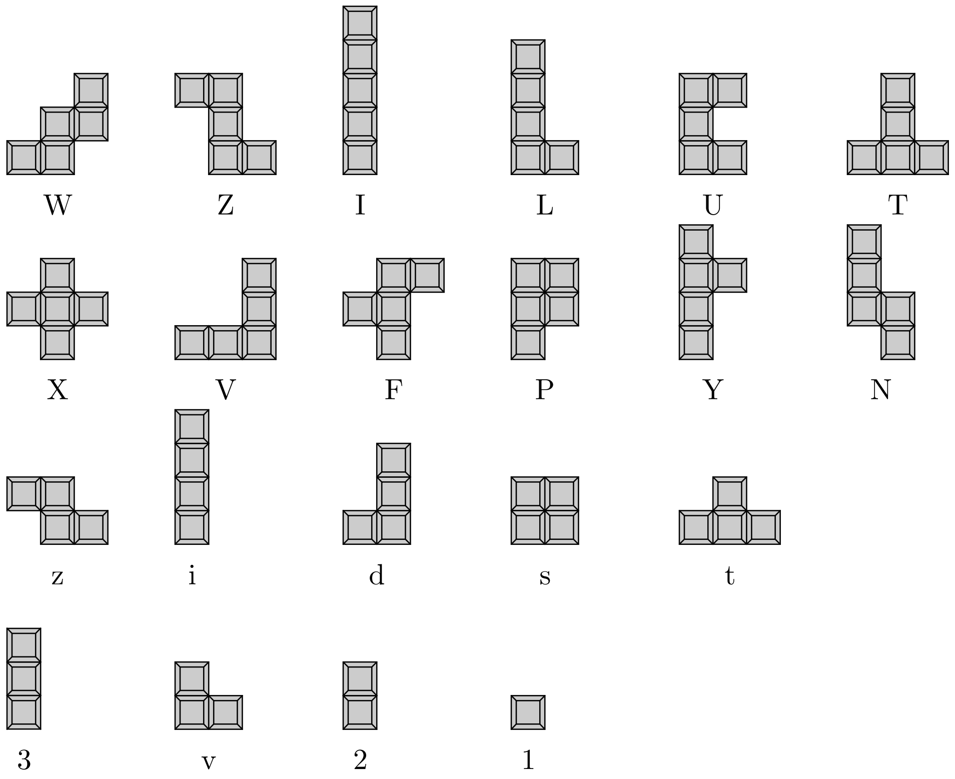
CS61B, Fall 2011 **Project #2: Distributed Game** **P. N. Hilfinger**

Due: Wednesday, 2 November at 2400

1 Background

Blokus Duo® is a popular two-person board game available in both on-line and physical versions (see also the official website). The players attempt to tile a square 14×14 board with a set of polyominoes under certain restrictions. The winner is the player who manages to tile the most territory, with bonus points being awarded in some circumstances. Either or both players may be AIs (artificial intelligences), which you get to create. We'll hold a contest for best automated player.

Figure 1: The pieces, with their names. Each player (orange and violet) gets one set of pieces. Each piece may be played either as shown, or rotated through 90, 180, or 270 degrees, and may be flipped over (reflected).



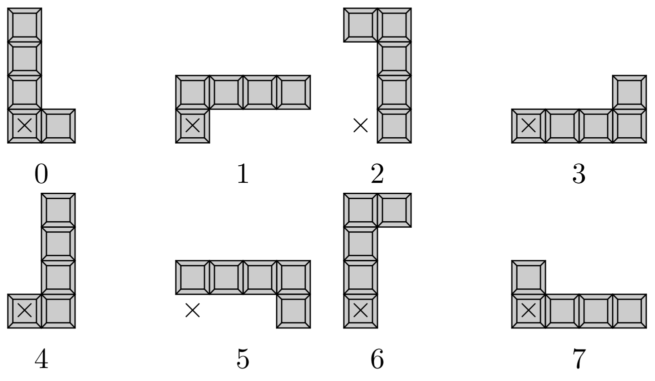


Figure 2: Possible piece orientations, with their labels. The symmetries of most pieces will causes several of the orientations to be the same. In these cases, we use the smallest orientation number. The "×" symbols mark the locations of the *reference point* for each figure—the lower-left of the smallest enclosing rectangle.

1.1 The pieces

Each player gets 21 pieces, as shown in Figure 1. Each piece may be placed on the board in any of eight orientations, as shown in Figure 2. The first orientation, which we'll call *orientation 0*, is always the one shown in Figure 1. Orientation 1 is the same, rotated clockwise 90°; orientation 2 is rotated 180°; and orientation 3 is rotated 270°. Orientations 4–7 result from flipping the piece in orientations 0–3 around the vertical axis.

1.2 The moves

Players alternate, with orange having the first move. A player moves by placing one of his remaining pieces on the board (with each of the squares comprising the piece covering one square of the board) in any of the possible orientations, subject to the following rules:

1. On the first move, a player must place the piece so that it covers one of the four corner squares.
2. On each subsequent move, a player may place the piece over any unoccupied squares such that it touches an existing piece of the same color on the board at (at least) one corner of a square.
3. Pieces of the same color may only touch at their corners; they may not contact each other along an edge of one of their squares. There is no restriction on touching (but not, of course, covering) pieces of other colors.

For example, the first and second configurations in Figure 3 are legal moves, which the last two are not.

1.3 Scoring

The game ends when the player whose move it is has no valid move. At this point, each player receives one point for each square on the board that is covered by that player's color.

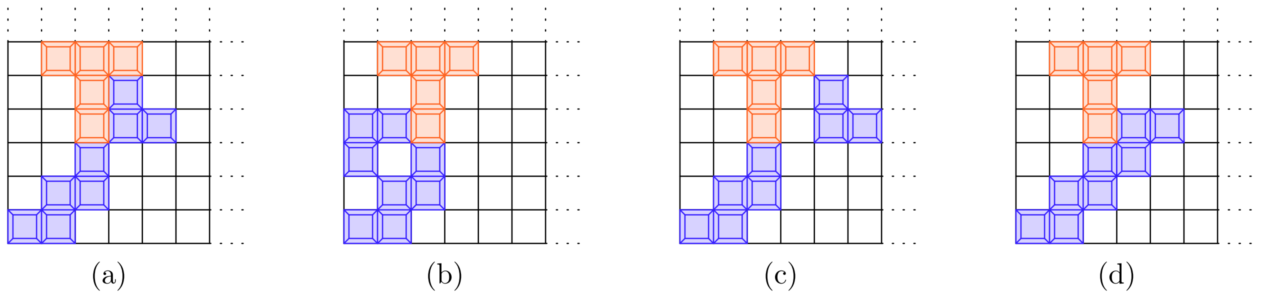


Figure 3: Examples of legal (a and b) and illegal (c and d) moves. In each case, there is a W piece in the lower-left corner, an enemy (orange) T piece on top of it, and we are placing a triomino. In case (c), the new piece does not touch any corner of the existing W piece. In case (d), it contacts the W piece along the edge of at least one square. In all cases, the enemy piece is irrelevant to the legality of the move, as long as it is not overlapped.

A player who plays all the pieces and places the monomino (single-squared piece) last gets an additional 5 points[1]. The winner is the player receiving the most points; ties are possible.

1.4 Notation

So far, there doesn't seem to be a single standard notation for Blokus® moves, so we'll just roll our own.

We'll denote rows and columns with lower-case hexadecimal digits (0–9 plus the letters a–d), numbering from the lower-left corner. We'll denote pieces by the 1-character names shown in Figure 1. Finally, we'll denote the eight piece orientations by the digits 0–7, as described in Figure 2. When multiple orientations of a piece produce the same shape, we will always use the smallest.

A move has the form *pcrd*, where *p* is a piece, *c* and *r* are the column and row of its *reference point,* and *d* is its orientation. The reference point of a piece is the location of the square that would be at the lower left corner of the smallest rectangle enclosing the piece (whether or not the piece happens to have a square at that point). For example, the three pieces in Figure 3a would be placed by the moves `W000`, `T132`, and `v330`.

2 Textual Input Language

Each execution of the program will play a single game, depending on the command-line arguments (see §4). When the user is supposed to input a move on the standard input, it should consist of a single line using the notation described in §1.4. At any time, the user may also enter any line beginning with '`b`', which should cause the program to print the current state of the board in format described in §3. Any line beginning with '`q`' should cause the program to exit, abandoning the current game. The end of file on the standard input should have the same effect.

[1]The official rules call for a 15-point bonus for placing all the pieces, but this has no effect on the winner, so I have simplified the scoring procedure.

As long as the commands described so far work properly, you may add any additional commands you want.

3 Textual output

This time, you can prompt however you want, and print out whatever user-friendly output you wish (other than debugging output or Java exception tracebacks, that is), subject to a few constraints, which we include to allow our testing software to understand your output. It's nice, for example, to print out the opponent's move, or perhaps a picture of the board. When users enter erroneous input, you should print an error message on the standard error output (`System.err` in Java), and the input should have no effect (and in particular, the user should be able to continue entering commands after an error). Never produce output on the standard error output unless there is erroneous input.

When one or both of the players is an AI, your program must print their moves as they make them, again using the notation from §sec:notation. Unless a game ends prematurely, your program must print a message after receiving the final move announcing the winner, and having the format

```
Orange wins (89-87)
```

or

```
Violet wins (87-89)
```

(that is, the winning color, followed by the scores for orange and violet, in that order), or

```
Tie game (87-87)
```

Use exactly these formats.

In response to the '`b`' command, print out the current board in the following format:

```
===
  OO------------
  -OO-----------
  --O-----------
  ---O----------
  ---OO---------
  --OO----------
  --------------
  --------------
  --------------
  --------------
  -------V------
  ------VV------
  -----VV-------
  VVVVV---------
===
```

Use *exactly* this format (complete with the '`===`' at the beginning and end and the two spaces at the start of each of the other lines). Don't add trailing spaces.

Constraints on output.

1. Except in response to the '`b`' command, your program must not output lines containing '`===`'.
2. Not counting moves from an AI, no output from your program (that is, no substring of the output of your program) may match a valid move, *except* that your program may echo a move that was just read in as input.

4 Running the Program

To execute the program, you will use the command

`java duo.Main` *Name1 Name2 [Seed [Initial-Move-File]]*

where '`[...]`' indicates optional arguments. Here, *Name1* and *Name2* are the (distinct) names of the orange and violet players, respectively. A name that begins with the '`@`' character indicates an AI player. If provided, *Seed* is a long constant that you should use to seed any random number generator you use. The idea is that if one or both players is an AI, then the same seed will reproduce the same game (a rather useful property for debugging purposes). If provided, the *Initial-Move-File* is the name of a file containing a sequence of move commands, one per line in the format given in §2. No other commands should appear in this file. The moves supplied in this file are made first to set up an initial position, before the players start playing.

5 Your Task

The staff directory will contain skeleton files for this project in `proj2`.

Please read *General Guidelines for Programming Projects.*

Be sure to include tests of your program (that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them. To help with testing and debugging, we will provide our own version of the program, so that you can test your program against ours (we'll be on the lookout for illegal moves). More details will follow.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes an illegal move, tell him and just give him another chance. We don't care about the message format you use to do this.

We will expect your finished programs to be workmanlike, and of course, will enforce the mechanical style standards. Make sure all methods are adequately commented—meaning that after reading the name, parameters, and comment on a method, you don't need to look at the code to figure out what a call will do. Don't leave debugging print statements lying around. In fact, don't use them; learn to use the debugger (either `gjdb` or that of `Eclipse` or your favorite Java-system vendor). Comments in your appropriate modules should describe how your AI chooses its move.

Your AI should be good enough to win more often than not against random play. In particular, it must be able to find a forced win when within a few moves of the end of the game.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that

```
gmake check
```

runs your tests.

6 Advice

At first glance, it might seem that with all the options available, the program would have to be a mass of **if** statements covering all possible cases. But with proper use of abstraction, this does not have to be. If you need random numbers, take a look at `java.util.Random` and Chapter 11 of *Data Structures (Into Java)*.

I suggest working first on the class `Board`, which is supposed to represent the game board. Next, implement the human player. Then you can tackle writing a machine player. Start with something really simple (perhaps choosing a legal move at random) and introduce strategy only when you get everything working properly.

Use `svn` regularly. It provides backup. It's the safest way to transport things from a copy of your project on a local computer to (`svn commit`) and from (`svn update`) the instructional machines. Frequent commits will limit the amount of work you have to do when a file gets messed up. Use `svn status` to check for files that should be added and to make sure all necessary commits have been done.

Use of `gmake style` not only encourages proper formatting, but also shows you missing documentation or left-over internal comments that might indicate work that still needs to be done, and can find certain error-prone constructs or bad practices.