Hoggle Assignment[1]

The game of Hoggle (Hexagonal Boggle)

This game is inspired by the commercial game *Boggle*. It is played on a grid with 19 spaces, into which you "shake" and randomly distribute 19 dice. These 6-sided dice have letters rather than numbers on the faces, and the sides are hexagons, so they create a hexagonal grid on which you form words[2].

In the "classic" physical version of Boggle, all players start together and write down all the words they can find by tracing a path through adjoining letters. When time is called, duplicates are removed from the players' lists and the players receive points for their remaining words - that is, those words each person **uniquely** found - based on the word lengths.

Your assignment is to write a program that plays a graphical rendition of this game, adapted to allow a human and the computer to play against each other. In most cases, the computer completely dominates the human, but it can be fun to play anyway.

The main focus of this assignment is designing and implementing the recursive algorithms required to find and verify words that appear on the Hoggle board. Because the learning goal is for you to concentrate on the strategy section, the starter code includes a `graphics` module which is responsible for the graphical display. There is still a lot of work left for you, though – definitely do not put it off until the night before (or even, say, 3 days before)!

How's this going to work?

First, you will read the dice in from a file, simulate "shaking" the dice up and laying them out on the board graphically.

The human player gets to go first (and thus has **some** chance of beating the computer). The player proceeds to enter words, one at a time. If a word meets the requirements (see "The Human Player's Turn" below), then the letters forming the word are highlighted graphically, the word is added to the player's word list, and the player gains points based on the word's length.

The player indicates that they are done entering words by hitting enter without typing any letters.

At this point, the computer takes over. The computer player searches through the board looking for words that the player didn't find, and awards itself points for finding all those words.

[1] Adapted from SIGCSE Nifty Assignments 2002, with partial credit to Julie Zelenski and Todd Feldman.

[2] If you are an experienced tabletop gamer, you are probably saying: "Wait, there's no such thing as a 6-sided die with hexagon-shaped sides!" You are right. Try not to let this bother you!

The Dice

The letters in Hoggle are not simply chosen at random. Instead, the letters on the faces of the dice are arranged in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. To recreate this, you have access to a text file, `dice.txt`, that contains descriptions of 19 dice similar to those used in Boggle. Each die description is a single line of 6 capital letters.

During initialization, you need to read the dice file and store it into a suitable data structure for subsequent use. At the beginning of each game, the "shaking" of the dice has two different random aspects to consider. First, the cubes themselves need to be shuffled so that the same die is not always in the same cell of the board. Second, a random side from each die needs to be chosen to be the face-up letter.

Once your initialization is complete, you are ready to implement two types of recursive search; one for the user and one for the computer. There are two distinct types of recursion happening here: For the user, you search for a specific word and stop as soon as you find it, while for the computer, you are searching for all words.

Important: While you may be tempted to try and integrate the two so they work as a single type of recursion, this is a very bad idea. Your program will get very messy trying to integrate the two, as they are not algorithms that can be unified well.

The Human Player's Turn

After the board is displayed, the player gets a chance to enter any word found on the board. Your program must read in a list of words until the user signals the end of the list by typing a blank line. As the user enters each word, your program must check that the word meets all of these conditions **IN ORDER**:

- It is at least 3 letters long
- It is not already on the player's word list
- It can be formed from the dice on the board. To be "formable", it must be composed of adjoining letters on the board AND not use any board position more than once.
- It is a real word (i.e. is in the lexicon)

If any of these conditions fail, the program tells the user about it and does not give any score for the word. (**Make sure you check them IN ORDER and report about the first condition a word fails.**) If the word does satisfy all the conditions, it is added to the user's word list and their score is updated. In addition, the dice making up the word highlighted in order from the beginning of the word to the end.

A word's point value is determined by its length: 1 point for the word itself and 1 additional point for every letter over three. (Note that the provided functions already take care of most of the displaying and scoring of words.)

The Computer's Turn

On the computer's turn, the task is to find all of the words that the human player missed by recursively searching the board for words beginning at each square on the board. In this phase, the same conditions apply as on the user's turn, plus the additional restriction that **the computer is not allowed to count any of the words already found by the player**.

The `hgraphics` module

The functions from the `hgraphics.h` interface are used to manage the appearance of the game. Read the `hgraphics.h` file to learn how to use these functions.

Solution strategies

In a project of this complexity, it is important that you get an early start and work consistently toward your goal. To be sure that you are making progress, it also helps to divide up the work into manageable pieces, each of which has identifiable milestones. Here's a suggested plan of attack that breaks the problem doing into the following five phases:

Task 1 - Dice reading, board drawing, dice shaking
Design your data structure for the dice and board. You should not use any global variables in this program. Read the dice file and store the dice. Create your shuffling routine. Use the `hgraphics` routines to draw the starting board.

Task 2 - User's turn (except for finding words on the board)
Write the loop that allows the user to enter words. Reject words that have already been entered or that don't meet the minimum word length or that aren't in the lexicon. Do not assume there is any upper limit on the number of words that may be found by the user. Put the `hgraphics` functions to work for you adding words to the graphical display and keeping score. At this point, the words the user enters may or may not be possible to form on the board - that's the next task.

Task 3 - Find a given word on the board
Now, apply your recursion talents to check if the user's words can actually be formed from the board. emember that a valid word must obey the adjoinment and non-duplication rules. This is a form of **recursive backtracking**: at each step of attempting to build the user's word, there are a number of choices for the "next dice" to use, and your search can run into the equivalent of a dead end, requiring backtracking.

Task 4 - Find all words on the board (computer's turn)
Now it's time to implement the computer player. With the power of recursion and an efficient, large lexicon, your computer player will usually achieve an epic victory by traversing the board and finding **every** word the user missed. This is an exhaustive search: you will completely explore all positions on the board, hunting for possible words. It is ALSO a form of **recursive backtracking**. This phase is where the most difficult applications of recursion come into play. It is easy to get lost in the recursive decomposition and you should think carefully about how to proceed.

Task 5 - Add polish and repetition. Once one whole game is working, make sure your code plays repeated games correctly. (Is it resetting everything it should when each game begins?) Fix any little aethetic details and deal with weird user input.

Extension possibilities

The Hoggle project has multiple opportunities for extensions. Successful extensions will be awarded **some** extra credit, but **ONLY** if they are extensions to a program that is already working. (Be 100% sure you have **tested** your existing game **completely** before starting on any extension!) The amount of extra credit granted is based on my judgement of the difficulty and completeness of your extension(s).

- Make **Q** useful! As in commercial Boggle, make it so that anytime Q shows up in a space, it looks like and operates as "QU".
- Improve the program's graphics. (One possibility: when highlighting, draw arrows or more interesting animation to make clear how a given word's letters are connected.)
- Let the user play a word by clicking on letters on the board.
- Add networked play. (I have no nicely-packaged multiplayer library to offer you, so you would need to do a lot of your own implementation of networking code, using something like the ENet library, but this would be a good project for someone who wants to learn something a little "extra", and isn't as hard as it might sound.)
- If you have an idea/suggestion of your own, let me know!