Advanced Data Structures

Assignment - Word Games

Manuel Haag

14th April 2024

1 Introduction

A fun little game is to form as many words as possible from a multi-set of letters. For example using the letters in `abracadabra`, 4 words can be formed: `bad, bar, cab, card`. Making use of the Trie data structure we will design an algorithm to efficiently solve this problem, which we call *Word Challenge*.

Another quite popular game is *Wordle*. The standard rules[1] are as follows. The player has to find a secret 5 letter english word. To make a guess, he can enter any 5 letter word from specific dictionary of english words. Each guess reveals how far the player is from the secret word. A letter of the guessed word is colored in green, if the letter is at the correct position. Does the letter occur in the word, but not at the current position, it is marked in yellow. Otherwise the letter is marked in gray, meaning it does not occur at all in the secret word. Figure 1 shows an example of a wordle game with 4 guesses.

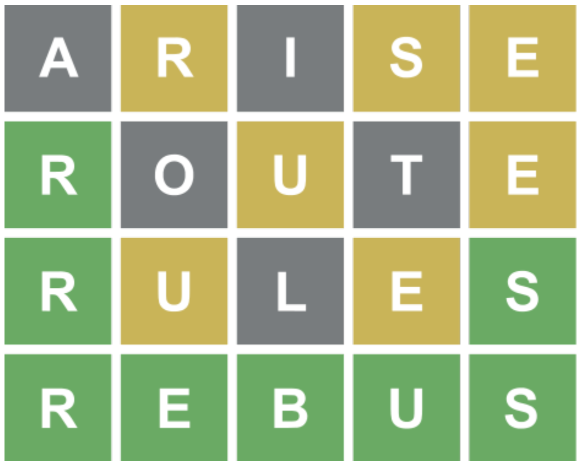


Figure 1: Wordle game #196 solved in 4 guesses.

The player wins the game if he can guess the secret word within 5 guesses. We will consider a variant of the classic *Wordle* game where the secret word can have any length and there is no limit on the maximal number of guesses. Our goal is to minimize the average number of guesses needed. In our work we implemented two heuristic strategies based on Tries and depth first search algorithms.

In the following sections we will present the Trie data structure and the search algorithms involved in solving these two problems. Finally, we show our results in an experimental evaluation.

[1]`https://en.wikipedia.org/wiki/Wordle`

We will use the following notation. Let Σ be an alphabet and $D \subseteq \Sigma^*$ a set of words. By $D_n \subseteq D$ we denote the subset of words of length n in D. For simplicity we assume Σ to consist only of the 26 english lowercase letters.

2 Trie

The Trie is the core data structure we will use to design algorithms for solving *Word Challenge* and *Wordle*. A Trie is a tree data structure for strings that allows to efficiently store a set of words $D \subseteq \Sigma^*$. It is a tree T where each edge is labeled with a letter $c \in \Sigma$. Traversing T from the root and concatenating the letters results in a word. Notice that the root node represents the empty word. A node $v \in V(T)$ has a flag indicating whether the word formed by the path from the root to v is contained in D. A word $w = w_1 w_2 \dots w_n$, $w_i \in \Sigma$, is inserted as follows into the Trie. We traverse T starting from the root and in the i-th step we follow the edge with label w_i. If this edge does not exist, we simply create this edge, a new node as a child and add them to T. After the n-th step, we flag the current node to indicate that $w \in D$. In our algorithms D will be know in advance, so we construct the Trie once at the start by inserting each word in D.

3 Word Challenge

The input to our program is a multi-set of letters S, which we will represent by an array of 26 integers. To convert letters to array indices, we simply subtract the smallest letter, in our case `a`. The i-th entry stores the number of occurrences of the i-th letter in S. Our algorithm computes a list of words W that can be formed by a subset of S. Listing 1 summarizes our algorithm. We perform a depth first search on the Trie T starting at the root. During the search we keep track of the current word formed by the path from the root to the current node v. If the current word w is indeed a word from the dictionary D, i.e. the flag of the current node is set, we add it to the output (line 7). Each time we traverse an edge with letter and target node (l, u), we decrement the count $C[l]$, since the current word uses an additional l. Conversely, when we backtrack, we increment the count of l again. We can only traverse an edge, if there is still at least one letter l available. Thus line 5 prunes subtrees that do not contain relevant words.

Algorithm 1: Word Challenge Search

```
   Input  : multi-set $S$, Trie $T$
   Output: list of words $W$
 1 $W \leftarrow \{\}$
 2 $C \leftarrow S$                                  /* counter initialized with multi-set */
 3 Function RecursiveSearch($v$, $w$):
 4     for $(l, u) \in N_T(v)$ do
 5         if $C[l] > 0$ then
 6             $C[l] \leftarrow C[l] - 1$
 7             $w' \leftarrow w \cdot l$
 8             if isWord($w'$) then
 9                 $W \leftarrow W \cup w'$
10             end
11             RecursiveSearch($u$, $w'$)
12             $C[l] \leftarrow C[l] + 1$
13         end
14     end
15 RecursiveSearch($0$, "")                                    /* start from the root */
```

The list of words W is implemented by a simple vector and new words are appended at the end. An additional requirement in the exercise statement is to output W sorted first by length and then by lexicographic order. We do this by storing words of length i in separate vectors W_i and merge by increasing i. We traverse the children nodes $N_T(v)$ sorted by the letter l attached to the edge. Thus we find the words in W_i already in the correct order.

4 Wordle Game

Before we introduce the guessing strategies for *Wordle* we explain how we represent the set of constraints given by the previous hints and how we compute the set of words fulfilling the constraints.

4.1 Modeling the Constraints

Let n be the length of the secret word. We use 3 arrays K, L, U and a boolean matrix M to store the constraints. The first array K represents known letters, which are indicated by green letters in the hint. For each position $i \in \{1, \dots, n\}$ with know letter $l \in \Sigma$ by some hint, we set $K[i] = l$. The arrays L and U store a lower and upper bound for the number of letters $l \in \Sigma$ in the secret word. We initialize L with 0 and U with n. Let $g(l)$ and $y(l)$ be the number of green and yellow l in the current guess. For each hint we update $L[l]$ by $\max(L[l], g(l) + y(l))$, since at least all yellow and green letters have to appear in any candidate. We update the upper bound U by $\min(U[l], n - k)$, where k is the total number of known letters $l' \neq l$ of the secret word. The number $n - k$ is the maximal number of positions that l can have in the current guess. If there is at least one gray l, we know that there are exactly $g(l) + y(l)$ letters in the secret word, so we set $L[l]$ and $U[l]$ to $g(l) + y(l)$. The matrix M stores flags, where $M[i][l] = 1$ means that the letter at the i-th position can not be l. We set the appropriate flag whenever we receive a yellow hint for a letter.

4.2 Candidate Search

Listing 2 summarizes the candidate search algorithm. Any valid word in T at depth n that fulfills all constraints is a candidate. We perform a depth first search on the Trie T starting from the root with depth $d = 0$ until a maximum depth n. During the search we will prune subtrees that violate constraints as early as possible. The vector W stores found candidate words, the array F stores at $F[l]$ the count of l in the current word. In line 14 we prune subtrees that can not fulfill the constraints, since there are more missing letters than steps until depth n left. In particular, if a letter $F[l]$ exceeds the upper bound $U[l]$, the search is pruned as well. When we reach a depth of n we add w to W, if it is a valid word (line 17). In line 27 we check, which subtrees we have to search recursively. The condition ensures that if we know the letter at position d, we will only search this subtree. If we do not know the letter, we search subtrees of letters l that are not yet marked as not existing ($M[i][l] = 0$) and do not violate the upper bound. Notice that the candidates W could still contain words we already guessed, so after the candidate search we remove them to avoid duplicate guesses.

```
Algorithm 2: Wordle Canditate Search
Input  : constraint arrays $K, L, U$ and matrix $M$, Trie $T$, secret word length $n$
Output: list of candidates $W$
$W \leftarrow \{\}$
$F \leftarrow [0, \dots, 0]$
Function missingLetters():
    $x \leftarrow 0$
    for $l$ = a to z do
        if $F[l] \leq U[l]$ then
            $x \leftarrow x + \max(0, L[l] - F[l])$
        end
        else
            return $\infty$                    /* more letters $l$ than upper bound */
        end
    end
    return $x$
Function CandidateSearch($v$, $d$, $w$):
    if $n - d <$ missingLetters() then
        return                                  /* prune search */
    end
    if $d = n$ then
        if isWord($w$) then
            $W \leftarrow W \cup \{w\}$
        end
        return
    end
    for $(l, u) \in N_T(v)$ do
        $w' \leftarrow w \cdot l$
        $b_1 \leftarrow K[d] = l$                         /* letter is known */
        $b_2 \leftarrow K[d] = 0$ and $F[l] + 1 \leq U[l]$ and $M[d][l] = 0)$   /* search subtree */
        if $b_1$ or $b_2$ then
            $F[l] \leftarrow F[l] + 1$
            CandidateSearch($u$, $d + 1$, $w'$)
            $F[l] \leftarrow F[l] - 1$
        end
    end
CandidateSearch($0$, $0$, "")               /* start from the root with depth 0 */
```

4.3 Guessing Strategies

Now we explain the two guessing strategies. The first strategy simply guesses a candidate $w \in W$ uniformly at random. Since we remove old guesses from W, we are guaranteed to find the secret word at some point. We refer to this strategy as **RandomCandidate**.

There is one major downside to **RandomCanditate**. It ignores that certain guesses have a higher chance of reducing the number of candidates than others. In certain cases it can even be beneficial to guess a word, which violates the current constraints, but greatly reduces the candidates. In *Wordle* these kind of guesses are allowed[2]. To make an informed guessed, we compute a score $f(w)$ for each word $w \in D_n$ and guess the word with the highest score. Let $c_l(w)$ for $l \in \Sigma$ and $w \in \Sigma^*$ be 1, if the word w contains the letter l and 0 otherwise. Let $\Sigma' \subseteq \Sigma$ be the set of letters with $L[l] = 0$ and $U[l] \neq 0$, i.e. letters for which we have no received a hint yet. Let p_l be the

[2]there is a hard mode in which the guesses also have to fulfill the constraints

3	iao
4	orae
5	serai
6	serian
7	erasion
8	serotina
9	tensorial
10	psalterion
11	ulcerations
12	culteranismo
13	postneuralgic
14	neuroplasmatic
15	trigonocephalus
16	trigonocephalous
17	superromantically
18	pseudoromantically
19	pseudohallucinatory
20	encephalomyocarditis

Table 1: Words with the highest score in the first guesses for **LetterFrequency** strategy.

probability that a candidate in the set W contains the letter $l \in \Sigma'$.

$$p_l = \frac{\sum_{w \in W} c_l(w)}{|W|}$$

We define the score function f as follows.

$$f(w) = \sum_{l \in \Sigma'} c_l(w) \cdot p_l$$

So words with a high score contain letters where we have no hints yet and that are frequent in W. Note that the first guess is based on no hints, so it will always be the same and can be precomputed. Table 1 shows the words used in the first guess. Once there are less than 10 candidates, we follow the **RandomCandidate** strategy. We call this strategy **LetterFrequency**.

5 Experiments

In this section we conduct a experimental evaluation of our algorithms. We implemented[3] our algorithms in C++20 and compiled with optimization flags `-O3 -march=native`. We ran our experiments on a laptop with Ubuntu 20.04.6 LTS, a AMD Ryzen 5 5600H CPU clocked at 3.3 GHz, 16 GB RAM and with a L1, L2, L3 cache of 192 KiB, 3 MiB and 16 MiB respectively. For our experiments we will use a dictionary D with 370105 english words consisting of the 26 english letters[4], where we removed words with less than 3 or more than 20 letters, since there are very few of them. Figure 2 shows the distribution of words lengths in D. The Trie build from D consists of 1027817 nodes.

5.1 Word Challenge

For the *Word Challenge* we run our algorithm for each $m \in [3, 20]$ with 1000 random words sampled from D_m. Figure 3 shows the results. We can solve the word challenge for each word length in

[3]code availabe at: `https://github.com/HaagManuel/WordGames`
[4]dictionary from: `https://github.com/dwyl/english-words/blob/master/words_alpha.txt`

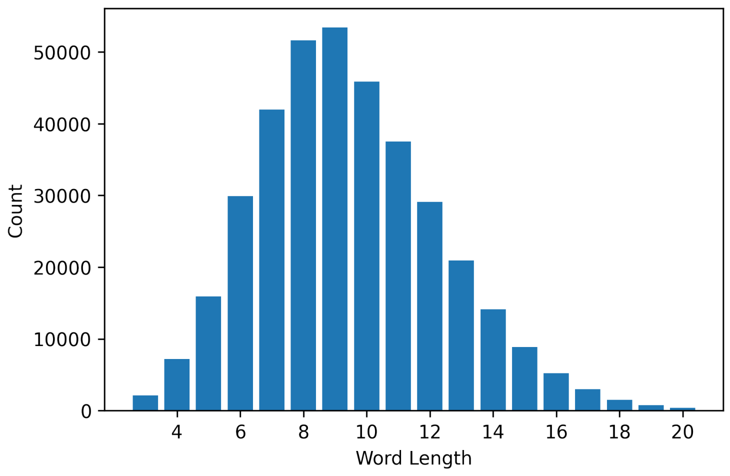


Figure 2: Distribution of word lengths in D.

under 1 millisecond. As expected, the time needed is proportional to the number of nodes visited. For words shorter than 10, there are less than 500 results, since the constraints are relatively strict. For words longer than 10, the result size is approximately 30% of the visited nodes. Recall, that there are around 1M nodes in the Trie and 300K words in D. Our algorithm successfully prunes subtrees and reduces the search to less than 3% of the tree.

5.2 Wordle

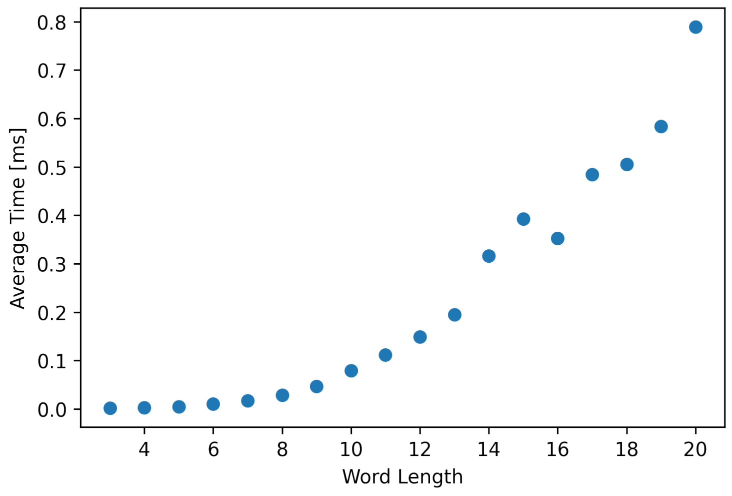
To evaluate our algorithms for *Wordle* we simulate both our strategies for each $m \in [3, 20]$ in 100 games with random secret words of D_m. Figure 4a shows time needed to solve *Wordle*. For **RandomCandidate** the run time is dominated by the candidate search. It increases with the word length, since the depth first search has higher depth limit and takes on average between 0.2 and 3.5 milliseconds. In the **LetterFrequency** strategy the time to compute the score function for each word of length m is the dominating factor. This is why the curve is similar to the word distribution. For $m \in [6, 10]$ it takes about 4 to 5 milliseconds, otherwise around 1 to 2 milliseconds.

Figure 4b shows the average number of guesses to solve the *Wordle* game. In general shorter words require more guesses. This is because we get less information about the letters per guess. For words longer than 14 letters, most of the time the words can be guessed with 2 tries. The informed **LetterFrequency** needs on average 0.5 to 1.5 guesses less for words shorter than 6. However, for words with 7 to 13 letters, the difference is smaller with 0.1 to 0.3 guesses on average.

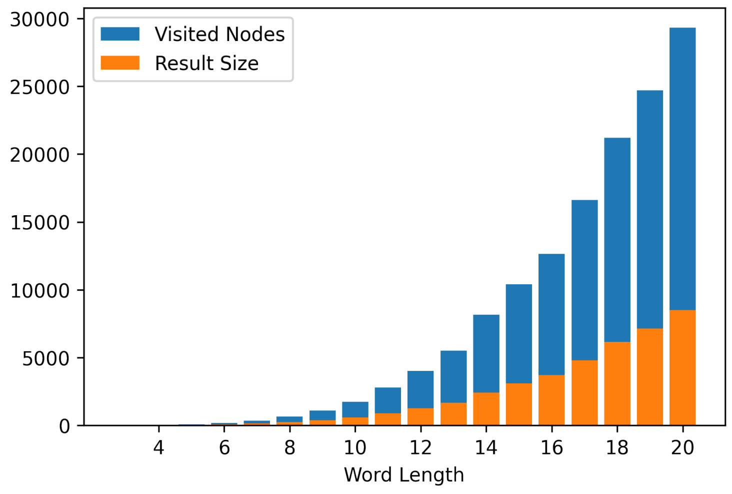
Looking at Figure 4c, we see that the percentage of candidates left after the first guess drastically decreases. For words with 6 or more letters there are less than 5% of candidates left. Again shorter words give less information per hint, so the reduction is smaller.

6 Conclusion

In our work we showed that the rather simple Trie datastructure can be used to implement efficient search in a dictionary. We applied depth first search in Tries to solve the *Word Challenge* and *Wordle* game. In both cases the search for potential solution candidates reduces quickly to few candidates by pruning the search early. Even though we used a large dictionary of around 300000 words that almost contains all english words, we were able to solve the games in a matter of milliseconds.

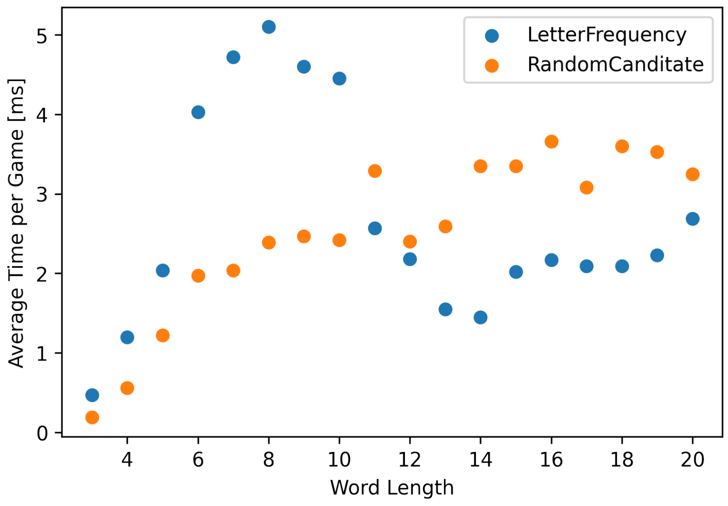


(a) Time to solve *Word Challenge*.

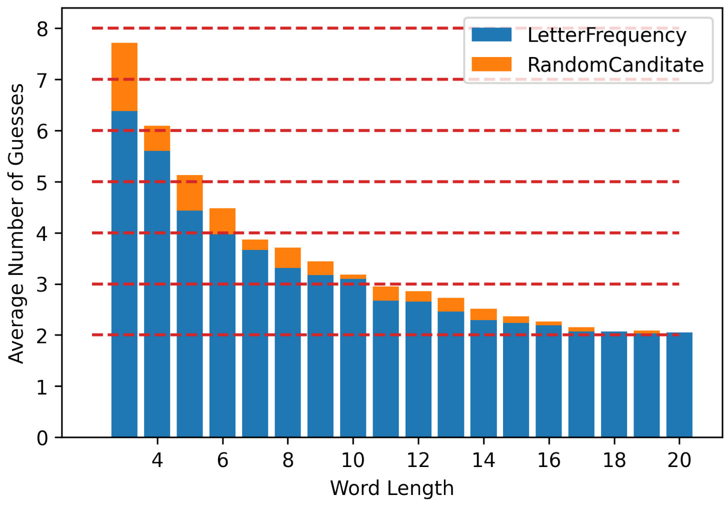


(b) Number of visited nodes and result size.

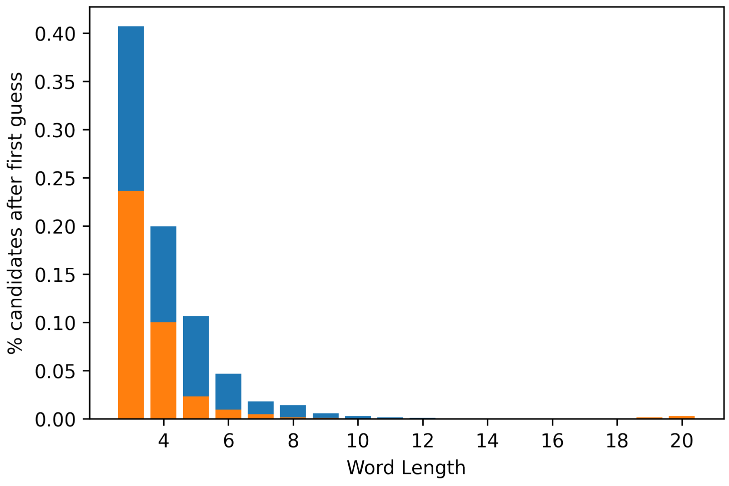
Figure 3: Results of solving *Word Challenge*.



(a) Time to play one game of *Wordle*.



(b) Number of guesses to solve *Wordle*.



(c) Percentage of candidates after the first guess.

Figure 4: Result of *Wordle* experiments.