An Algorithm of Hangman Game by Zhen Wang

All the algorithm or code below is in the HangmanApi 'guess' function. This algorithm roughly consists of three parts and the following is a brief introduction of each part (a more detailed intro and discussion is at the second and final pages of this report):

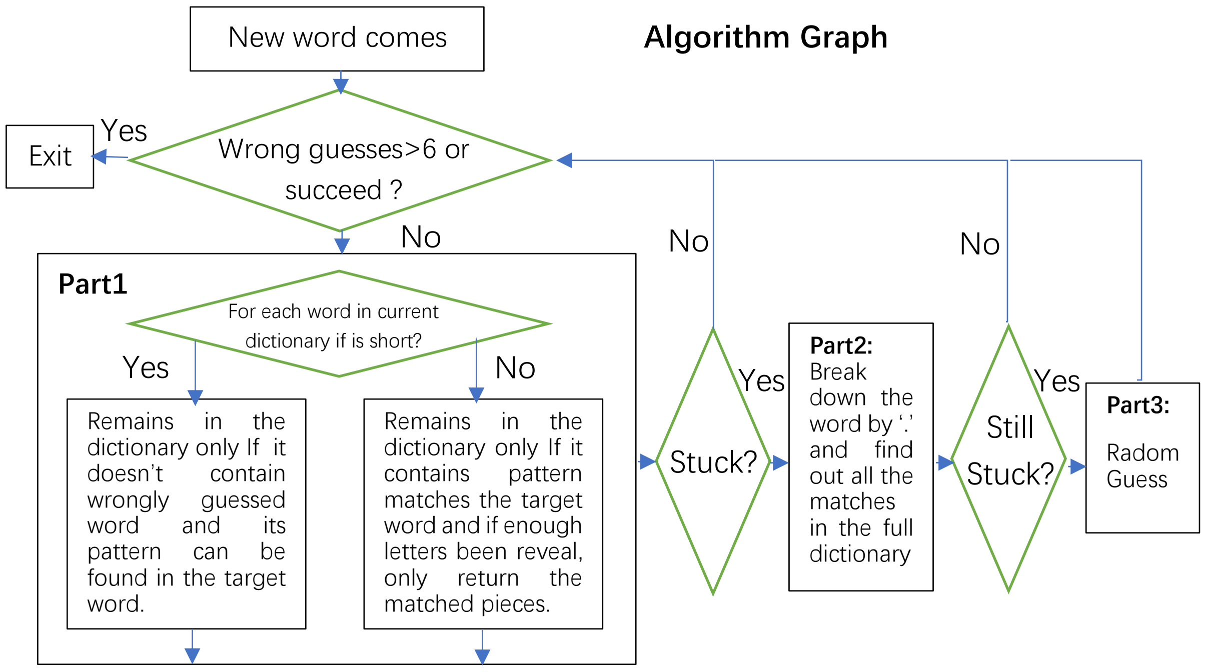
The general idea of each part is to generate a words dictionary for each word to be guessed in each round and returns the most appeared letter in this dictionary except those already been guessed. So what matters most is to generate a dictionary fitting the given word well since the latter part is simply counting.

A Brief Intro

Part1: strictly match: this one serves to guess at the beginning stage of the game when no or only a few letters has been correctly guessed and it can actually handle roughly *35% to 45%* of words. It basically creates a new 'reference words dictionary' (**current dictionary**) from the training data at the beginning of each game and as the game proceeds and hopefully more and more letters of the word are revealed, the scale of the dictionary shrinks to match the word more precisely.

Part2: match by decomposition: when the word is too 'special' that the first part doesn't work, that is, the first part stuck and cannot figure out a suitable letter to guess, this part takes charge of the game. It generally finds out all the '.' positions in the word (unguessed parts) and extracts the adjunct 4 letters together with these '.' and finds out all the matches in the training set (**full dictionary**). Because its search range is potentially much bigger than the first part so it uses all the words in the training set instead of those in the 'current dictionary'. This part roughly raises the prediction accuracy by *10%-20%*.

Part3: random guess: when all of the above stuck, this part gives out an answer of the most appeared words in the full dictionary except those letters that have been guessed. This part guarantees that this function always returns an answer.



New word comes
Algorithm Graph
Exit
Yes
Wrong guesses>6 or succeed ?
No
Part1
For each word in current dictionary if is short?
Yes
No
Remains in the dictionary only If it doesn't contain wrongly guessed word and its pattern can be found in the target word.
Remains in the dictionary only If it contains pattern matches the target word and if enough letters been reveal, only return the matched pieces.
No
Stuck?
Yes
Part2: Break down the word by '.' and find out all the matches in the full dictionary
No
Still Stuck?
Yes
Part3: Radom Guess

More Details

Denoting the word to be guessed as WORD.

Strategy 1 of Part1: it can be seen in the algorithm graph above that this part actually consists of two strategies that deal with longer words and shorter words separately. I chooses those shorter than WORD and longer than 0.58 times the length of WORD as shorter words and further screens these words by deciding **whether these words are 'elements' of WORD** using the standard: 1) if it contains incorrectly guessed letters, then deletes it from the 'current dictionary'; 2) replacing the letters in this word that haven't been guessed yet with '.' (pretending that they're unknown) and afterwards, if WORD doesn't contain the pattern same as the 'masked word' (using search function), then deletes it; 3) if none of the above happens, keeps it in the 'current dictionary'.

Why setting a lower bound of the short word? Because if too many extremely short words like 'aaa', 'a', 'aag' that in most cases would pass the following screening process but in most cases wouldn't provide any useful information and serves as noises, the predicting success rate might be lower. Why using 0.58 times the length of the word as the threshold? I have tested the algorithm both by setting the threshold at a fixed level, say 5, and also by setting at a floating level that equals a ratio times the length of the word. The problem of fixed level is that it is more time consuming and somehow couldn't outperform the floating level based on my tests (may be not enough). The ratio 0.58 is decided also by testing (also close to golden ratio by coincidence).

However, deciding whether a word is an 'element' of WORD is still a challenge for this algorithm.

Strategy 2 of Part1: for longer words I chooses those words longer than WORD. The main goal of this strategy is to decide **whether the WORD is an 'element' of a specific 'long word'**: 1) when there're less than 30% of letters of WORD revealed, I consider this situation as lack of knowledge so I keep all the words **in complete** whose **head or end** match the pattern of WORD (using re.search). For instance, if the word revealed is 'a..l.', I retain the whole word 'applepie'; 2) when there're above 30% of letters revealed, I assume this situation as informative, so I only keeps the part(s) matching WORD. For instance, if the word revealed is 'appl.', I only extract 'apple' from 'applepie'.

Why using different methods beyond or below 30%? Because if below, what we know is little and the reason we reject a word needs to be stronger than if we know a lot. Also, if we only keeps the matching part of each word from the beginning of the game, we will only have word pieces of the same length as WORD which gives us less information than if we take the words as a whole. For instance, matching '. . .' with true underlying word 'pie' may end up extract 'app' from 'applepi' which is a loss of information. As more and more information we obtain, it is 'noise' that we are more and more concerned about rather than the adequacy of information. Say we want to match 'pi_', we want to extract 'pie' rather than 'applepie' in this scenario. Why 30%? Also based on test but may lack of strong enough reason.

Why only consider words matching at the beginning or end? What about words matches WORD in the middle? This might also miss potential useful information. But I assume that most of the cases (or just enough cases) elements are located at the front or end of a word. Also if too broad range we consider when lacking information, too much noise may potentially be taken into account which will lower predicting power.

Part2: strategy 1 of part 1 may miss some useful words that might be elements of long words (because for long words elements may be possibly shorter than length of WORD times 0.58). This part aims to fix this issue by breaking down WORD into pieces and extracting the adjunct 4 letters together with these unguessed '.' and finding out all the matches in the training set (**full dictionary**) as described in the intro. Also, whether '.' is located at the head or tail of WORD is also considered. For example, if WORD revealed is '.ccom.lishmen.' with true underlying 'accomplishment', this method first transforms it into '_.ccom.lishmen._' where '_' marks beginning or ending of a word. It then extracts three parts from it since there are three unknown letters – '_.cco', 'om.li', and 'en._'. The next step is looking for word pieces that matches this pattern in the full dictionary. I connect all words in the full dictionary with '_' to mark the beginning or end of a word and find out all matching pieces using findall.

Why adjunct '4'? I assume that most useful word element can be effectively presented using 5 letters with 1 unknown even if their length might not be exactly 5.

Ways to improve

All the thresholds and numbers in the algorithm might seem to be too subjective and may need to be tuned simultaneously by more tests. These include the threshold to decide whether the word is short in part1 strategy1, the threshold of using a different method in part1 strategy2 and the number of adjunct letters in part2. Also, the overall and detailed procedure of the algorithm may also need to be optimized.

Thanks for reading!