COMP 424: Project Code Description

Project TA: Kian Kenyon-Dean (kian.kenyon-dean@mail.mcgill.ca)
Code repository: `https://github.com/kiankd/comp424`

1 Overview

We provide a full implementation of Tablut. Player games requires: a server (the real board) to be launched, and then two clients (agents) to connect to the server via TCP sockets; this allows for clients to be run on separate computers. We also provide a GUI which displays the game in a nice and intuitive way for learning Tablut[1]. Note that games between agents *can be played without the GUI to increase speed.*

All source code is found in the **src** directory. **You must NOT edit any packages**, other than **student_player**, which can be modified as much as you want so long as it (1) compiles, and, (2) that you do not use any external libraries, but you can use default internal Java libraries (like Math and Util). If you edit other packages you must be aware that your edits will not be used during the tournament; e.g., so if you know what you're doing, you may want to edit **Autoplay.java** to run many test games of your agent against another agent (or perhaps against itself).

Below I provide a full elaboration of the source code, and identify all the public methods that should be useful for you. If you inspect the source code, there are some methods that I do not identify below, and others are private; trust my judgment as the designer for why certain methods are private. If you think something should be public, rewrite it yourself in the **student_player** package.

```
src
|-- autoplay                                Autoplays games; can be ignored.
|
|-- boardgame                               Package for implementing boardgames, provides
|                                            useful software infrastructure for logging,
|                                            GUI, and server TCP protocol. Primarily, it
|                                            abstracts game logic, but can be ignored for
|                                            this project.
|
|-- coordinates                             Simple package for using board positions.
|  |
|  |-- Coord.java                           Class for representing a single coordinate on
|  |                                         a game board. Note that x and y are somewhat
|  |                                         misleading parameter names, x really means
|  |                                         'row', and y really means 'column'.
|  |
|  |  |-- List getCoordsBetween(Coord c)   Returns a list of all the coords between the
|  |  |                                      current coord and c.
|  |  |
|  |  |-- int maxDifference(Coord c)       Returns the maximum difference between a Coord
|  |  |                                      and c's x and y values.
|  |  |
```

[1] I highly recommend playing the game as a HumanPlayer against a GreedyPlayer at least 5 times as both a *Muscovite* and a *Swede*; this will give you a good understanding of how the game works and its complexity.

```
|  |  |-- int distance(Coord c)            Returns the manhattan distance between the
|  |  |                                     current coord and c.
|  |
|  |-- Coordinates.java                    Static class that deals with all Coords. Note
|  |                                        that we use the flyweight design pattern with
|  |                                        Coords. Namely, you CANNOT construct a new
|  |                                        Coord. Rather, you 'get' the Coord with this
|  |                                        static class. This allows for very efficient
|  |                                        use of memory, letting you avoid expensive
|  |                                        operations so as to focus time on fast AI.
|  |  |
|  |  |-- Coord get(int i, int j)          Returns a Coord object corresponding to the
|  |  |                                     position at row i, column j, on the board.
|  |  |                                     If the position i, j does not exist on the
|  |  |                                     board, then this will throw an IndexError.
|  |  |
|  |  |-- boolean isCorner(Coord c)        Returns true if c is a corner, false else.
|  |  |
|  |  |-- boolean isCenter(Coord c)        Returns true if c is the center, false else.
|  |  |
|  |  |-- boolean isCenterOrNeighborCenter(Coord c)  Returns true if c is the center or
|  |  |                                               neighbors center, useful for
|  |  |                                               checking king protection rules.
|  |  |
|  |  |-- int distanceToClosestCorner(Coord c)       Returns the manhattan distance to
|  |  |                                               the corner closest to c.
|  |  |
|  |  |-- Coord getSandwichCoord(Coord f, Coord m)   Returns a Coord, if it exists,
|  |  |                                               corresponding to the coordinate
|  |  |                                               with which f could sandwhich m.
|  |  |
|  |  |-- List getCorners()                Returns a list of all corner Coords.
|  |  |
|  |  |-- List getNeighbors(Coord c)       Returns a list of all Coords neighboring c.
|  |  |
|  |  |-- Iterable iterCoordinates()       Very useful function. Iterates over all
|  |  |                                     coordinates on the board. Very nice since it
|  |  |                                     allows us to do for-each loops clearly.
|
|-- student_player                         Package for YOU!
|  |
|  |-- StudentPlayer.java                  The class you will implement your AI within.
|  |
|  |-- MyTools.java                        Placeholder for any extra code you may need.
|
|-- tablut                                 The package implementing all game logic.
|  |
|  |-- TablutBoardPanel.java               Implements the GUI, can be ignored.
|  |
|  |-- TablutBoard.java                    Used for server logic, can be ignored.
|  |
```

```
|  |-- TablutMove.java                    A move object for Tablut! Relevant functions:
|  |  |
|  |  |-- Coord getStartPosition()        Returns the start Coord of this Move.
|  |  |
|  |  |-- Coord getEndPosition()          Returns the end Coord of this Move.
|  |
|  |-- TablutBoardState.java              Implements all game logic, most important.
|  |  |                                    Note that a TablutBoardState (TBS) manages
|  |  |                                    logic concerning whose turn it is, rules,
|  |  |                                    and the positions of all pieces.
|  |  |
|  |  |-- Enum Piece                      Four types: BLACK, WHITE, KING, EMPTY.
|  |  |
|  |  |-- Object clone()                  Clones the TBS, allowing manipulation of the
|  |  |                                    clone without affecting the original. Must
|  |  |                                    be typecasted to a TBS to work on the clone.
|  |  |
|  |  |-- List getAllLegalMoves()         Returns all legal moves for the current player
|  |  |                                    FROM THE PERSPECTIVE OF THE TBS.
|  |  |
|  |  |-- List getAllLegalMovesForPosition(Coord c)   Returns all legal moves starting
|  |  |                                                from current position, for the
|  |  |                                                current player FROM THE PERSPECT-
|  |  |                                                IVE OF THE TBS.
|  |  |
|  |  |-- boolean isLegal(TablutMove m)   Returns whether or not m is a legal move.
|  |  |
|  |  |-- void processMove(TablutMove m)  Processes m on the board and updates the TBS
|  |  |                                    as if the move was played. This allows you
|  |  |                                    to project and determine the impact of moves.
|  |  |
|  |  |-- HashSet getPlayerPieceCoordinates()        Returns all Coords of pieces for
|  |  |                                               the current player on the TBS.
|  |  |
|  |  |-- HashSet getOpponentPieceCoordinates()      Returns all Coords of pieces for
|  |  |                                               the current opponent on the TBS.
|  |  |
|  |  |-- Coord getKingPosition()         Returns Coord of where King is at on board.
|  |  |
|  |  |-- Piece getPieceAt(Coord c)       Returns the Piece at c on the board.
|  |  |
|  |  |-- boolean isOpponentPieceAt(c)    Returns if an opponent piece is at Coord c.
|  |  |
|  |  |-- boolean coordIsEmpty(c)         Returns if the Coord c is empty on the board.
|  |  |
|  |  |-- int getOpponent()               Returns opponent of current player on TBS.
|  |  |
|  |  |-- int getNumberPlayerPieces(int player)      Returns number of pieces for the
|  |  |                                               player passed.
|  |
|  |-- TablutPlayer.java                  Abstract class that all players extend.
```

```
|  |
|  |-- GreedyTablutPlayer.java            Example greedy player code - review it.
|  |
|  |-- RandomTablutPlayer.java            A totally random player, easy to beat.
|
```

2 Coding Tips

The TablutBoardState (TBS) you are given to work with in your ChooseMove method is a copy of the real TBS, so you can modify it without worrying about changing the real TBS. However, you should clone the TBS and then process moves on the cloned TBS to assess the impact of doing different moves on the TBS; note that a cloned TBS that processed a move can also be cloned again.

When you want to work with any Coordinates on the board, you must use the static method *Coordinates.get(i, j)* to get the Coord corresponding to point *(i, j)* on the board. You cannot construct new Coord objects, as that would lead to unnecessary memory inefficiencies, and also would not allow us to Hash Coords using their memory addresses. I wrote the code with this design pattern since it will improve efficiency and allow your algorithm to have more time for "thinking", rather than data processing.

Whenever you have a *List* or *HashSet* of Coords **do not ever call *.clear()***. This will destroy the flyweight Coord objects and will cause your algorithm to crash, making you lose the game.

3 Workflow: Eclipse

We strongly recommend that you develop your agent using Eclipse. If you use something else, you have to figure out how to compile and run the project yourself.

The root directory of the project package is a valid Eclipse project. There are two ways to get it running, based on the following clicks:

- From Github: **File** → **Import** → **Git** → **Projects from Git** → **Clone URI** → put in field URI `https://github.com/kiankd/comp424` → keep pressing **Next** until the end!
- From source: **File** → **Import** → **General** → **Existing projects into workspace** and then select the root of the project package.

You may need to set the launch configurations manually as follows: **File** → **Import** → **Run/Debug** → **Launch configurations** and then navigate to the **eclipse** directory and select those launches.

4 Quick Start Tablut with Eclipse

If using eclipse and you've set it up as described above, you can easily start a game. Simply click **Run** in Eclipse, then click on GUI. This will launch the GUI for you. Then, click the launch tab in the GUI, and select **Launch Server**. Then, the first client you launch will be the *Muscovites*; for example, select **Launch Human Player** next. Then, to set the *Swedes*, select the next client class; for example, select **Launch Client (tablut.GreedyTablutPlayer)**. You can now play against a basic AI in the GUI to get a feel for the game. Changing the Clients you select will determine who/what plays who/what – so you can run two humans against each other too, and similarly two agents against each other.

5 Playing Games

Here we provide documentation for the server and client programs. The commands outlined in this section are the commands that are called by the provided build.xml file (if using ant) and launch configurations (if

using Eclipse). The details provided in this section are especially useful for advanced use of the provided code, such as playing games where the clients are located on different machines than the server. All of these commands assume that you have compiled the code and stored the class files in a directory named bin located inside the root directory of the project package (Eclipse will automatically do these things).

In a nutshell, if you want to run from the command line, you will need three terminal shells open and simultaneously running: one to launch the server, one to launch a Client for the *Muscovites* player, and one to launch a Client for the *Swedes* player.

5.1 Launching the Server

To start the server from the root folder of the project package, run the command:

```
java -cp bin boardgame.Server [-p port] [-ng] [-q] [-t n] [-ft n] [-k]
```

where:

- (-p) port sets the TCP port to listen on. (default=8123)
- (-ng) suppresses display of the GUI
- (-q) indicates not to dump log to console.
- (-t n) sets the timeout to n milliseconds. (default=700)
- (-ft n) sets the

  rst move timeout to n milliseconds. (default=30000) (-k) launch a new server every time a game ends (used to run multiple games without the GUI)

For example, assuming the current directory is the root directory of the project package, the command:

```
java -cp bin boardgame.Server -p 8123 -t 300000
```

launches a server on port 8123 (the default TCP port) with the GUI displayed and a timeout of 300 seconds.

The server waits for two clients to connect. Closing the GUI window will not terminate the server; the server exits once the game is finished. If the -k arg was passed, then a new server starts up and waits for connections as soon as the previous one exits. Log files for each game are automatically written to the logs subdirectory. The log file for a game contains a list of all moves, names of the two players that participated, and other parameters. The server also maintains a file, outcomes.txt, which stores a summary of all game results. At present this consists of the integer game sequence number, the name of each player, the color and name of the winning player, the number of moves, and the name of the log file.

5.2 Launching a Client

As stated previously, the server waits for two client players to connect before starting the game. If using the GUI, one can launch clients (which will run on the same machine as the server) from the Launch menu. This starts a regular client running in a background thread, which plays using the selected player class. In order to play a game of Tablut using the GUI, choose **Launch human player** from the **Launch** menu (as described in Section 4).

Clients can also be launched from the command line. From the root directory of the project package, run the command:

```
java -cp bin boardgame.Client [playerClass [serverName [serverPort]]]
```

where:

- playerClass is the player to be run (default=tablut.RandomTablutPlayer)

- serverName is the server address (default=localhost)
- serverPort is the port number (default=8123)

For example, the command:

```
java -cp bin boardgame.Client tablut.RandomTablutPlayer localhost 8123
```

launches a client containing the random Tablut player, connecting to a server on the local machine using the default TCP port. The game starts immediately once two clients are connected.

The two approaches of launching players from the GUI and launching players from the command line can also be combined. For instance, one can use the GUI to manually play against the random player (or any other agent) by first launching a human player using the GUI, and subsequently launching the random player, either by selecting it in the GUI or running the appropriate command from the command line. The order can also be switched; the player that connects to the server first plays as *Muscovites* and moves first.

6 Autoplay

The provided Autoplay script can be used to play a large batch of games between two agents automatically. It launches a Server with the options **-k -ng**, and then repeatedly launches pairs of agents to play against one another. To use Autoplay, run the following command from the root directory of the project package:

```
java -cp bin autoplay.Autoplay n_games
```

where **n_games** is a positive integer number of games to play. The default behaviour of Autoplay is to play the student agent against the random player, with the random player agent going first every second game. You can modify this behaviour by editing **Autoplay.java**.

7 Implementing a Player

New agents are created by extending the class **tablut.TablutPlayer**. The skeleton for a new agent is provided by the class **student_player.StudentPlayer**, and you should proceed by directly modifying that file. In implementing your agent, you have two primary responsibilities:

1. Change the constructor of the **StudentPlayer** class so that it calls **super** with your student number.
2. Change the code in the method **chooseMove** to implement your agent's strategy for choosing moves (the real work).

To launch your agent from the command line, run the command:

```
java -cp bin boardgame.Client student_player.StudentPlayer
```

You can also launch your agent by selecting it from the Launch menu in the GUI. The project specification has further details on implementing your player, and, in particular, what you need to submit.