
CS 2110 Summer 2011: Assignment 1 — Hangman

Due July 5 at 5pm

This assignment is to be done by students individually; future assignments will be done in groups of two. You are expected to respect the code of academic integrity, as described in the course syllabus.

1 Implement the Game of Hangman¹

It's hard to write computer programs to play games. When we as humans sit down to play a game, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, on the other hand, blindly follow a preset algorithm that (hopefully) causes it to act somewhat intelligently. Though computers have bested their human masters in some games, most notably checkers, chess, and recently jeopardy, the programs that do so often draw on hundreds of years of human game experience or crawl the Web to find answers, and use extraordinarily complex algorithms and optimizations to out calculate their opponents.

While there are many viable strategies for building competitive computer game players, there is one approach that has been fairly neglected in modern research – cheating. Why spend all the effort trying to teach a computer the nuances of strategy when you can simply write a program to play dirty and win handily all the time? In this assignment, you will build a mischievous program that bends the rules of Hangman to trounce its human opponent time and time again. In doing so, you'll cement your skills with abstract data types and iterators, and will hone your general programming savvy. Plus, you'll end up with a piece of software which will be highly entertaining. At least, from your perspective.

In case you aren't familiar with the game Hangman, the rules are as follows:

1. One player chooses a secret word, then writes out a number of dashes equal to the word length.
2. The other player begins guessing letters. Whenever she guesses a letter contained in the hidden word, the first player reveals each instance of that letter in the word. Otherwise, the guess is wrong.

¹This assignment was created by Keith Schwarz and adapted with permission.

3. The game ends either when all the letters in the word have been revealed or when the guesser has run out of guesses.

Fundamental to the game is the fact the first player accurately represents the word she has chosen. That way, when the other players guess letters, she can reveal whether that letter is in the word. But what happens if the player doesn't do this? This gives the player who chooses the hidden word an enormous advantage. For example, suppose that you're the player trying to guess the word, and at some point you end up revealing letters until you arrive at this point with only one guess remaining:

D O - B L E

There are only two words in the English language that match this pattern: "doable" and "double". If the player who chose the hidden word is playing fairly, then you have a fifty-fifty chance of winning this game if you guess 'A' or 'U' as the missing letter. However, if your opponent is cheating and hasn't actually committed to either word, then there is no possible way you can win this game. No matter what letter you guess, your opponent can claim that she had picked the other word, and you will lose the game. That is, if you guess that the word is "doable", she can pretend that she committed to "double" the whole time, and vice-versa.

Let's illustrate this technique with an example. Suppose that you are playing Hangman and it's your turn to choose a word, which we'll assume is of length four. Rather than committing to a secret word, you instead compile a list of every four-letter word in the English language. For simplicity, let's assume that English only has a few four-letter words, all of which are reprinted here:

ALLY BETA COOL DEAL ELSE FLEW GOOD HOPE IBEX

Now, suppose that your opponent guesses the letter 'E'. You now need to tell your opponent which letters in the word you've "picked" are E's. Of course, you haven't picked a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with E's highlighted in each word:

ALLY **BETA** COOL **DEAL** **ELSE** **FLEW** GOOD HOPE**E** IBEX**E**

If you'll notice, every word in your word list falls into one of five "word families":

- - - - - , which contains the word ALLY, COOL, and GOOD.
- - E - - , containing BETA and DEAL.
- - - E - , containing FLEW and IBEX.

- E - - E, containing ELSE.
- - - - E, containing HOPE.

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal any one of the above five families. There are many ways to pick which family to reveal – perhaps you want to steer your opponent toward a smaller family with more obscure words, or toward a larger family in the hopes of keeping your options open. In this assignment, in the interests of simplicity, we’ll adopt the latter approach and always choose the largest of the remaining word families. In this case, it means that you should pick the family - - - - . This reduces your word list down to ALLY COOL GOOD and since you didn’t reveal any letters, you would tell your opponent that his guess was wrong. Let’s see a few more examples of this strategy. Given this three - word word list, if your opponent guesses the letter O, then you would break your word list down into two families:

- - O O - , containing COOL and GOOD.
- - - - - , containing ALLY.

The first of these families is larger than the second, and so you choose it, revealing two O’s in the word and reducing your list down to

COOL GOOD

But what happens if your opponent guesses a letter that doesn’t appear anywhere in your word list? For example, what happens if your opponent now guesses ‘T’? This isn’t a problem. If you try splitting these words apart into word families, you’ll find that there’s only one family: the family - - - - in which T appears nowhere and which contains both COOL and GOOD. Since there is only one word family here, it’s trivially the largest family, and by picking it you’d maintain the word list you already had. There are two possible outcomes of this game. First, your opponent might be smart enough to pare the word list down to one word and then guess what that word is. In this case, you should congratulate him – that’s an impressive feat considering the scheming you were up to! Second, and by far the most common case, your opponent will be completely stumped and will run out of guesses. When this happens, you can pick any word you’d like from your list and say it’s the word that you had chosen all along. The beauty of this setup is that your opponent will have no way of knowing that you were dodging guesses the whole time – it looks like you simply picked an unusual word and stuck with it the whole way.

2 The Assignment

Your assignment is to write a computer program which plays a game of Hangman using this “Adversarial Hangman” algorithm. You will implement four Java classes, de-

scribed below, and answer a few questions (Section 2.2).

Your program should do the following:

1. Read the file `dictionary.txt`, which contains the full contents of the Official Scrabble Player's Dictionary, Second Edition. This word list has over 120,000 words, which should be more than enough for our purposes.
2. Prompt the user for a word length, reprompting as necessary until she enters a number such that there's at least one word that's exactly that long. That is, if the user wants to play with words of length - 42 or 137, since no English words are that long, you should reprompt her. Also if the user fails to provide an integer you should reprompt her.
3. Play a game of Hangman using the Adversarial Hangman algorithm, as described below:
 - (a) Construct a list of all words in the English language whose length matches the input length.
 - (b) A user starts off with 5 guesses.
 - (c) Print out how many guesses the user has remaining, along with any letters the player has guessed and the current blanked-out version of the word.
 - (d) Prompt the user for a single letter guess, reprompting until the user enters a letter that she hasn't guessed yet. Make sure that the input is exactly one character long and that it's a letter of the alphabet. If not, reprompt her.
 - (e) Partition the words in the dictionary into groups by word family.
 - (f) Find the most common "word family" in the remaining words, remove all words from the word list that aren't in that family, and report the position of the letters (if any) to the user. If the word family doesn't contain any copies of the letter, subtract a remaining guess from the user.
 - (g) If the player has run out of guesses, pick a word from the word list and display it as the word that the computer initially "chose".
 - (h) If the player correctly guesses the word, congratulate her.

To guide the organization of your program, we provide you with three interfaces: `WordGuesser`, `WordChooser`, `Hangman`.

- Interface `WordGuesser` deals with user input.
 - Your task is to write a class `HumanGuesser` that implements the `WordGuesser` interface by requesting and parsing the human input through the command line.
- Interface `WordChooser` deals with the gaming strategy.

- Your task is to write a class `AdversarialWordChooser` that implements the `WordChooser` interface by following the adversarial strategy of not committing to a word.
- In addition, we ask you to write a class `RandomWordChooser` that also implements the `WordChooser` interface but follows the basic strategy of choosing a word uniformly at random at the beginning of the game.
- Interface `Hangman` lets the `WordGuesser` and the `WordChooser` play against each other. It keeps track of the game state and determines the winner.
 - Your task is to write a class `HangmanGame` that implements the `Hangman` interface.

2.1 Tips on storing word families

It's up to you to think about how you want to partition words into word families. Think about what data structures would be best for tracking word families and the master word list. Would an associative array work? How about a stack or queue? Thinking through the design before you start coding will save you a lot of time and headache.

Since you're building this project from scratch, you'll need to do a bit of planning to figure out what the best data structures are for the program. There is no "right way" to go about writing this program, but some design decisions are much better than others (e.g. you can store your a single word family in a stack, but this is probably not the best option; similarly you can store a set of word families in an array of some kind, but again this is probably not the best option). Here are some general tips and tricks that might be useful:

1. Letter position matters just as much as letter frequency. When computing word families, it's not enough to count the number of times a particular letter appears in a word; you also have to consider their positions. For example, `BEER` and `HERE` are in two different families even though they both have two `E`'s in them. Consequently, representing word families as numbers representing the frequency of the letter in the word will get you into trouble.
2. Watch out for gaps in the dictionary. When the user specifies a word length, you will need to check that there are indeed words of that length in the dictionary. You might initially assume that if the requested word length is less than the length of the longest word in the dictionary, there must be some word of that length. Unfortunately, the dictionary contains a few "gaps". The longest word in the dictionary has length 29, but there are no words of length 27 or 26. Be sure to take this into account when checking if a word length is valid.
3. Don't explicitly enumerate word families. If you are working with a word of length n , then there are 2^n possible word families for each letter. However, most

of these families don't actually appear in the English language. For example, no English words contain three consecutive U's, and no word matches the pattern E - EE - EE - - E. Rather than explicitly generating every word family whenever the user enters a guess, see if you can generate word families only for words that actually appear in the word list.

2.2 Questions

Question 1. Data structures

Which data structures do you use to store and update the word families? What methods on those data structures do you use?

Question 2. Strategy

Claim. If the goal of the adversarial hangman is that the human loses, it is not always optimal to pick the largest word family when a human guesses a new letter.

Prove this claim by coming up with an example in which it would have been better to select a smaller word family. Here are some guidelines: Assume that the human has 2 guesses. Come up with a set of words (not necessarily found in any dictionary). Determine the first guess of the human that splits the set of words into two word families A, B . One of them is larger than the other, say $|A| > |B|$. Show that if A was picked then the human wins (i.e., come up with a sequence of guesses for the human that will lead to a final word family of size 1 — think about the circumstances under which a human left with a single guess can ask for multiple letters). Show that if B was picked then the human loses (this can be done by showing that no matter which letter the human guesses next there are at least two word families remaining).

2.3 Grading Guidelines

The following grading guidelines that will be used to assess this assignment:

1. **Adhere to the interfaces.** This is a pre-requisite for us grading your assignment. No exceptions.
2. **Follow good programming practices.** Follow the style guidelines, document your code, and write a user friendly game. Style guidelines can be found on the course website, under Assignments.
3. **Answer questions.** Be sure to answer the two questions (Section 2.2). Conciseness and clarity are key to a good answer.

2.4 Submission

Submissions should be handed in via CMS <http://cms.csuglab.cornell.edu>. You should submit a jar file called `assignment1.jar` containing:

- `HumanGuesser.java`
- `RandomWordChooser.java`
- `AdversarialWordChooser.java`
- `HangmanGame.java`
- “answers.txt” containing the answers of 2.2. (You can create text files in Eclipse.)
- (Optional) `MyWordChooser.java` — described in the extension below.

In Eclipse, you can create the jar file as follows.

- Go to File → Export, select JAR file.
- Select only the `cs2110.netid.assignment1` package.
- Check the box “Export Java source files and resources.”
- Name the JAR file `assignment1.jar`.
- Click finish.

3 Extension (Optional)

Completing the following extension will earn you karma points, as explained in the syllabus.

The algorithm outlined in this handout is by no means optimal, and there are several cases in which it will make bad decisions. For example, suppose that the human has exactly one guess remaining and that computer has the following word list:

DEAL TEAR MONK

If the human guesses the letter ‘E’ here, the computer will notice that the word family `- E - -` has two elements and the word family `- - - -` has just one. Consequently, it will pick the family containing `DEAL` and `TEAR`, revealing an E and giving the human another chance to guess. However, since the human has only one guess left, a much better decision would be to pick the family `- - - -` containing `MONK`, causing the human to lose the game. There are several other places in which the algorithm does not function ideally. For example, suppose that after the player guesses a letter, you find that there are two word families, the family `- - E -` containing 10,000 words

and the family - - - - containing 9,000 words. Which family should the computer pick? If the computer picks the first family, it will end up with more words, but because it revealed a letter the user will have more chances to guess the words that are left. On the other hand, if the computer picks the family - - - - , the computer will have fewer words left but the human will have fewer guesses as well. More generally, picking the largest word family is not necessarily the best way to cause the human to lose. Often, picking a smaller family will be better. After you implement this assignment, take some time to think over possible improvements to the algorithm. You might weight the word families using some metric other than size. You might consider having the computer “look ahead” a step or two by considering what actions it might take in the future.

4 Setup

1. Download and set up Java and Eclipse. Instructions are available on the course website (under Assignments).
2. Create a new Eclipse project using the Java Project Wizard (File → New → Java Project). Name it CS2110.
3. By default, Eclipse compiles automatically and only before it is run. You should reconfigure it so that you can compile more frequently (to help you catch syntax errors). Under the Project menu, uncheck Build Automatically. To compile, Project → Build All.
4. Download the source jar file (called `assignment1.jar`) from CMS, <http://cms.csuglab.cornell.edu>.
5. After downloading the jar, import it into your Eclipse project. Go to File → Import, choose Archive File, use the Browse... button to find the jar you just downloaded and under ‘Into Folder:’ use the Browse... button to select `CS2110/src`.
6. You should have 3 new files in a new `cs2110.assignment1` package. Do not modify these files.
7. Now, create your own package (File → New → Package). You should name it `cs2110.netid.assignment1`, where *netid* is **your** NetID. This is where your original source files will reside. We will follow this naming convention for later assignments.
8. To get started, first create the `HumanGuesser` class. To create the class, go File → New → Class; be sure that you create this class **inside** the package `cs2110.netid.assignment1`.
9. The `HumanGuesser` class is supposed to implement the `WordGuesser` interface, so modify the class definition accordingly (add the phrase `implements WordGuesser`

after `public class HumanGuesser` and before the `{`).

10. Eclipse will not recognize the `WordGuesser` interface and squiggly red lines may appear. You must add an import statement. Between the package name and the class definition, add the statement `import cs2110.assignment1.WordGuesser;` Or, you can simply hover your mouse over the phrase `WordGuesser` and Eclipse will prompt you to import it.
11. The `HumanGuesser` class now claims to implement the `WordGuesser` interface, but lacks definitions for any of the interface's methods. Your task is to write implementations of these methods (and possibly other methods, like constructors and private helper methods).
12. Repeat a similar process for the remaining classes.

5 Useful Resources

To review basic Java file I/O commands take a look at the textbook Appendix C.

Java data structures that might become handy:

- **Interface Set** <http://download.oracle.com/javase/1.4.2/docs/api/java/util/Set.html>
 - **Class HashSet** <http://download.oracle.com/javase/1.4.2/docs/api/java/util/HashSet.html>
- **Interface Map** <http://download.oracle.com/javase/1.4.2/docs/api/java/util/Map.html>
 - **Class HashMap** <http://download.oracle.com/javase/1.4.2/docs/api/java/util/HashMap.html>
- **Interface List** <http://download.oracle.com/javase/1.4.2/docs/api/java/util/List.html>
 - **Class LinkedList** <http://download.oracle.com/javase/1.4.2/docs/api/java/util/LinkedList.html>

For the `RandomWordChooser` you might find the `Random` class useful <http://download.oracle.com/javase/1.4.2/docs/api/java/util/Random.html>.

6 Need help?

Follow and participate in the discussions <http://groups.google.com/group/cs2110-2011/>
Come to our office hours: time slots are listed on the course website and the instructor is also available by appointment. lose