Navigation

Project 2: Tablut

"*Tafl emk ǫrr at efla....*" Jarl Rognvald Kali Kolsson

[Revisions to the project spec since its release are underlined.]

Introduction

Tablut is one of the family of *tafl* (or *hnefatafl*) games: Nordic and Celtic strategy games played on checkered boards between two asymmetric armies. These are ancient games, possibly related to the Roman game *ludus latrunculorum*, that were generally displaced by chess in the 12th century. Tablut is a variant from Lapland (the northernmost region of Finland). The detailed rules of these games are generally disputed; tablut, having survived into the 1700s, is perhaps best documented (by none other than Carl Linnaeus, the fellow responsible for our state animal being known as *Ursus arctos californicus* and our state trees as *Sequoia sempervirens* and *Sequoiadendron giganteum* in scientific circles). However, his account is itself incomplete and additional confusion resulted from the later mistranslation of his manuscript from Latin. Therefore, any of you familiar with modern reconstructions of the game should not expect that our version will be identical to yours.

Tablut is played on a 9x9 checkerboard between a set of 9 white pieces and 16 black pieces. The middle square is called the *throne* (or castle). One of the white pieces is the king, and the others, his guards, are known as *Swedes*. The white side wins if the king reaches one of the edges of the board. The black pieces are known as *Muscovites* (referring to the Grand Duchy of Moscow). Their aim is to capture the king before he reaches the edge of the board.

Navigation

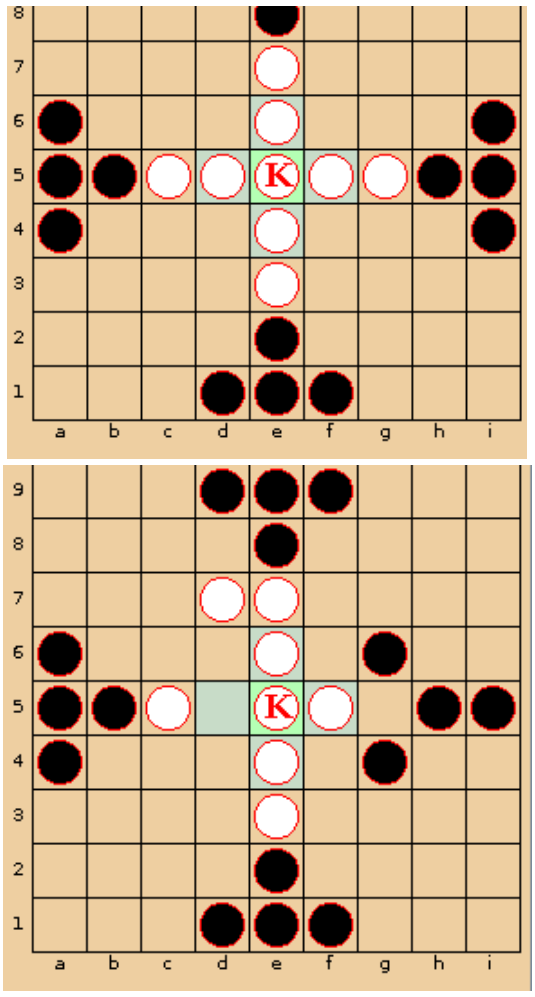

Figure 1. On the left: a Tablut board showing the standard numbering of squares, and the initial placement of the pieces. On the right: the board that results after the three moves i6-g, d5-7, i4-g (the last move is a capture).

All pieces move like chess rooks: any number of squares orthogonally (horizontally or vertically). Pieces may not jump over each other or land on top of another piece. No piece other than the king may land on the throne, although any piece may pass through it when it is empty. The black (Muscovite) side goes first.

A piece other than the king is captured when, as a result of an enemy move to an orthogonally adjacent square, the piece is enclosed on two opposite sides (again orthogonally) by *hostile squares*. A square is hostile if it contains an enemy piece, or if it is the throne square and is empty (that is, it is hostile to both white and black pieces). The occupied throne is also hostile to white pieces when three of the four squares surrounding it are occupied by black pieces. Captures

land between two enemy pieces without being captured. A single move can capture up to three pieces.

The king is captured like other pieces except when he is on the throne square or on one of the four squares orthogonally adjacent to the throne. In that case, the king is captured only when surrounded on all four sides by hostile squares (of which the empty throne may be one).

A side also loses when it has no legal moves on its turn, or if its move returns the board to a previous position (same pieces in the same places and the same side to move). As a result, there are no drawn games.

Navigation

Notation

A square is denoted by a column letter followed by a row number (as in e4). Columns are enumerated from left to right with letters a through i. Rows are enumerated from the bottom to the top with numbers 1 through 9. An entire move then consists of the starting square, a hyphen, and the ending row (if vertical) or column (if horizontal). Thus, `b3-6` means "Move from b3 to b6" and `b3-f` means "Move from b3 to f3."

Commands

When running from the command line, the program will accept the following commands, which may be preceded by whitespace.

- **new**: End any game in progress, clear the board to its initial position, and set the current player to black.
- A move in the format described in Notation.
- **seed** *N*: If the AIs are using random numbers for move selection, this command seeds their random-number generator with the integer *N*. Given the same seed and the same opposing moves, an AI should always make the same moves. This feature makes games reproducible.

black or white, case-insensitive.

Navigation

- **manual** *C*: Make the *C* player a human player (entering moves as manual commands).
- **limit** *N*: Make *N* be the maximum number of moves that a player may make during the current game. A player loses if the game has not ended by the time he must make his $(N+1)$st move. The command is in error if either player has already made at least *N* moves. The move limit is removed by a **new** command. (A move limit is not a normal provision of Tablut; we've added it for testing convenience so that we can test whether a program can find a forced win within a given number of moves.)
- **dump**: Print the current state of the board in ***exactly*** the following format:

```
===
 - - - B B B - - -
 - - - - B - - - -
 - - - - W - - - -
 B - - - W - - - B
 B B W W K W W B B
 B - - - W - - - B
 - - - - W - - - -
 - - - - B - - - -
 - - - B B B - - -
===
```

  Here, `K` denotes the king, `W` another white piece (Swede) and `B` a black piece (Muscovite). You must not use the `===` lines for any other output).
- **quit**: Exit the program.

Feel free any other commands you think might be nice.

Output

When an AI plays, it should print out each move that it makes using exactly the format

```
* a1-4
```

player's moves.

When one side wins, the program should print out one of

```
* White wins.
* Black wins.
```

(also with periods) as appropriate. Do not use the * character in any other output you produce.

You may prompt a manual player for input using the form

```
...>
```

where "..." may be any text. The grading scripts will discard any text from the beginning of a line up to a > character.

Your Task

Your job is to write a program to play Tablut. To run it in text mode, use the command

```
java -ea tablut.Main
```

to enter commands from the terminal or use

```
java -ea tablut.Main INPUT
```

to feed it commands from file INPUT.

The AI in your program should be capable of finding a win that is within 4 moves. Experiment a bit to see what works. The autograder will allow 3 minutes for a fully automated game.

The GUI is an optional (extra credit) part of this project. We will actually do automatic testing only on the commands

```
java -ea tablut.Main
```

and

```
java -ea tablut.Main INPUT
```

Navigation

Staff Program

The staff-tablut program on the instructional machines runs our solution to the project. This version has additional bells and whistles that you are ***not*** required to duplicate. It is ***not*** the standard for this project, just as example of a solution. In particular, your GUI (if you do it) need not look anything like ours. The autograder will use this program in some of its tests to check your program's output.

Testing

We have only provided a token UnitTest file; you can add additional unit test files and list them in UnitTest so that they all get run by

```
java -ea tablut.UnitTest
```

(which is what make check does).

We have provided two simple integration tests in the testing subdirectory. They are certainly not adequate to test your program. Be warned, we will not run the autograder until near the deadline; ***DO NOT RELY ON THE AUTOGRADER FOR YOUR TESTING!***

The integration test program, test-tablut, feeds commands to one or two running programs and passes appropriate moves from one to the other, allowing you to test your program and to test it against another program (such as the staff version). To run test-tablut, you'll use

```
python3 testing/test-tablut TESTFILE-1.in
```

to run TESTFILE-1.in through your program and

```
python3 testing/test-tablut TESTFILE-1.in TESTFILE-2.in
```

to run two programs simultaneously so that each one sends all of its AI's moves (such as "* i4-g" as described previously) to the other program. (Replace "TESTFILE" with the actual name of your test file.)

Navigation

running a program preceded by "#*", such as

```
#* java -ea tablut.Main
```

(You will probably use just the command command above; the autograder will sometimes replace it with an invocation of staff-tablut.) The rest of the .in file is fed to this program as the standard input, except for lines that start with "#*" in the first column, which are special instructions to the testing script.

- The command #*time *MOVE GAME* puts a time limit of *MOVE* seconds on each move in a game and *GAME* seconds for one side's moves in an entire game (i.e., an entire sequence of moves controlled by one of the move/win commands below).
- The command #*move means "wait for the program to output an AI move, and then continue with the script." When used with the two-argument form of test-tablut, it also sends this move as input to the other program.
- The command #*move/win is intended for use when both players are AIs, and means "wait for the program to output a complete sequence of AI moves, followed by "* ... wins." It does not print either the moves or the "win" message.
- The command #*move/win+ is the same as #*move/win, but also prints the "* ... wins." message.
- The command #*win+ waits for a "* ... wins." message from the program and prints it.
- All lines that don't start with #* are sent to the program being tested.

A few other commands apply only to the two-argument form of test-tablut. They are intended to allow two programs to play each other.

- The command #*remote move/win means "Wait for an AI move from the other program, give it to this program, then execute a #*move command. Repeat until one side sends a win message. Do not print the moves or win message.
- The command #*remote move/win+ is the same as #*remote move/win, but prints the "win" message.

will, at a certain point, contain the commands

```
#*move
#*remote move/win
```

and the other will contain

```
#*remote move/win
```

so that the first sends a move from its AI to the other program, which then waits for a response from its AI to send back, and so forth.

For the remote commands, both programs should generate "wins" messages, and test-tablut will check that they are the same.

The test-tablut script throws out any other output from either program except for properly formatted board dumps, as are supposed to be produced by the **dump** command described previously. You can see all the output by running it with

```
python3 testing/test-tablut --verbose TESTFILE-1.in
```

or

```
python3 testing/test-tablut --verbose TESTFILE-1.in TESTFILE-2.in
```

which will show all the commands sent to each program and all their output.

The test-tablut program will report an error if a program hangs or times out, or if it exits abnormally (with an exception or an exit code other than 0). Finally, if there is a file TESTFILE-1.std or TESTFILE-2.std, make check will check it against the output from the program for TESTFILE-1.in (likewise for TESTFILE-2.std against the output for TESTFILE-2.in); make check uses on the script testing/tester.py to do this comparison.

Checkpoint

By **Monday, November 11 (11/11)**, if your implementation is able to pass a staff integration test, you will receive **3 extra points** . The test will cover the following:

Navigation

2. Board initialization;
3. Non-capturing moves;
4. Properly handling illegal moves (i.e., not changing the board contents).

Not all of this behavior is covered in the provided tests, so you are encouraged to write your own. Submit your checkpoint in the proj2 directory as proj2a.

Extra Credit

First get your program working, and then, if you feel the urge, try the extra-credit GUI (Graphical User Interface). If you do, the option

```
java -ea tablut.Main --display
```

should create the GUI. If you don't implement the GUI, this option should cause your program to exit with a non-zero code via System.exit(2). Your GUI does ***not*** have to look at all like ours. This will be worth up to 3 points.

Advice

Your final work must be your own, but especially on this project, feel free to get together with other students to discuss ideas and plan strategies. Of course, you should always feel free to consult your TA or me.

The board is an obvious place to start. We have provided suggestions for methods that you can use if you want, but you are not required to do so. We have structured the skeleton so that the different kinds of player (ordinary human at the keyboard using text commands, AI, or human using a GUI) are represented as different subtypes of a type Player, an example of how using OOP can cut down on pervasive conditional tests for types of player.