CS 351: Design of Large Programs
Project 3: Scrabble

Brooke Chenoweth

Fall 2024

1 Game Description

Scrabble is a word game in which two to four players score points by placing tiles bearing a single letter onto a board divided into a 15 by 15 grid of squares. The tiles must form words that, in crossword fashion, read left to right in rows or downward in columns, and be included in a standard dictionary.[1]

In this assignment, you will be implementing a two-player version of the game, with the human user playing against a computer opponent.The dictionary will consist of a file with list of acceptable words. Your program should be able to handle any dictionary file I throw at it.[2]

1.1 Game Rules

Adapted/stolen from `https://www.hasbro.com/common/instruct/Scrabble_%282003%29.pdf`

1.1.1 Set Up

There are 100 tiles a standard Scrabble game. They are placed in bag or scrambled face down on the table to serve as a pool of tiles.

Initially, each player draws seven tiles and places them on their tray. The player with a tile earliest in the alphabet goes first. (If it makes it easier, you may have the human player always go first in your game, just make sure you document this choice.)

1.1.2 Game Play

1. The first player combines two or more of his or her letters to form a word and places it on the board to read either across or down with one letter on the center square. Diagonal words are not allowed.

[1] Thank you, Wikipedia! `https://en.wikipedia.org/wiki/Scrabble`

[2] The standard Scrabble dictionary is very large, so you may want to swap in a smaller dictionary file when testing.

2. Complete your turn by counting and announcing your score for that turn. Then draw as many new letters as you played; always keep seven letters on your rack, as long as there are enough tiles left in the bag.

3. Play passes to the left. The second player, and then each in turn, adds one or more letters to those already played to form new words. All letters played on a turn must be placed in one row across or down the board, to form at least one complete word. If, at the same time, they touch others letters in adjacent rows, those must also form complete words, crossword fashion, with all such letters. The player gets full credit for all words formed or modified on his or her turn.

4. New words may be formed by:

   - Adding one or more letters to a word or letters already on the board.
   - Placing a word at right angles to a word already on the board. The new word must use one of the letters already on the board or must add a letter to it.
   - Placing a complete word parallel to a word already played so that adjacent letters also form complete words.

5. No tile may be shifted or replaced after it has been played and scored.

6. Blanks: The two blank tiles may be used as any letters. When playing a blank, you must state which letter it represents. It remains that letter for the rest of the game.

7. You may use a turn to exchange all, some, or none of the letters. To do this, place your discarded letter(s) facedown. Draw the same number of letters from the pool, then mix your discarded letter(s) into the pool. This ends your turn.

8. Any play may be challenged before the next player starts a turn. If the play challenged is unacceptable, the challenged player takes back his or her tiles and loses that turn. If the play challenged is acceptable, the challenger loses his or her next turn. Consult the dictionary for challenges only. All words made in one play are challenged simultaneously. If any word is unacceptable, then the entire play is unacceptable. Only one turn is lost on any challenge.

9. The game ends when all letters have been drawn and one player uses his or her last letter; or when all possible plays have been made.

I don't expect that challenges as described in item 8 will be relevant to the computer game. The computer player will only play valid words and it is reasonable to allow the human player to just keep trying until they manage to make a legal play.

Any modifications/simplifications you make to the rules (not allowing the player to pass, for example) must be documented in the your readme file. Bear in mind that some game changes may prevent you from getting full credit (for example, you are expected to properly handle blank tiles).[3]

[3]Ask me if you are uncertain if a particular rules modification would be acceptable.

1.1.3 Scoring

1. The score value of each letter is indicated by a number at the bottom of the tile. The score value of a blank is zero.

2. The score for each turn is the sum of the letter values in each word(s) formed or modified on that turn, plus the additional points obtained from placing letters on Premium Squares.

3. Premium Letter Squares: A light blue square doubles the score of a letter placed on it; a dark blue square triples the letter score.

4. Premium Word Squares: The score for an entire word is doubled when one of its letters is placed on a pink square: it is tripled when one of its letters is placed on a red square.

   Include premiums for double or triple letter values, if any, before doubling or tripling the word score. If a word is formed that covers two premium word squares, the score is doubled and then re-doubled (4 times the letter count), or tripled and then re-tripled (9 times the letter count). NOTE: the center square is a pink square, which doubles the score for the first word.

5. Letter and word premiums count only on the turn in which they are played. On later turns, letters already played on premium squares count at face value.[4]

6. When a blank tile is played on a pink or red square, the value of the word is doubled or tripled, even though the blank itself has no score value.

7. When two or more words are formed in the same play, each is scored. The common letter is counted (with full premium value, if any) for each word.

8. BINGO! If you play seven tiles on a turn, it's a Bingo. You score a premium of 50 points after totaling your score for the turn.

9. Unplayed Letters: When the game ends, each player's score is reduced by the sum of his or her unplayed letters. In addition, if a player has used all of his or her letters, the sum of the other players' unplayed letters is added to that player's score.

10. The player with the highest final score wins the game. In case of a tie, the player with the highest score before adding or deducting unplayed letters wins.

2 Assignment

2.1 File Formats

All your programs will need to be able to read in a dictionary file of words. Your scoring program and solver programs will also need to be able to read test cases of using of board configurations (and trays, for the solver). I will also provide you with files of tile frequency

[4]The square is already covered with a tile at that point, so it is as if the multiplier isn't even there.

and point values, but since we will assume standard scrabble tiles for all tests, you may choose to just hard code those values in your program rather than initializing the tiles from a configuration file.[5]

2.1.1 Dictionary File

A dictionary file will be a simple text file of words, with one word per line.

- Some large word lists can be found at `https://www.wordgamedictionary.com/dictionary/`[6]

  We will test your program with some version of the TWL06 and/or SOWPODS list. These lists have many, many words,[7]

- Do not make assumptions about the order, uniqueness, or capitalization of the words. Most word lists are given in alphabetical order with no duplicates, but it would be fair game for us to test your progam with something that isn't.

2.1.2 Board Configuration

The board configuration will be a text file representing the game board and tiles on it.

- The first line contains a single integer value, representing the numbers of rows and columns in the board. (We are assuming all game boards are square.)
- The following lines each represent a row of the board, represented as a sequence of squares separated by spaces.
- Each square will consist of two characters:
  - If the square is unoccupied, the two characters will represent the word and letter multipliers (if any) or just a period if there is no multiplier (that is, the multiplier is one). So, `3.` would represent a triple word square, `.2` would represent a double letter square, and `..` would be a plain non-premium square.
  - If the square has a tile on it, the two characters will be a blank and the letter of the tile. A lowercase letter represents an ordinary tile and an uppercase letter represents a blank tile being used as that letter.

2.1.3 Tray Configuration

The tray configuration in a testing file will consist of a single line containing the letters of the tiles on the tray. Lowercase letters will be used for regular tiles and blanks will be represented by an asterisk.

[5] Keep your design and implementation flexible in case things change!

[6] If you download them, make sure you edit off the header rows at the top so your file just contains the words.

[7] TWL06 has more than 170,000 and SOWPODS is about half again as large.

2.1.4 Tile Frequency File

I have provided you with a tile frequency file as a text file with one line per letter containing the letter (asterisk for blank), point value, and number of tiles for that letter, separated by blanks. You may choose to use this file or not. We will be assuming standard Scrabble tile distributions in all our tests.[8]

2.2 Computer Player

The computer player should always play the highest scoring move possible on its turn. If more than one move is tied for highest scoring, pick one however you like (randomly, first/last found, some sort of game strategy, etc.)

2.2.1 Time Requirements

Consider the algorithms and data structures you use for the computer player carefully. The computer player should not take more than 2 seconds to find a solution on a standard scrabble board with a tray containing 2 blanks and 5 letters when run on one of the CS lab machines and using the SOWPODS word list.[9]

2.3 Required Programs

You will create three separate programs for this project, using your common objects. One program is a command line puzzle solver that can validate and score potential plays, one is a command line program that we can use to test your computer player, and the final program is, of course, a full Scrabble game.

You may develop these programs in whichever order you choose, but I will point out that your computer player cannot find the best scoring play if it cannot correctly compute the score.

2.3.1 Validating and Scoring Program

Both the computer player and human vs computer game need to be able to verify that playing a given set of tiles at a particular set of positions is legal and, if so, how many points is that worth. Errors in your legal move and scoring logic can break other game functionality, so I want you to submit a program that will let us test that part in isolation from the rest of the game.

1. Name this jar file `scorechecker.jar` to simplify automated testing.

2. Your program will take a command line argument for the name of the dictionary file to load.

   The same word list will be used for all tests in a given run of the program.

[8] Keep your design and implementation flexible in case things change!

[9] This is subject to vary a bit for particularly perverse boards, but in general the solution should be found fairly quickly.

So, if your jar file was named "scorechecker.jar" and you were going to use the "sowpods.txt" word list, you could run the program with the command:

```
java -jar scorechecker.jar sowpods.txt
```

3. Your program will two board configurations from standard input in the format described in section 2.1.2 above. Presumably the second board will be the same as the first after some tiles have been added.

```
7
3. .. .. 2. .. .. 3.
.. .3 .. .. .. .3 ..
.. .. .2 .. .2 .. ..
2. .. ..  c  a  t 2.
.. .. .2 .. .2 .. ..
.. .3 .. .. .. .3 ..
3. .. .. 2. .. .. 3.
7
3. .. .. 2. .. ..  d
.. .3 .. .. .. .3  o
.. .. .2 .. .2 ..  g
2. .. ..  c  a  t  s
.. .. .2 .. .2 .. ..
.. .3 .. .. .. .3 ..
3. .. .. 2. .. .. 3.
```

4. Your program will determine if the two boards are compatible, and if so, find the tiles played, determine if the play is legal, and the score for the play.

```
original board:
3. .. .. 2. .. .. 3.
.. .3 .. .. .. .3 ..
.. .. .2 .. .2 .. ..
2. .. ..  c  a  t 2.
.. .. .2 .. .2 .. ..
.. .3 .. .. .. .3 ..
3. .. .. 2. .. .. 3.
result board:
3. .. .. 2. .. ..  d
.. .3 .. .. .. .3  o
.. .. .2 .. .2 ..  g
2. .. ..  c  a  t  s
.. .. .2 .. .2 .. ..
.. .3 .. .. .. .3 ..
```

```
3. .. .. 2. .. .. 3.
play is d at (0, 6), o at (1, 6), g at (2, 6), s at (3, 6)
play is legal
score is 48
```

5. The program will continue to read pairs of board configurations until the end of the input. *You will not receive full credit if you only solve one test case!*

6. I will provide you with samples of input and output for the scoring program. Your output must match mine *exactly* to simplify grading.[10]

   Bear in mind that we will not just be testing your program with the sample input/output files, but also using some unknown test cases, so make sure your program can handle that. If you make your own test cases, share them with the class in the discussion forum. Challenge each other's implementations!

2.3.2 Word Solver

We need to be able to test your computer players in a systematic way, so you will provide a console based program that can read in game configurations and find the highest scoring move.

1. Name this jar file `solver.jar` to simplify automated testing.

2. Your program will take a command line argument for the name of the dictionary file to load.

   The same word list will be used for all tests in a given run of the program.

   So, if your jar file was named "solver.jar" and you were going to use the "sowpods.txt" word list, you could run the program with the command:

   ```
   java -jar solver.jar sowpods.txt
   ```

3. Your program will read a game configuration from standard input in the format described in section 2.1.2 above.

4. Your program will read a line of characters from standard input representing the computer player's tray as described in section 2.1.3 above.

5. Example board and tray input:

   ```
   7
   3. .. .. 2. .. .. 3.
   .. .3 .. .. .. .3 ..
   .. ..  a  d .2 .. ..
   ```

[10] This includes whitespace, capitalization, and punctuation!

```
2. ..  u  h .. .. 2.
.. ..  l  o .2 .. ..
..  m  a  t .. .3 ..
 r  e  S  i  d .. 3.
toloeri
```

6. Your program will determine the highest scoring legal move possible on the given board using the letters on the tray. You will output the word played, the score for the play, and the resulting board.

   Your specific output should print the original board configuration and tray, then the word and its score, and finally the resulting solution board so we can see where you'll play. Print a blank line between solutions to make the output a bit more legible.

   Example solution output:

```
Input Board:
3. .. .. 2. .. .. 3.
.. .3 .. .. .. .3 ..
.. ..  a  d .2 .. ..
2. ..  u  h .. .. 2.
.. ..  l  o .2 .. ..
..  m  a  t .. .3 ..
 r  e  S  i  d .. 3.
Tray: toloeri
Solution troolie has 67 points
Solution Board:
3. .. .. 2. ..  t 3.
.. .3 .. .. ..  r ..
.. ..  a  d .2  o ..
2. ..  u  h ..  o 2.
.. ..  l  o .2  l ..
..  m  a  t ..  i ..
 r  e  S  i  d  e 3.
```

7. The program will continue to read board and tray configurations and output solutions until the end of the input. *You will not receive full credit if you only solve one test case!*

8. I will provide you with samples of input and output for the solver program. Your output must match mine *exactly* to simplify grading.[11]

   Bear in mind that we will not just be testing your program with the sample input/output files, but also using some unknown test cases, so make sure your program can handle that. If you make your own test cases, share them with the class in the discussion forum. Challenge each other's implementations!

[11] This includes whitespace, capitalization, and punctuation!

2.3.3 Scrabble Game

Make a two player game of Scrabble, human against computer. Use the standard Scrabble board, tiles, and dictionary.

I expect your program to have a GUI implemented with JavaFX, but the exact user interface is up to you. Make sure you document how to use the program in your readme file.