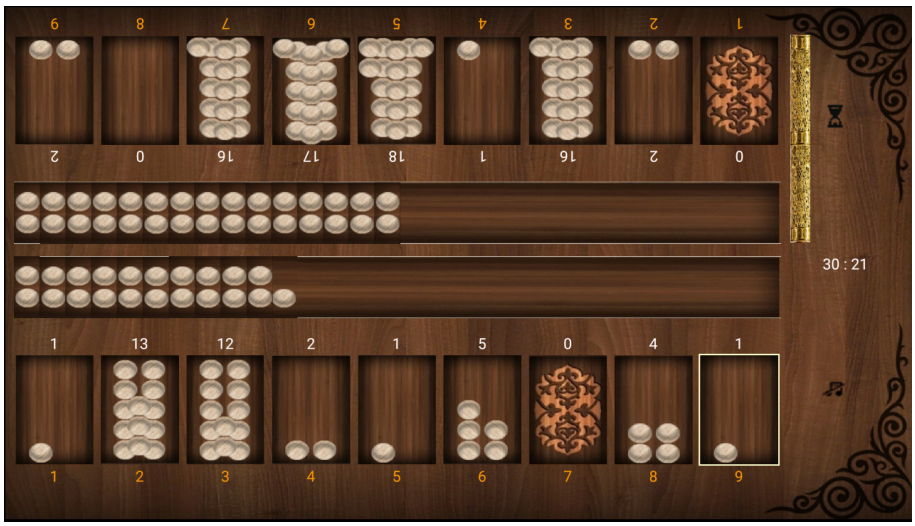

Figure 1.11: Gameplay page

You should tap one of your holes to make a move when it is your turn. If it is opponent's turn, then you are not allowed to tap any holes. The game will go on according to Toguz Korgool Rules (Section 1.1) until it is over and the alert dialog (Figure 1.12) pops up.

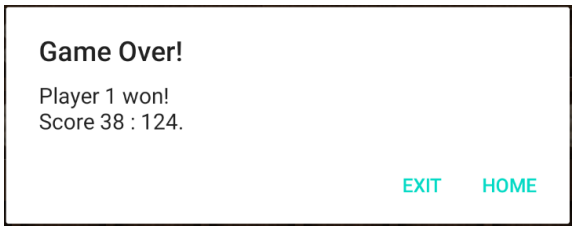

Figure 1.12: Game over alert dialog

During the game in multiplayer internet mode, if your opponent leaves the game, you will be informed about it and redirected to the home page. On each page, tapping the back button redirects you to the home page.

1.3 Server application

The server application is a console Java application, which connects players and maintains communication between them. In this section, we will explain how to install and run the server application.

1.3.1 Installation and running

Java version 8 or newer should be installed on your machine in order to run the program. You can download the Java from the official website. To run the program, first, download the server application class files from here. Then, run the Main class in the terminal. If successful, the string "Server started at port: 9000" will be printed in the console. To stop the server, press Ctrl+C.

In this chapter, we have given the Toguz Korgool game rules, installation and running of the client and server applications, and user experience of the main features. In the following chapter, we will explain how these applications are implemented.

Chapter 2

Developer Documentation

This chapter explains the technical aspect of the project. First, the specification of the problem is given. Then, the necessary information for setting up the development environment is described. Later, we will explain the implementation of the project. Finally, we will see how the program is tested.

2.1 Specification

To implement the Toguz Korgool game application, we first need to decide what features the application will provide. The users should be able to play the game against another human opponent or computer.So the user can choose to play the game in single-player or multiplayer modes.

In single-player mode, the computer player will think before it makes a move: it will evaluate the consequences of the next possible moves, guess the opponent's next moves, and make the optimal move in an acceptable time. The application will also save the game state if the user leaves the game while it is not over. The user can continue to play the unfinished game or start a new game. By choosing the game's difficulty level, the user can decide how well the computer opponent will play.

In multiplayer mode, the user will play the game against another human opponent over the internet or locally if the opponent and the user are at the same physical location. In case the connection is lost to the server or the opponent over the internet has left the game, the user will be informed about it. The user can decide if she wants to play against a specific or random human opponent over the

internet. In local multiplayer mode, the game state will be saved as in single-player mode.

The server application should be able to accept all requested connections and handle the communication between players at the same time. While the game is going on, if one of the players leaves the game, the server will notify that player's opponent about it. The player might intentionally leave the game, or there might be connection issues. The server should handle those issues. According to the game type the player wants to play, the server should match the player with the requested specific player or find another random connected player.

2.2 Development environment

This section will explain how to set up the development environment and run this project's applications. This project is available to the public on Github. In order to examine the source code, you can clone the remote git repositories of this project to your local machine.

You can visit the following link to see and download the server application source code: https://github.com/Sadykova-Nurbiike/toguz-korgool-server. Since the server code is written in Java, Java should be installed on your machine to compile and run it. You can explore the server source code in any favorite IDE/editor of your choice. To run the app, compile all .java files and run Main.class file. If you see the "Server started at port: 9000" message on the terminal, that means that running the server was successful.

The source code of the client application can be found in the following repository: https://github.com/Sadykova-Nurbiike/toguz-korgool-client. To be able to explore and execute the client application source code, you need the latest Android Studio (AS) to be installed on your machine. If you do not have it, you can download it from https://developer.android.com/studio/. AS has native integration with git and GitHub. When you open AS, it offers the option to **Check out project from Version Control**. When you select that option, the drop-down list will show available version control systems. You need to chose **Git**. You should then type the URL of the client application repository and choose the directory where you want to store

the project. After that, press **Clone**, then, AS will open the project ready to go.

You can now run the application on a real device or an emulator. We will explain how to run on an emulator; if you want to run on a real device, you can refer to [3]. To run the application on an emulator, follow the following steps [4]:

1. Create an Android Virtual Device (AVD) that the emulator can use to install and run your app in AS. AVD properties used to test and develop the application is shown in Figure 2.1.

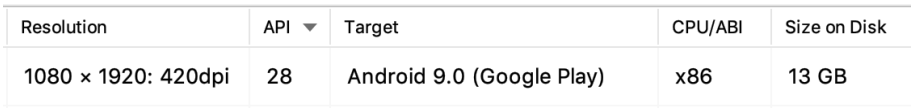

Figure 2.1: Emulator properties

2. Select **app** from the run/debug configurations drop-down menu in the toolbar.
3. Select the AVD that you have created to run the application on from the target device drop-down menu.
4. Press **Run**

AS installs the app on the AVD and starts the emulator. You now can see the home page (Figure 1.7) displayed in the application.

2.3 Server source code

Implementation of the server application will be presented in this section. The server connects players and handles communication between them. Players are clients that requests connection to the server. By using threads, we enable the server to handle multiple clients at the same time. According to the client's desire, the server should match the client with another specific or random client. A brief success use case is as follows:

1. The server starts by binding on a port.
2. The server waits for a client to connect.
3. After the client is connected, the server creates a thread to manage the client.

4. Then, the client's desired opponent type is checked. It can be random or specific. If the client's opponent is already connected, then two clients are matched in a Table where they can interact under some protocol; otherwise, the client is stored in a hashtable.

5. Steps 2, 3, 4 are performed until the program terminates.

The classes the server consists of are the following:

- *Main*: Entry point of the program. Initializes the Server class.
- *Server*: Starts the server and waits for the clients to join.
- *ClientHandler*: Handles communication between the server and the client.
- *Table*: Holds two matched clients and sets a communication between them.

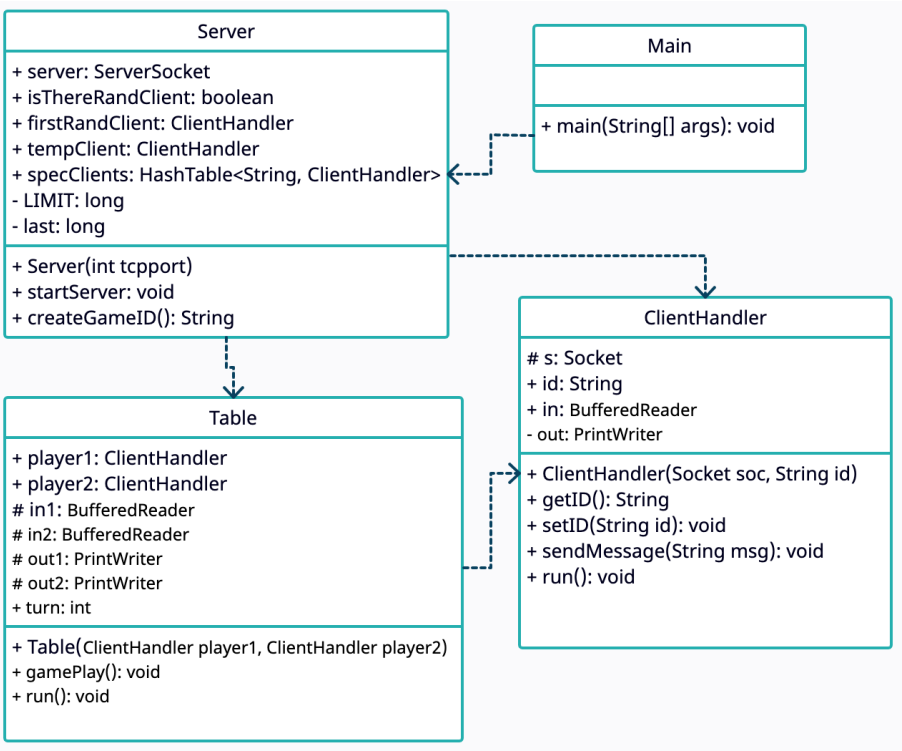

Figure 2.2: UML class diagram of the server

Let us now have a look at some classes in more detail.

Server

In this class's constructor, a new ServerSocket object is initialized to listen to a specified port. After, method startServer() is called.

Method startServer() runs an infinite while loop. The server waits for a client to connect inside the while loop. Once a client is connected, a new socket is created by calling the accept() method of the ServerSocket. Then, in order to achieve communication with the client, a new socket object is created. To handle more than one client, the server has to put each socket in a new thread. Therefore, the server puts every new socket in a ClientHandler object. ClientHandler class will be described later in this section. Then, the client informs the server of his desired opponent type. If the client wants to play with a random opponent, the server checks if there is another connected random player. If there is, the server matches the two clients into a Table thread; if not, the client is stored in a variable and will wait until another random client connects. If the client wants to play with a specific opponent, then the server waits for the client to inform if he wants to create or join a game. In case the client wants to create a game, the server generates a ten-digit unique string game id and stores the game id and client in a hashtable. Then, the server sends the game id to the client. The client will wait until his opponent joins the game. In case the client wants to join the created game, the server waits for him to send id of the game he wants to join. Once the server receives the game id and game id exists, the server matches the two clients into a Table thread. In the Table thread, the two clients will communicate under the actual protocol. Table class will be explained later in this chapter. If the received game id does not exist, then the server will inform the client. These steps will be repeated in the loop until the program stops.

ClientHandler

This class manages every connected client as a separate thread. It extends a Thread class, but it never calls its run() method. Its constructor accepts a socket object and a string, which will be the id of the thread. In order to read and write socket streams, BufferedReader and PrintWriter objects are assigned to the thread.

Table

This class controls messages sent between the two players. In the constructor, the server sends START1 or START2 message to each client to inform clients which player they are. The player is assigned appropriate holes in the Toguz Korgool board

according to which player she is. Table class extends the Thread class; it is why the server can handle more than one game simultaneously. The start() method of the Thread is called at the end of the constructor.

In the gamePlay() method, the server sends messages between the two clients. There are two if statements that check whose turn it is. If it is the first player's turn, then the server expects to receive a message from her. Once the message has arrived, then it is sent to the second player. In case the message is null, then it implies that the first player is not connected anymore. If it is the case, then the server informs the second player that her opponent has left the game by sending an EXIT message. Otherwise, an integer is sent to the second player. It is the move of the first player. If it is the second player's turn, it is the same, except the second player is sending to the first player. gamePlay() is called in the run() method of the thread.

```
1  public void gamePlay() {
2      turn = 1;
3      try {
4          while (true) {
5              if (turn == 1) {
6                  String received = player1.in.readLine();
7                  System.out.println(received);
8                  if (received == null) {
9                      player2.sendMessage("EXIT");
10                     System.out.println("EXIT message has sended to
    player 2");
11                     break;
12                 } else {
13                     player2.sendMessage(received);
14                     turn = 2;}}
15             else if (turn == 2) {...}
16         }
17     } catch (IOException e) {
18         System.out.println("The game has terminated because of
    connection problems");
19         player1.sendMessage("EXIT");
20         player2.sendMessage("EXIT");}}
```

Code 2.1: Server: method gamePlay

2.4 Client source code

In this section, we will present the client application's project structure. Important files and classes will be reviewed in detail.

2.4.1 Project structure

Client project consists of one Android app module which is called *app*. By default, AS displays project files in the Android project view, as shown in Figure 2.3.

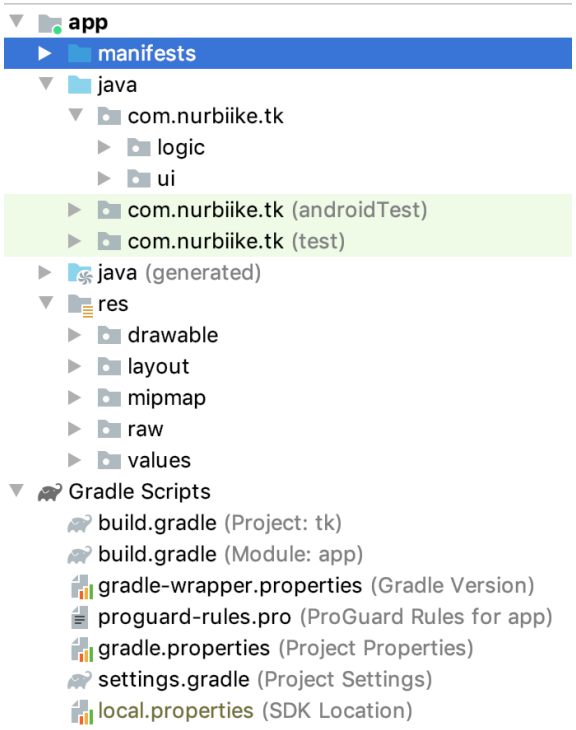

Figure 2.3: Client application project structure

At the top level under **Gradle Scripts**, all the build files are visible. The file with the name *build.gradle (Project: tk)* is for the project, and the other one with the name *build.gradle (Module: app)* is for the app module. The file *build.gradle* is used to control how the Gradle plugin builds the application. In the module's *build.gradle* file, all the external binaries or other library modules are included to the application's build as dependencies.

Folders in the **app** module are the following:

- **manifests**: Contains the *AndroidManifest.xml* file. The manifest file describes the essential characteristics of the application and defines each of its compo-

nents. Among many other things, this file requests internet access permission and declares all the activities and their properties.

- **java**: Contains the Java source code files. There are three folders:
  - **com.nurbiike.tk**: Contains folders **logic** and **ui**. The folder **logic** consists of the files *MoveSound.java*, *Move.java*, *GameState.java*, *GameStateMultiPlayer.java* and *GameStateSingle.java*, which define classes that are used to implement the gameplay of the Toguz Korgool game. Description of the above files will be given in Section 2.4.2. The folder **ui** consists of *UtilMethods.java* and activities. Activities and their corresponding layouts will be explained in Section 2.4.3.
  - **com.nurbiike.tk (androidTest)**: Contains instrumented unit tests. They are described in Section 2.5.2.
  - **com.nurbiike.tk (test)**: Contains unit tests. They are described in Section 2.5.1.
- **res**: Contains additional files and static content that are used in the code. It consists of folders listed as follows:
  - **drawable**: Contains bitmap files.
  - **layout**: Contains XML files that define the layout for the activities' user interface. Activities and their corresponding layouts will be explained in Section 2.4.3.
  - **mipmap**: Contains drawable files for different launcher icon densities.
  - **raw**: Contains audio files.
  - **values**: Contains XML files that contain simple values, such as strings, integers, and colors.

In this section, we went through the project structure of the client application. In the following section, we will explain classes that are used to implement the logic behind the Toguz Korgool game.

2.4.2 Game classes

In this section, the implementation of the Toguz Korgool game logic is described. In single-player mode, a computer player serves as an opponent for the user. To enable the computer to make an optimal move, we implemented a method, which uses

a minimax algorithm with alpha-beta pruning. First, we will explain how a minimax algorithm with alpha-beta pruning works. We will then illustrate the relationship between the most important classes. Later, we will dive deeper into some classes and describe their essential methods and attributes.

Minimax with alpha-beta pruning

Toguz Korgool is a turn-taking, two-player, zero-sum, and perfect information game, i.e., two players take turns to make a move. If one player wins, then the other player losses or the game will end with a draw. The player also sees the other player's holes and can evaluate the opponent's next possible moves - there are no elements of chance in the game. We use a minimax algorithm with alpha-beta pruning to enable a computer to make moves in the Toguz Korgool game. [5] [6] were referred to to solve this problem.

The minimax algorithm is a searching algorithm that can be used for optimal decision-making in perfect information, zero-sum, two-player games. Although it gives an optimal move, it is not efficient. Optimization technique alpha-beta pruning can be used for the minimax algorithm to improve its efficiency.

First, let us introduce game trees. A **game tree** is a directed graph consisting of nodes and edges, where nodes are legal states in a game and edges are legal moves. One of the players making a move means that the current game state is changing to another legal game state.

The complete game tree is a tree that contains as many nodes as the game's all possible outcomes for every legal move. In the complete game tree, the root node is a starting, and all the leaf nodes are ending states. For many games, the game tree becomes huge after a few moves and consumes many resources. That's why it is not practical to create a complete game tree explicitly, although we can get the best move by doing so. The nodes should be implicitly created while they are visited.

The **minimax algorithm** systematically traverses the game tree and uses a simple evaluation function. Since it is not practical to traverse the whole tree, in

most games, minimax applies search until some specified depth and uses a better yet simple heuristic evaluation function. With this approach, minimax not necessarily finds the best move, but it finds a good move. To determine a good move, it needs to evaluate the node's goodness/value to compare it with another. It is important to note that the evaluation function should not depend on the search of previous or the following nodes. It only should evaluate the game state.

Here, we will illustrate minimax's steps. In this case, we're looking for the minimum value. The green level is the Max player's turn, while the pink level is the Min player's turn. The Max player calls the Max() method on nodes in the child nodes, and the Min player calls the Min() method on child nodes.

1. Evaluating leaf nodes:

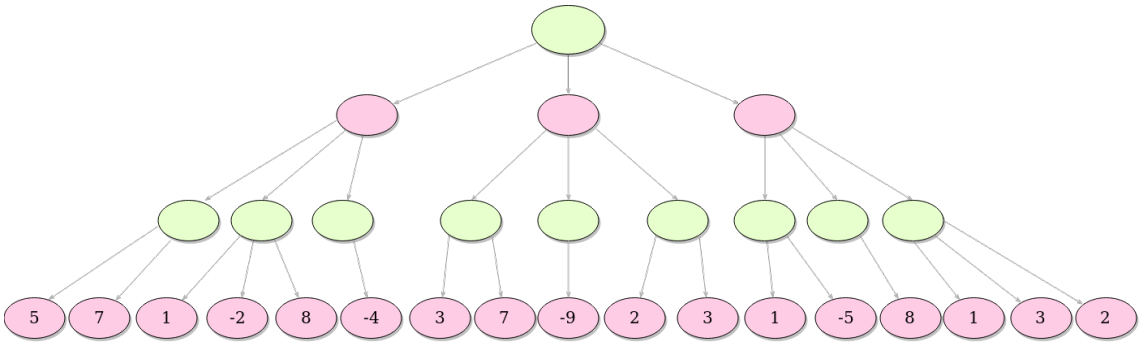

Figure 2.4: Minimax tree: evaluating nodes [7]

2. Picking the best move for Max player using depth 3:

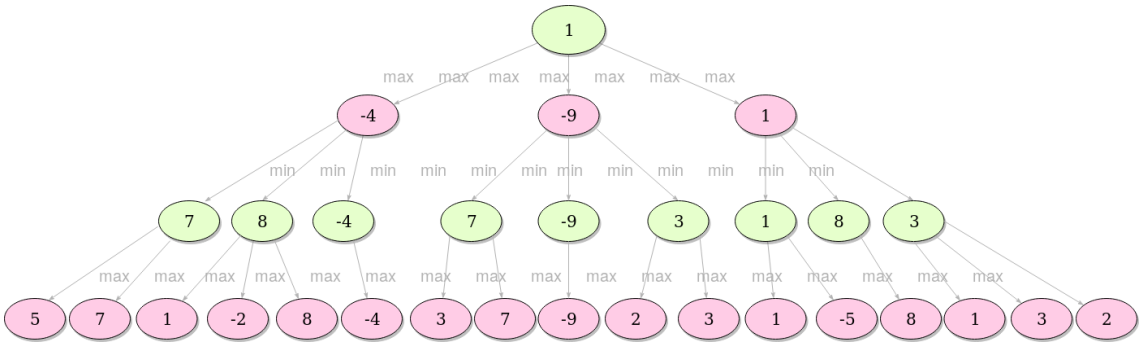

Figure 2.5: Minimax tree: picking the best move [7]

The idea is to find the best possible move for a given node, depth, and evaluation function.

We've assumed that the Max player seeks positive values, and the Min player seeks negative values. The algorithm primarily assesses only nodes at the given depth, and the rest of the procedure is recursive. If it's a Max player's turn, the rest of the nodes' values are the maximum of their respective children's values; if it's Min player's turn, the minimum value analogously. The value in each node is the next best move regarding the provided information.

The pseudo-code for the depth limited minimax algorithm is illustrated in Figure 2.6:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Figure 2.6: Depth limited minimax algorithm [8]

Alpha-beta is an optimization technique; when applied to a minimax tree, it returns the same move as minimax would but cuts off branches that cannot possibly influence the final decision.

The main idea is to keep two values throughout the search:

- Alpha: The best-explored move for player Max
- Beta: The best-explored move for player Min

Initially, alpha's value is negative infinity, and the beta's value is positive infinity, i.e., the worst possible scores for both.

If we apply alpha-beta pruning to the previous tree, then it will look like this:

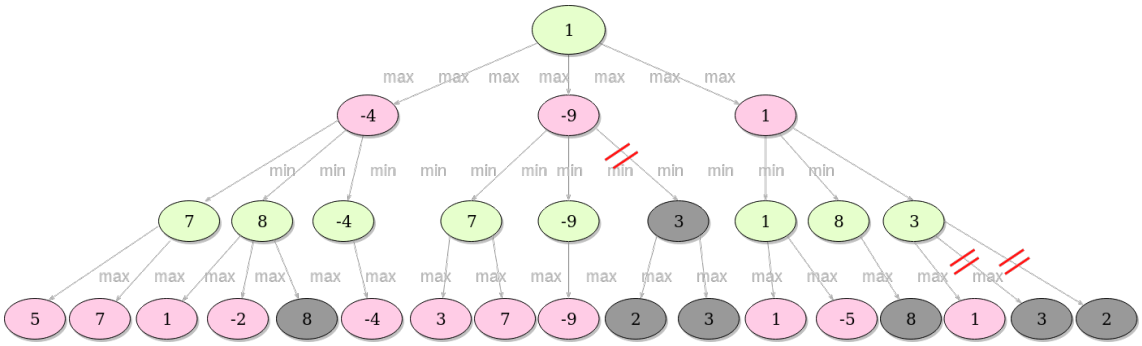

Figure 2.7: Minimax tree after alpha-beta pruning applied

Let us have a look at the second grey area. Note the node with the value -9 at the first level of the tree. At that moment, the value of alpha is -4. The value of the beta at the current node can be only <=-9 because the current node is Min player. Since the parent node of the current node is Max player, it won't choose the value <=-9 over -4. Thus, we don't need to explore further the children of the current node because it won't affect our decision. All grey nodes are part of the pruned branches.

The pseudo-code for depth limited minimax with alpha–beta pruning is illustrated in Figure 2.8:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cutoff *)
        return value
```

Figure 2.8: Depth limited minimax with alpha–beta pruning [9]

Game classes relationship

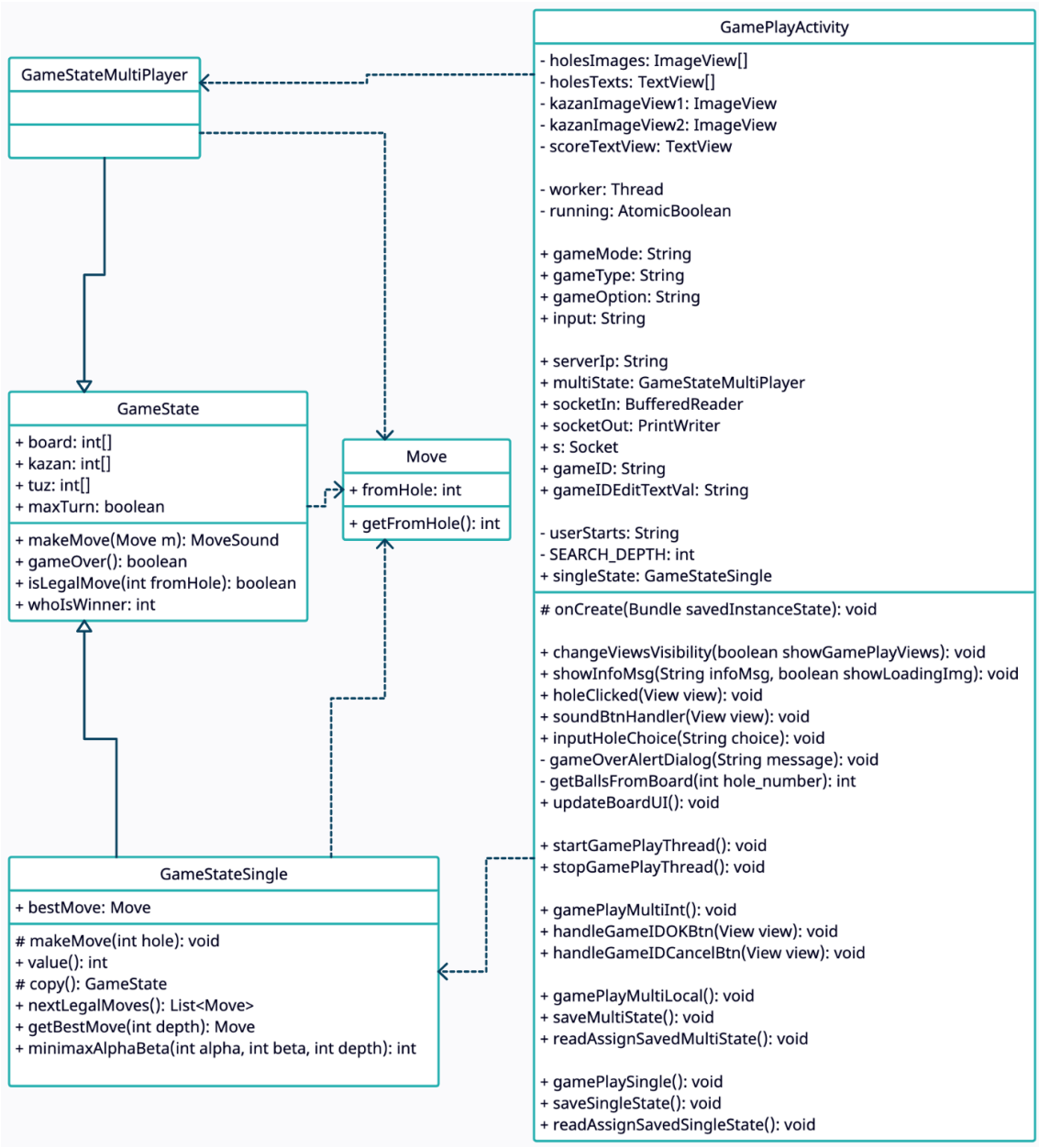

Figure 2.9: UML class diagram of the client

Only the most essential classes, methods and attributes are included in the above UML class diagram to keep it visually simple and easy to understand. However, we will explain all classes shortly or in detail later in this section.

Class Move

A Move object is a move in a game. It has one attribute of type int that is called fromHole. It represents the starting hole of the move.

Class GameState

It is an abstract class that will be a parent class of the GameStateSingle and GameStateMultiPlayer concrete classes. It represents a Toguz Korgool game state. In the following, we will describe its most important attributes and methods.

board[] is an int array with size 18. The board represents a Toguz Korgool board's holes. Elements 0-8 in the array are the upper holes of the board, and elements 9-17 are the lower holes. Upper holes belong to the second player, while lower holes belong to the first player.

kazan[] is an int array with size 2. It stores the values of each player's kazan. Element 0 of the array is the kazan of the first player, and element 1 is the kazan of the second.

tuz[] is an int array with size 2. Element 0 of the array is the number of the first player's tuz hole, and element 1 is the number of the second player's tuz hole. Initially, since both players do not have tuz, each element of tuz[] is assigned -1.

maxTurn is a boolean variable. It is used to determine which player's turn. It is the Max player's turn to move when maxTurn is equal to true; otherwise, the other Min player's turn. If maxTurn is true, then it always means that it is the user's opponent's turn.

your_number is an int variable, which indicates if the user is the first or the second player.

Constructor GameState() initializes the attributes of the object. This constructor is used in the game's single-player mode because we assign an initial value for your_number attribute to 1. It implies that the computer player is always the second player. In single-player mode, by a first and second player, we only define which holes are owned by the computer player and which ones are by the user. According to what the user wants, she can start the game or let the computer

player start.

Constructor GameState(int playerNumber) is used in the multiplayer mode of the game. It is the same as GameState(), except it accepts an argument playerNumber, which will be assigned to your_number attribute of this object. Users can be the first or the second player in multiplayer mode. The first player starts the game.

Method makeMove(Move m) accepts an argument of type Move and returns an enumeration of type MoveSound. It makes a given move according to Toguz Korgool rules and consequently changes the GameState object's state. Each move can gain zero or more balls or can make a tuz. This method returns a MoveSound object, which will be used to make an appropriate sound after each move in UI.

Method gameOver() checks whether one of the player's kazan value is equal to or more than 81 or one of the player's all holes are empty. If the mentioned conditions are met, then the game is over, and the method returns true; otherwise, it returns false.

Method isLegalMove(int fromHole) verifies if the hole with the given number is valid or not and returns a boolean value accordingly.

Method whoIsWinner() returns -1 if an opponent is a winner, 1 if a player is a winner, 0 if the game is a draw. It is used in gameplay methods to display an appropriate message to the user at the end of the game.

Class GameStateSingle

Class GameStateSingle extends class GameState. In addition to GameState methods, it contains methods used to enable the computer player to make a move. This class also implements interface Clonable because we make its state's copy inside the search algorithm, which finds a good move for a computer player.

Method value() returns an int value, which is 10 multiplied by the user's kazan value subtracted by the computer player's. Thus this method assesses the value of the current state of the game. It is good for the player if her kazan contains more balls than her opponent's.

Method copy() creates a new GameStateSingle object and assigns to that object's maxturn, board, kazan, and tuz fields the same value as its own. So the newly created object is a copy of the current object. This method is called in minimaxAlphaBeta method, where the algorithm retains the game's current state before it makes a move.

Method nextLegalMoves() creates next legal moves for the current player and returns a list of all next legal moves. This method is used in minimaxAlphaBeta method.

Method getBestMove(int depth) calls minimaxAlphaBeta method and returns the optimal move from the current state. Its argument depth is the maximum depth for the search algorithm minimaxAlphaBeta. Initial values of alpha and beta arguments of minimaxAlphaBeta method are Integer.MIN_VALUE and Integer.MAX_VALUE respectively, which are the worst values for each.

Method minimaxAlphaBeta(int alpha, int beta, int depth) finds the computer player's optimal possible move from the current game state. Argument alpha stores the best-explored move for player Max, while argument beta stores the best-explored move for player Min. Argument depth is the number that shows how many states the algorithm should think ahead. The more depths the algorithm searches, the better will be the chosen move. This method calls itself recursively to assess the current state at each depth.

The best move is stored in the bestMove field. First, the algorithm generates all next legal moves, and then it checks if the current state is terminal, i.e., the depth argument is 0, or there are no legal moves from the current state. If it is the case, then it returns the value of the current state. Otherwise, it iterates through all of its

legal moves. Before each move, it first remembers the current state, then makes a move. After it calls on a new state itself recursively and gets the value of the move, then it compares the value of the move with variable alpha's value if it is Max's turn or compares with beta if it is Min's turn. It changes the value of alpha/beta and bestMove if the current move's value is better than previously explored moves. The value of the final bestMove is the move that the computer player makes.

```
1  public int minimaxAlphaBeta(int alpha, int beta, int depth) {
2      List<Move> nextLegalMovesList = nextLegalMoves();
3      if (depth <= 0 || nextLegalMovesList == null ||
       nextLegalMovesList.isEmpty()) {
4          return value();
5      }
6      for (Move move : nextLegalMovesList) {
7          GameStateSingle gs = (GameStateSingle) copy();
8          gs.makeMove(move);
9          int value = gs.minimaxAlphaBeta(alpha, beta, depth - 1);
10         if (maxTurn && value > alpha) {
11             alpha = value;
12             if (alpha >= beta) {
13                 return alpha;
14             }
15             bestMove = move;
16         } else if (!maxTurn && value < beta) {
17             beta = value;
18             if (alpha >= beta) {
19                 return beta;
20             }
21             bestMove = move;
22         }
23     }
24     return maxTurn ? alpha : beta;
25 }
```

Code 2.2: Method minimaxAlphaBeta

Class GamePlayActivity

In this activity, the game is played. It handles the UI of the game board, initializes the GameStateSingle/GameStateMultiPlayer object, and calls the appropriate gameplay method. This section will explain only how gameplay is implemented; its UI-related part will be presented in Section 2.4.3. Method startGamePlayThread() is called in the entry point of GamePlayActivity after UI is initialized. The main thread handles UI elements and monitors/waits for user interaction, and performs the appropriate action. Since we need game logic to run simultaneously, we call the gameplay method on a different thread.

Method startGamePlayThread() creates a new thread and starts it. In the run method of the new thread, the program calls gamePlaySingle(), gamePlayMultiLocal(), or gamePlayMultiInt() method according to the type of the game user wants to play. The type can be SINGLE, MULTILOCAL, INTRAND, or INTSPEC, respectively.

Method gamePlaySingle() initializes the object of type GameStateSingle, which will be the game state. If the user wants to continue the previously left game, we restore the previous game state by reading it from SharedPreferences. SharedPreferences stores key, value pairs in an Android device's file system. Details about it will be given later. If the user wants to play the new game, then we continue with the new game state. Each time when the game state changes, we update UI by calling method updateBoardUI().

We will then enter a while loop, which iterates until the game is over. There are two blocks of code inside the loop, one for each player. We enter the appropriate block according to the player's turn. If it is the user's turn, then we wait for the user to choose a hole from the UI. If the chosen hole is invalid, we ask the user to choose again until he makes a valid choice. Then move is made on the game state. If it is a computer player's turn, then we call on the game state getBestMove method by passing to it the SEARCH_DEPTH. Once we get the best move, we make that move on the game state.

When the game is over, an alert dialog is displayed to the user with an

appropriate message and score of kazans. Then the user is redirected to the home activity.

Method gamePlayMultiLocal() is the same as gamePlaySingle() method, except it initializes an object of type GameStateMultiPlayer instead of GamePlaySingle. The other difference is that inside, while loop in both blocks, it waits for a user to choose a move from the UI.

Method gamePlayMultiInt() first connects to the server. If the server is down or there are connection issues, then it displays an error message to the user and redirects the user to the home activity. If the connection is successful, then we check whether the user wants a random or specific opponent. If it is the first one, then we send an INTRANDOM message to the server and wait for the opponent to join the game. If it is the second one, then depending on whether the user wants to create or join a game, we send to the server CREATEGAME or JOINGAME message. In case the user wanted to create a game, we wait for the server to send the game id. Once it is received, we display the game id to the user to share it with the person he wants to play the game. The user will wait until his opponent joins a game. In case the user wanted to join a game, we display an input box, where he can type game id. Then, we send that game id to the server. If the game id is invalid, then we get a NOTFOUND message from the server. In that case, we inform the user about it and redirect him to the home activity, where he can try again to connect to the game with the correct id or play it in another mode.

Once the server found an opponent and connected the players, we get START1 or START2 message. START1 means that the user will be the first player and will own holes 9-17 (lower row) on the board. Also, the user makes the first move. Analogously, START2 means that the user will be the second player and own holes 0-8 (upper row). We will display a message to inform which player he is. Each time when the game state is changed, we change UI accordingly.

Then, we enter a while loop, which iterates until the game is over. Inside while loop, there are two blocks. One block will be executed when it is the user turns, the other one when it is the opponent's. If it is the user's turn, we wait for him to

choose a hole to move. If he chooses a wrong hole, we ask him to choose another hole repeatedly until the hole is valid. Then we make a move on the game state and send the chosen hole number to the server. The server sends it to the opponent. If it is the opponent's turn, then we wait for the server to send us a number that indicates the opponent's move. Once the message has arrived, we make a move on the game state. Note that the server can send an EXIT message instead of a number; it means that the opponent has left the game. In that case, or if the connection to the server is lost, we inform the user about it by showing a message and redirect him to home activity.

Once the game is over, we show an alert dialog to the user with the game's result.

2.4.3 Activities and layouts

The Android app consists of one or more app components that are the essential building blocks of it. [10] A system or a user can enter an app by each component because they are entry points. Unlike apps on many other systems, Android apps do not have only one entry point. The system starts a process for that app if it is not running when it starts a component. Components are independent of each other. Services, content providers, broadcast receivers and activities are the four types of app components. This project contains only activities.

An **activity** is the entry point for interacting with the user. It embodies a single screen with UI. Each activity class is a subclass of the Activity class. An app draws its UI in a window which is provided by an activity.

This project consists of 7 activities. HomeActivity is the main Activity, which is the first screen that appears when the app is launched. Each Activity can start another activity. The diagram in Figure 2.10 shows from where we can access particular Activity.

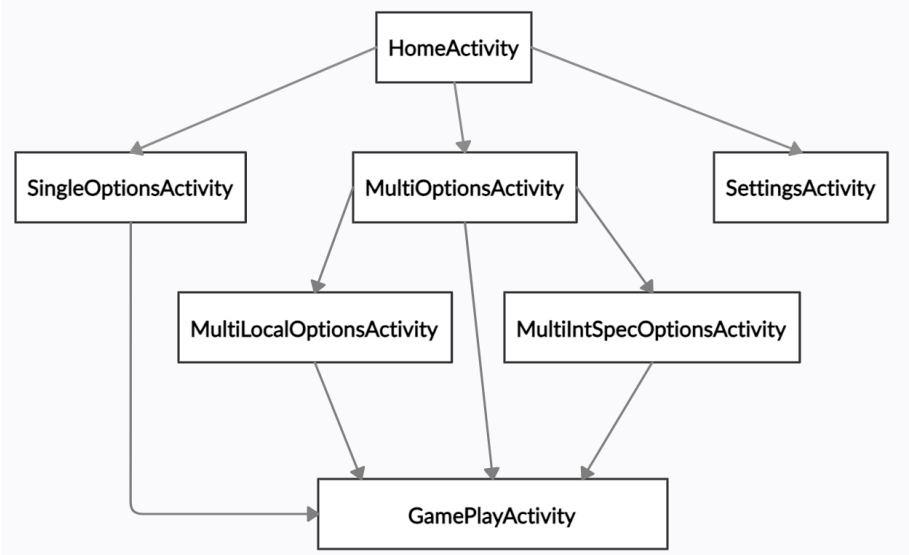

Figure 2.10: Flow between activities

On each Activity, tapping the device's Back button starts HomeActivity.

Before the system can start an activity, it must know that the Activity exists by reading the app's manifest file, AndroidManifest.xml. All activities are declared in this file.

```
<manifest...>
    ...
    <application...>
        <activity android:name=".ui.MultiIntSpecOptionsActivity"
    .../>
        <activity android:name=".ui.MultiLocalOptionsActivity".../>
        <activity android:name=".ui.MultiOptionsActivity".../>
        <activity android:name=".ui.SingleOptionsActivity".../>
        <activity android:name=".ui.SettingsActivity".../>
        <activity android:name=".ui.HomeActivity"...>
            ...
        </activity>
        <activity android:name=".ui.GamePlayActivity".../>
    </application>
</manifest>
```

Code 2.3: AndroidManifest.xml

A **layout** defines the structure for a UI in an app. Its elements are View and ViewGroup objects. A View draws something visible to the user, and a user can

interact with it. ViewGroup is a container that is not visible. It defines the layout structure for layout elements, as displayed in Figure 2.11.

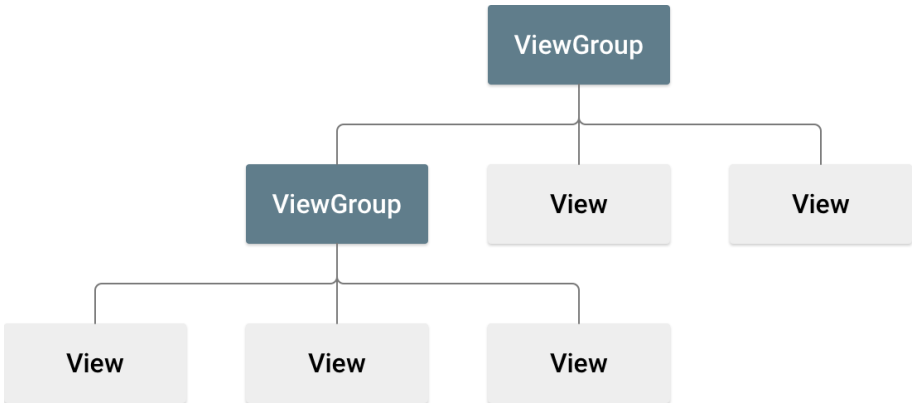

Figure 2.11: Illustration of a view hierarchy, which defines a UI layout

The View objects can be one of many subclasses, like EditText or ImageView. They are usually referred as widgets. The ViewGroup objects can be one of many subclasses, like ConstraintLayout or LinearLayout, that define different layout structures. The ViewGroup objects are usually referred as layouts. UI elements are declared in XML. We used Android Studio's Layout Editor to construct an XML layout using a drag-and-drop interface. Then, we manipulated their properties programmatically in activity classes. We separate the UI of the app from the code that controls its behavior by declaring it in XML.

Every XML layout file is compiled into a View resource when we compile an app. In Activity.onCreate() callback method, we should load the layout resource. We can do it by calling setContentView(); we need to pass the reference to layout resource in the form: R.layout.layout_file_name.

Each View and ViewGroup object has its variety of XML attributes. Some of them are specific to a View; for example, TextView supports the fontFamily attribute. Some are common, for example, id attribute. And others are "layout parameters," which define View's location in layout.

```
<TextView
    android:id="@+id/toguzTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="80dp"
```

```
6     android:layout_marginLeft="80dp"
7     android:layout_marginTop="140dp"
8     android:text="@string/logo_toguz"
9     android:textSize="60sp"
10    app:fontFamily="serif"
11    app:layout_constraintStart_toStartOf="parent"
12    app:layout_constraintTop_toTopOf="@+id/nineTextView" />
```

Code 2.4: Example view

A view may have an integer ID to distinguish it within the layout tree. We can use it inside the code to create an instance of the view object and capture it from the layout.

Now, let us go through the activities and their layouts.

HomeActivity

In this activity, a user can choose the single or multiplayer mode or go to SettingActivity. The layout of this activity is shown in Figure 1.7.

Method onCreate(Bundle savedInstanceState) is executed when the system launches an activity. Every activity should implement this method. This method initializes the most important components of activity. Most importantly, we call setContentView() here, which defines the layout for the activity's UI. (Please note that throughout this section, when we introduce a method, we give its name and parameters, for example, onCreate(Bundle savedInstanceState), but in all other times when we refer to a method, we give its name and append opening and closing parentheses to indicate that it is a method, for example, onCreate().)

Inside the onCreate() method of HomeActivity, we call the setWindowStatus() method, which sets the status bar and navigation bar color to black opaque and makes them appear on top of the screen. This way, the app will use the whole screen even when status and navigation bars are displayed. Then we call the setWindowStatusHidden() method, which hides the status and navigation bars. These two methods are called in all activities. They are implemented in UtilMethods class. Inside onCreate() method, to specify the XML layout we call setContentView() by passing file's resource ID R.layout.activity_home to it. The

layout file is defined in the project's res/layout/activity_home.xml file.

Method onTouchEvent(MotionEvent event) is executed when a user swipes the screen. If the user swiped the screen's top edge, it makes status and navigation bars visible for 5 seconds and hides them back. The same method is implemented in all activities.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    int y = (int) event.getY();

    if (y < 20 && event.getAction() == MotionEvent.ACTION_DOWN) {
        setWindowStatus(getWindow());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        setWindowStatusHidden(getWindow());
    }
    return false;
}
```

Code 2.5: HomeActivity: onTouchEvent()

Method handleSingleBtn(View view) is executed when a user taps a button with the text "SINGLE PlAYER." It launches the SingleOptionsActivity. We can transition from one activity to another by calling the startActivity() method, passing an Intent object as an argument. The Intent object specifies either the exact activity we want to start or describes the type of action we want to perform. Also, an Intent object can transfer small quantities of data that can be used by the activity that is started.

```
public void handleSingleBtn(View view) {
    startActivity(new Intent(HomeActivity.this,
   SingleOptionsActivity.class));
}
```

Code 2.6: HomeActivity: handleSingleBtn()

Method handleMultiBtn(View view) is executed when a user taps a button with the text "MULTIPLAYER." It launches the MultiOptionsActivity.

Method handleSettingsBtn(View view) is executed when a user taps a button with the text "SETTINGS." It launches the SettingsActivity.

SettingsActivity

In this activity, a user can set the game settings in the single-player mode by choosing the level and informing if he wants to start the game. The layout of this activity is defined in the activity_settings.xml file. It is displayed in Figure 1.8.

Each time when activity is exited, it is destroyed. So all of its states is lost. To restore the activity's state when we launch it later, we need to save its state. To save a relatively small collection of key-values, we used the SharedPreferences APIs. A SharedPreferences object points to a file that contains key-value pairs and provides simple methods to store and retrieve them. The shared preferences file is stored on a device's disk.

Method onCreate(Bundle savedInstanceState) updates activity's UI with a previously saved state. If the shared preferences file is not created before, it makes the RadioGroup objects for settings to check the default value.

```
1 SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE)
    ;
2 int searchdepth = sharedPref.getInt(getString(R.string.
    single_searchdepth), res.getInteger(R.integer.
    single_searchdepth_easy_level));
```

Code 2.7: SettingsActivity: Read from SharedPreferences

Then the method sets listeners for both RadioGroup objects. When the RadioGroup object's checked value is changed, we update the corresponding key-value pair in the app's shared preferences file.

```
1 SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE)
    ;
2 SharedPreferences.Editor editor = sharedPref.edit();
```

```
editor.putInt(getString(R.string.single_searchdepth), level);
editor.commit();
```

Code 2.8: SettingsActivity: write to SharedPreferences

We can later read these values in GamePlayActivity or inside this activity when we start it again.

SingleOptionsActivity

The layout of this activity is defined in the activity_single_options.xml and looks like in Figure 2.12.

Figure 2.12: SingleOptionsActivity layout

Method handleBtns(View view) is called when either of the "NEW GAME" or "CONTINUE" buttons are tapped. It informs GamePlayActivity that the user wants to play the game in single-player mode and start a new game or continue from the previously left game depending on which button of this activity is tapped. It does it by putting key-value pairs to intent, which starts the GamePlayActivity.

```
public void handleBtns(View view) {
    Button btn = (Button) view;
    String option = btn.getTag().toString();

    Intent gamePlayIntent = new Intent(this, GamePlayActivity.class
   );
    gamePlayIntent.putExtra(getString(R.string.intent_key_mode),
   getString(R.string.intent_val_mode_single));
```

```
7       gamePlayIntent.putExtra(getString(R.string.intent_key_option),
        option);
8       startActivity(gamePlayIntent);
9   }
```

Code 2.9: SingleOptionsActivity: handleBtns()

MultiOptionsActivity

The layout of this activity is defined in the activity_multi_options.xml and displayed in Figure 1.9.

Method handleOption(View view) is executed when one of this activity's buttons is tapped. If the button "INTERNET RANDOM" is tapped, then it informs GamePlayActivity that the user wants to play the game in multiplayer mode with a random opponent over the internet and starts GamePlayActivity. If the button "INTERNET SPECIFIC" is tapped, then it starts MultiIntSpecOptionsActivity. If the button "LOCAL" is tapped, then it starts MultiLocalOptionsActivity.

MultiIntSpecOptionsActivity

The layout of this activity is defined in the activity_multi_int_spec_options.xml. It is shown in Figure 2.13.

Figure 2.13: MultiIntSpecOptionsActivity layout

Method handleChoice(View view) is executed when one of this activity's buttons is tapped. It informs the GamePlayActivity that the user wants to play the game in multiplayer mode with a specific opponent over the internet and whether she wants to create or join a game depending which button is tapped. Then, it starts GamePlayActivity.

MultiLocalOptionsActivity

The layout of this activity is defined in the activity_multi_local_options .xml. This activity looks exactly like SingleOptionsActivity (Figure 2.12).

Method handleChoice(View view) is executed when one of this activity's buttons is tapped. It informs the GamePlayActivity that the user wants to play the game in multiplayer mode locally and whether he wants to start a new game or continue. It starts GamePlayActivity.

GamePlayActivity

The layout of this activity is defined in the activity_gameplay.xml. This activity is displayed in Figure 2.14. It contains many widgets and nested layouts. We programmatically change the visibility of them.

Figure 2.14: GamePlayActivity layout

Method onCreate(Bundle savedInstanceState) creates an instance of the view objects and capture them from the layout to manipulate. It also creates

MediaPlayer objects to play them when a move is made during the game. Then, it reads game mode-related information sent by other activities from the intent that started this activity. At the end, it calls startGamePlayThread() method.

Method changeViewsVisibility(boolean showGamePlayViews) hides/shows game board views and shows/hides infoMsgTextView. infoMsgTextView is a TextView, where we display a message to the user, such as which player user is before the game starts or if the app cannot connect to the server.

Method holeClicked(View view) is executed when a user taps one of the hole ImageViews. It calls the inputHoleChoice() method, which sets the input instance field of this activity to the tapped hole number. The input is used in gameplay() methods.

Method soundBtnHandler(View view) is executed when the sound button is tapped. It changes the boolean soundOn instance field's value and changes the sound button's resource image to the appropriate one. When the soundOn field is set to true, we play sound after each move is made.

Method gameOverAlertDialog(String message) is called inside gameplay methods when the game is over. It creates an AlertDialog object that informs the user who won the game, the value of kazans, and provides home and exit buttons. The exit button exits the app, while the home button starts HomeActivity.

Method updateBoardUI() updates the widgets of the gameplay board. It changes the visibility of image views, which shows whose turn it is. It displays an animated sand watch image view only when the user is waiting for a computer/opponent over the internet to make a move. It updates TextView and ImageView of each hole. It makes the last moved hole's ImageView have a yellow background. It plays appropriate sound according to the last move if soundOn is set to true. It Changes the value of score label TextView and kazan ImageViews source images.

Method moveToMainActivity() calls saveSingle() or saveMultiState() methods depending on game mode. It stops gameplay thread and starts HomeActivity.

Method saveMultiState() saves the multiState instance field's state to the SharedPreferences. multiState is an object of type GameStateMultiPlayer, and it is initialized if the game mode is multiplayer.

Method readAssignSavedMultiState() restores the multiState object's state from SharedPreferences.

Method saveSingleState() saves the singleState instance field's state to the SharedPreferences. singleState is an object of type GameStateSingle, and it is initialized if the game mode is single-player.

Method readAssignSavedSingleState() restores the singleState object's state from SharedPreferences.

2.5 Testing

Testing is important in the software development process. By testing a program, we will notice bugs early in the development phase and ensure that it works as expected [11]. When we add new code to the program by running tests, we also guarantee that it does not break the program's already working functionality.

This project contains two folders in which tests are placed:

- The **androidTest** folder contains the tests that run on an emulator. They are integration tests where the JVM alone cannot verify the app's functionality. These tests need access to instrumentation information, like Context for the app.
- The **test** folder contains unit tests that run on local machines. Unit tests validate each class by checking if it's methods work as required.

2.5.1 Unit tests

We used Truth library and JUnit 4 to write unit tests. Truth is used to make assertions in tests. JUnit is a widely used testing framework for Java. JUnit 4 class contains test methods that are annotated by the @Test annotation. There are other annotations such as @Before and @After. When a method is annotated with @Before/@After, then it is executed before/after each test method. See the example test method in Code below:

```
@Test
public void testMove_BallsGained(){
    GameStateMultiPlayer gameState = new GameStateMultiPlayer(1);
    assertThat(gameState).isInstanceOf(GameStateMultiPlayer.class);

    gameState.makeMove(new Move(17));

    int[] expectedKazan = {10, 0};
    int[] expectedBoard = {10, 10, 10, 10, 10, 10, 10, 0, 9, 9, 9,
   9, 9, 9, 9, 9, 9, 1};
    int[] expectedTuz = {-1, -1};
    assertThat(gameState.kazan).isEqualTo(expectedKazan);
    assertThat(gameState.board).isEqualTo(expectedBoard);
    assertThat(gameState.tuz).isEqualTo(expectedTuz);
}
```

Code 2.10: GameStateMultiPlayerTest: testMove$_{B}allsGained()$

Follow these steps to run unit tests:

1. By clicking **Sync Project** in the toolbar, ensure that project is synchronized with Gradle.

2. You can run tests in one of the ways below:

   - Open the **Project** window, then right-click a test and click **Run** to run a single test.
   - Right-click a class or method in the test file and click **Run** to test all methods in a class.
   - Right-click on the folder and select **Run tests** to run all tests in a folder.

Test results are then displayed in the **Run** window. For a better illustration of steps, follow this link.

2.5.2 Instrumented unit tests

We used JUnit 4, Hamcrest library, and AndroidX Test APIs that include Espresso to write instrumented tests. Hamcrest is a library for writing flexible assertions in tests. Espresso enables us to perform UI interactions. See the example instrumented unit test class in Code below:

```
@RunWith(AndroidJUnit4.class)
public class HomeActivityTest {

    private static final String PACKAGE_NAME = "com.nurbiike.tk";

    //to test an activity we use @Rule
    //ActivityTestRule enables lunching of the activity
    //Note: ActivityTestRule was not imported automatically by the IDE. Steps needed:
    //  added       androidTestImplementation 'com.android.support.test:rules:1.0.2'    to app/build.gradle
    //  added       import androidx.test.rule.ActivityTestRule;    to this file
    //  synced the project with the gradle files by clicking the sync button
    @Rule
    public ActivityTestRule<HomeActivity> mActivityTestRule = new ActivityTestRule<HomeActivity>(HomeActivity.class);

    //a reference to HomeActivity
    private HomeActivity mHomeActivity = null;

    //this is used to set the necessary preconditions prior to execution of @Test
    @Before
    public void setUp() throws Exception {
        mHomeActivity = mActivityTestRule.getActivity();
        Intents.init();
    }
```

```
24
25      @Test
26      public void testLaunch() {
27          View vNineTextView = mHomeActivity.findViewById(R.id.
    nineTextView);
28          View vToguzTextView = mHomeActivity.findViewById(R.id.
    toguzTextView);
29          View vKorgoolTextView = mHomeActivity.findViewById(R.id.
    korgoolTextView);
30          View vSinglePBtn = mHomeActivity.findViewById(R.id.
    singlePBtn);
31          View vMultiPBtn = mHomeActivity.findViewById(R.id.multiPBtn
    );
32          View vSettingsBtn = mHomeActivity.findViewById(R.id.
    settingsBtn);
33          assertNotNull(vNineTextView);
34          assertNotNull(vToguzTextView);
35          assertNotNull(vKorgoolTextView);
36          assertNotNull(vSinglePBtn);
37          assertNotNull(vMultiPBtn);
38          assertNotNull(vSettingsBtn);
39      }
40
41      @Test
42      public void verifySinglePBtn(){
43          // Click Single Player button to intent to
    SingleOptionsActivity
44          onView(withId(R.id.singlePBtn)).perform(click());
45
46          // Verifies that the SingleOptionsActivity received an
    intent
47          // with the correct package
48          intended(allOf(
49                  hasComponent(hasShortClassName(".ui.
    SingleOptionsActivity")),
50                  toPackage(PACKAGE_NAME)));
51      }
52
53      @Test
```

```
54      public void verifyMultiPBtn(){...}
55
56      @Test
57      public void verifySettingsBtn(){...}
58
59      @Test
60      public void useAppContext() {
61          // Context of the app under test.
62          Context appContext = InstrumentationRegistry.
    getInstrumentation().getTargetContext();
63
64          assertEquals("com.nurbiike.tk", appContext.getPackageName()
    );
65      }
66
67      //this is used to cleanup after the execution of @Test
68      @After
69      public void tearDown() throws Exception {
70          Intents.release();
71      }
72  }
```

Code 2.11: HomeActivityTest

The way to run instrumented unit tests is similar to running unit tests. Test results are also displayed in the **Run** window.

In this chapter, we have finished the implementation and testing phase. In the next one, we will discuss possible improvements to it.

Chapter 3

Future Work

There is always room for improvement. Firstly, we can host the server in the cloud to make it up all the time. I currently can run it locally and provide my device's IP address in the mobile app's code.

We can also make a mobile app to support different screen sizes by providing alternative resource files and configure the appropriate resource folder according to an Android device's screen size. In an Android app, its resources and source code are separate. We can also support several languages.

Furthermore, we can improve the searching algorithm that finds the best move for a computer player from the current game state. The algorithm is predictable now. For one game state, it always gives the same move. We might add some randomness to the algorithm, such as giving the second or third best move sometimes. Currently, the value() function decides the goodness of the board state, taking into account only values of kazans. The more balls in the player's kazan, the better the board state for him. However, we can consult the Toguz Korgool expert to improve this function because tuz hole has strategic importance in the game. There might other points that I am not aware of.

Bibliography

[1] Arty Sandler. *Toguz Kumalak*. URL: http://www.iggamecenter.com/info/en/toguzkumalak.html (visited on 11/15/2020).

[2] *Mancala*. URL: https://en.wikipedia.org/wiki/Mancala (visited on 11/15/2020).

[3] *Run Android app on a real device*. URL: https://developer.android.com/training/basics/firstapp/running-app#RealDevice (visited on 12/17/2020).

[4] *Run Android app on an emulator*. URL: https://developer.android.com/training/basics/firstapp/running-app#Emulator (visited on 12/17/2020).

[5] Peter Norvig Stuart J. Russell. *Artificial Intelligence: A Modern Approach*. Third Edition. Pearson Education, 2010, pp. 161–169. ISBN: 9780136042594.

[6] Sebastian Lague. *Algorithms Explained – minimax and alpha-beta pruning*. URL: https://www.youtube.com/watch?v=l-hh51ncgDI&ab_channel=SebastianLague (visited on 11/24/2020).

[7] Mina Krivokuća. *Minimax with Alpha-Beta Pruning in Python*. URL: https://stackabuse.com/minimax-and-alpha-beta-pruning-in-python/ (visited on 10/02/2020).

[8] *Minimax*. URL: https://en.wikipedia.org/wiki/Minimax (visited on 10/02/2020).

[9] *Alpha–beta pruning*. URL: https://en.wikipedia.org/wiki/Alpha-beta_pruning (visited on 10/02/2020).

[10] *Android developer guides*. URL: https://developer.android.com/guide (visited on 12/13/2020).

[11] *Test apps on Android*. URL: https://developer.android.com/training/testing (visited on 12/28/2020).

List of Figures

List of Codes