Brandon Kirklen
Collier Robinson
Chris Peterson
2015-05-04
CPE 102-22

CPE 102 Project 1 Algorithm Explanation

1. Functionality
    a. No multiple wrong guesses
    b. Show correct answer upon loss
    c. Ignore case
    d. Hints
    e. Menu
        i. Choose Difficulty
    f. Difficulty
        i. Select from sorted list
        ii. Link a difficulty score with each word
            1. Greedy Algorithm which perfectly optimizes the number of guesses needed to solve the word
    g. GUI
        i. Add a background and title
        ii. Other changes detailed in art
2. Art
    a. Background
    b. Clean up the existing drawing
        i. Change the stand
        ii. Enhance smoothness of lines
        iii. Made lines thicker and more rounded
    c. Man
        i. Face should change from alive to dead
        ii. Face becomes worried as game progresses
        iii. Add more parts
        iv. Hangman “drops” when game is over

We broke the changes required for this project into three parts, the main functionality of the game which was subdivided into the difficulty algorithm and the rest, and the art components.

Difficulty was at first a daunting task, in order to accomplish this we first needed to “solve” hangman. How difficult would it be to guess each word? Fortunately there is a lot of support online for this including an implementation in mathematica as well as python, unfortunately, neither of these languages convert well to java. But the basic concept was rather than attempting to play hangman with each of the words, we use a set dictionary list to optimize our playing algorithm taking advantage of the fact that we would already know what the perfect guess would entail at each step. To do this we first

established a new container class which would hold the completed list as well as the number of wrong guesses it took to find a given word in addition to the total number of guesses. Note that the total number of guesses doesn't really matter as far as the difficulty matters, assuming perfect playing, but it does help for the final sort and looking at it from a human playing aspect.

Sorted List is the first method needed. For each possible letter guess in each word, we needed a way to differentiate it from other words of its type. For example, if we wanted to look at the guess e in the word list 'deed even eyes mews peep star' we'd want the following grouping "['star'], ['mews'], ['even', 'eyes'], ['deed', 'peep']". As you can see words with the letter e in the same position, ignoring the value of the other characters, are grouped together. To accomplish this, we iterated over each letter in all the words, for each letter which matched the given guess, we bitshifted 1 by its corresponding position in the word then used this value as a key in a hashtable. This gives the following result set "[(0, ['star']), (2, ['mews']), (5, ['even', 'eyes']), (6, ['deed', 'peep'])]". Notice the key isn't its index in the list, rather it's a unique value. This method returns this hashmap.

Price of guess is the next method required. This method takes in the same parameters as the previous function, a list of words and a guess letter. Then it iterates over each word in the list looking for the guess letter. It then returns the number of words in the list which do not contain that letter. This will be used for finding the best letter to guess for a given list of words, as the best letter will have the highest likelihood of being correct if fewer of the given words don't have it.

Best guess is the penultimate method required and begins to draw on the stand alone methods above. The input parameters are a list of words and a string of letters which have already been guessed to get to that list. The method then calculates which would be the ideal next guess for that word list. First, using the full set of ascii lowercase characters, it eliminates any already guessed letters and stores this as an arraylist of unguessed letters. In set theory this is called the symmetric difference, or the combination of two sets without included the things which occur in both. Next for each letter in the unguessed letter set it computes the price of that letter using the price of guess function. The lowest price wins and is returned.

Word guesses is the final method and the most complicated. In order to check every word in the dictionary, we'll need to use recursion. So the first step is to add the recursion break. This is an if statement at the beginning of the method which checks to see if there's only one word left in the passed in array, if so we're done! And it adds that word, along with the correct and incorrect guesses needed to get to that word to our final results arraylist of type Hangman word, our container file from earlier which already has the variables needed to store all this.

Main word guessed section of the method is more complicated and contains a recursive call to itself. First the method calculates the best guess for the current words list for each letter that hasn't been used yet. Next it uses the best guess method to get a bestguess char which then is passed to sorted list to

make a hashmap of all words using that best guess. This hashmap will be a map of all the unique positions of each letter grouped together according to which letter should be guessed to eliminate the most wrong answers. But this doesn't give us any results, to get those we need to do this again on each list of words at each key in the table. This is the next for loop, for each entry in the hashmap, which is a list of words with a common positioning of best guess letter, we run the word guesses algorithm again. There is a special case of when we run this recursion and that's when the best guess is not in the word. This means that our key value for the map is going to be 0 because there is no character bestguess in the word. If this is true, we add one to the wrong parameter when we pass it into the function. This wrong value is set upon initialization of the overall function call. Our new words list which will be passed into the method again will be all the words with the same key value. Repeat this all the way down until each time we recurse there is only one possible word. Like we discussed about, when this happens we breakout and add the result to our results table.

Just when you thought we were done, we have a couple more steps to complete the final output. These final steps are found in the constructor method. First we need to read in our master list of words, we'll call this rawWords which is an arraylist of strings with each word in the master text file. Then we call the wordguesses method on this arraylist of words. As a final step we're going to want to sort our results arraylist of type hangmanWords. Rather than attempting to iterate through this list ourselves, we chose to provide the compareTo method within the class definition. This allows us to use the collections methods of sort on the arraylist of type hangmanWords. First we'll compare the number of wrong guesses required to find a given word, then we'll sort based on the total number of guesses required. Implementing this allows our final step in calculating the difficulty which is to sort the list. This is all done at the end of the constructor method and results in the final results list.

GUI Components/Features

The original version of the game was missing a number of features we wanted to add, and had a number of weird quirks we wanted to remove. First we added a menu bar to store the options on. First was New Game, which gets a new word and resets everything, pretty straightforward. Next was Exit, also straightforward. After that was Hint, which takes away 3 lives but fills out a word. It basically picks a random _ in the word, finds what it's supposed to be, and replaces it. Hint also needed to be disabled when the player had too few lives or was too close to finishing. Next were the difficulty buttons, which changed a variable that was called every time we used newWord(int i). Difficulty ranged from 0 (easy) to 2 (hard). This is pretty easy to change later, for example to implement an Elite difficulty or to change how words are assigned based on difficulty.

As far as misc. features go, we made the input ignore case, added invalid input text, and fixed up a few coding things.

Brandon Kirklen
Collier Robinson
Chris Peterson
2015-05-04
CPE 102-22

Art Stuff

We updated all of the art and the background, and gave the game the title “Hangman!”. Originally the background was to be a random color whenever the game was loaded, but this kept breaking the game and had to be removed; the background is now a bluish-green color. The gallows that came with the game seemed very minimal, so we redesigned it, making it more pronounced and removing drawing in the gallows as penalties for wrong guesses. In order to still give the user ten guesses, the hangman’s face now changes as he gets closer to his impending doom, with several of the face changes taking up turns. When the user runs out of guesses, the hangman “drops”, changes to a “dead” face, and the game is over. We also changed all of the original lines, making them thicker and more rounded. While this made the body and gallows look much better, it completely ruined the face. In order to fix this, we applied antialiasing to the lines, giving everything a much smoother appearance.