Scrabble Game Analysis Program

By: Qusai Elwazir

CS-310
May 22, 2025

Table of Contents

1 Introduction:

This program was created as a project for an advanced data structures and algorithms course. The goal was to create a program that could assess any game state of scrabble and output the optimal move.

This document is organized as follows:

- Sections 2, 3, and 4 describe program methodology and algorithm analysis rather than actual implementation.
- Sections 5 and 6 focus on the real implementation and runtime of the algorithms described in preceding sections.
- Section 7 is an analysis on the computational problem described by the game Scrabble.

2. Dictionary Processing:

2.1 Dictionary Constraints:

For this project I used the **Official Scrabble Players Dictionary** which is composed of just under 300,000 words with a length of less than or equal to 15 letters. The criteria by which the program must search this dictionary is by set of letters due to the nature of scrabble. The set of letters that we will search by is of maximum length 15. If we were to do this with a brute force algorithm we would, in the worst case, make 15x300,000 comparisons per word search. That is not acceptable. In order to solve this problem we need to run a preprocessing algorithm on the dictionary to make it easily searchable by a set of letters. What we are looking for in our final processed structure is a map of unique keys that pertain to sets of words that share letters.

2.2 Fundamental Theorem of Arithmetic and Application:

The fundamental theorem of arithmetic states that every positive integer, n, is the product of the elements of exactly one multiset of prime numbers.

$$\forall n \in \mathbb{Z}^{+} \ where,\ k, n_i \in \mathbb{N},\ p_i \in \mathbb{P} \ it\ is\ true\ that,\ n = \prod_{i=1}^{k} p_i^{n_i}$$

We can use this fact to codify words, and sets of words, as numbers. To do this we will assign each letter in the alphabet a prime number so that a complete word can be represented as the unique product of its letters. In this way we can consolidate words that share the same composition by referring to the set of all words that are a composed of a given multiset of primes by the product of the elements of that multiset. We can use this to organize our dictionary into a map whose keys are products whose prime factorizations relate to a set of letters, and whose values are the sets of all words that are composed of that related set of letters. Pseudocode for this process can be found in figure 2.2 on the next page.

Figure 2.2.1, Dictionary Processing Algorithm:

```
process_dict(alphabet, dictionary): ## O(a + d)
     FOR l IN alphabet: ## O(a) for alphabet of size a
          l.id = i_th_prime(i)
     map E(int, set)
     FOR w IN dictionary: ## O(d) for dictionary of size d
          key = product_of(letter_ids(w))
          IF key NOT IN E:
               E(key, new set)
               E(key).add(w)
          ELSE:
               E(key).add(w)
     return E
```

Figure 2.2.1: The processing algorithm has parameters of ‘alphabet’ and ‘dictionary’ where alphabet is a list of letters and dictionary is a list of words composed of only the letters in ‘alphabet’. This algorithm first assigns each letter in the alphabet a prime number. It then initializes E (type map) which will be the processed dictionary output of this algorithm. Next, for each word in the dictionary, it takes the product of its letters assigned prime numbers and assigns ‘key’ that value. If this product is novel to the map, it will create a new entry in the map with key: ‘key’ and value: ‘new set’. It then adds the current word to that set. If the product of letters (key) is not novel to the map, then it will simply add the current word to the set associated with that key value. It will then return the map E which contains key value pairs of prime products of letters and the set of all words composed of the letters associated with that prime product respectively. This algorithm runs in linear (O(n)) time where ‘n’ is the sum of the lengths of the input lists.

3. Dictionary Searching:

3.1 Search Constraints:

Using the processed dictionary we must be able to: 1. Tell if a string of letters is a word, 2. Get the set of all words composed of a set of letters, 3. Get the set of all words that are composed of the elements of the power set of a set of setters 4. Get the set of all words that are composed of the elements of the power set of a set of setters and that contain a substring "s".

3.2 Word-Truth Value of Strings:

In order to tell if a word is a string we must first find the set of all words that are composed of the letters in that string. Then we must compare that string to every word in that set. Where the length of the string is 'L' we will have at most ${}^{L}P_{L}$ *or* $L!$ elements in the set related to the string we are assessing.

Figure 3.2.1, Word Truth Value Algorithm:

```
is_a_word(word, map): O(n!)
          A = get_all_words_with_composition(word, map) # O(1)
          FOR real_word IN A: # O(n!) = O(P(n,n))
               IF word == real_word:
                    return TRUE
          return FALSE
```

Figure 3.2.1: The word truth value algorithm takes parameters 'word' which is a list of letters, and 'map' which is a map type that was created running the algorithm described in figure 2.2.1 on a dictionary (list of words). First this algorithm calls 'get_all_words_with_composition', which is the function described in figure 3.3, and sets A equal to that set. Then for each word in A ('real_word') it compares the argument 'word' to 'real word' and returns true if they are equal. If the string argument 'word' is not a word this loop must iterate through all of A until it returns false. This algorithm runs in $O(n!)$ but as described in section 3.6 this results in O(1) when implemented.

3.3 Finding Words of an Exact Composition:

Many tasks in this program rely on being able to search the dictionary in the most basic way. That is, to find the value associated with a key in a processed dictionary. This process is only used directly to check the word-truth value of strings, but is also used to implement other, more important algorithms.

Figure 3.3.1, Set of All Words of Composition X Algorithm:

```
get_all_words_with_composition(letters, map): ## O(1)
     key = 1
     FOR letter in letters ## O(len(letters)):
          key *= letter.id
     return map(key) #O(1)
```

Figure 3.3.1: The Set of all words of composition X algorithm takes parameters ‘letters’ which is a string of characters, and ‘map’, is a map type that has been created by running the algorithm described in figure 2.2.1 on a dictionary (list of words). This algorithm first initializes ‘key’ to an impossible key value. It then finds the product of the associated primes of all elements in ‘letters’ and stores it in ‘key’. It then uses this key to search the ‘map’ argument and return the set type value associated with ‘key’. Since this algorithm simply searches a map it runs in constant time.

3.4 Finding Words of Broad Composition:

When searching for the set of playable words we must account for every word that can be made with a set of letters. The algorithm described in figure 3.3.1 fails to do this because it returns only the set of words that are composed of exactly the set of letters it takes as an argument. That being said, searching for exact compositions is useful for implementing all algorithms described in this section 3. In order to find all words that can be created with a given set of letters we must find all subsets of our original set and run algorithm 3.3.1 on each of them.

Figure 3.4.1, Set of All Words with composition Powerset-X Algorithm:

```
get_all_words_power_set(letters, map):
     list E = list(power_set(letters))
     set A
     FOR e in E: ## O(2^n) elements in E
          key = 1
          FOR letter in e: # O(n)
               key *= letter.id
          A.add_elements(map(key)) # O(1) but n! Elements added
     return A
```

Figure 3.4.1: The set of all words with composition, Powerset X algorithm has parameters of 'letters' which is a list of characters and 'map', which is a map type that has been created by running the algorithm described in figure 2.2.1 on a dictionary (list of words). It then initializes a list 'E' to be the list of all elements in the powerset of the set 'letters'. Initialize set type A. Then, for each element 'e' in 'E', 'key' is first set to an impossible value of 1 and is then set to be the product of the associated primes of the letters that compose the string argument 'letters'. It then searches the argument 'map' using the 'key' (which is the key corresponding to the current element in the powerset of 'letters') to find the set of all words that are composed of that subset of 'letters'. It then adds the elements of that set to A. This is repeated until all subsets of 'letters' have been searched by performing map(key). It then returns 'A' which will contain all words that can be created with a given set of letters 'letters'. This algorithm has a high complexity due to the complexity of finding power sets. It runs in $O(2^n + n \cdot 2^n)$ where n is the length of the argument 'letters'. The $n \cdot 2^n$ term comes from finding the powerset of 'letters' and the 2^n term comes from iterating through E which is of length 2^n. This algorithm is the most complex algorithm implemented in this program and is the cause of all complexity growth when we analyze the unbounded problem of scrabble in section 7.

3.5 Finding Words Containing Substrings:

When looking for words on a board we don't only take into account the letters in our hand and letters we may potentially intersect with. We also take into account multi-intersection words. For example if ‘bird’ is on the board, you may intersect on every letter of ‘bird’ to create the word ‘birds’ even if the letters ‘b’, ‘i’, ‘r’, and ‘d’ are not in your hand. This means that we must be able to search using static substrings. We must do this by adding the given substring to the string we search by and then for each search, find the intersection between the found set of words containing the additional letters in the substring and the set of words containing the exact substring.

Figure 3.5.1, Set of All Words with Substring X algorithm :

```
get_all_words_with_substr(letters, substring, map):
     set E = get_all_words_power_set(letters + substring , Map)
     list A
     FOR e in E: ## E is worst case size n!(2^n)
          IF substring in e:
               A.add(e)
     return A # worst case size n!(2^n)
```

Figure 3.4.1: The set of all words with substring X algorithm has parameters ‘letters’ which is the list of letters that we are searching by, ‘substring’ which is a list of letters and ‘map’ which is a processed dictionary. This algorithm first runs the algorithm described in figure 3.4.1 on the sum of the list of letters ‘letters’ and the list of substring. Then it initializes A which will be the set of all words containing substring ‘substring’. Then for each element ‘e’ in ‘E’ we check if the ‘substring’ is a substring of e and if it is we will add ‘e’ to ‘A’. We will then return ‘A’. Since this algorithm relies on the algorithm described in figure 3.4.1 and must iterate through the output of algorithm 3.4.1 it will run in $O(n! \cdot 2^n + 2^n + n \cdot 2^n)$ the additional $n \cdot 2^n$ term is a result of iterating through the elements of E which is size $n! \cdot 2^n$ in worst case..

3.6 Implicit Dictionary Bounding

The most frequently used algorithm described in this section will be the algorithm described in figure 3.2.1. This algorithm checks if a string is a word and runs in $O(n!)$ time where n is the number of letters in the string. The frequent use of algorithm 3.2.1 will introduce many $n!$ terms in the time complexity analysis of later algorithms discussed in section 4. For this reason it is important to discuss the nature of this term and its relationship to its input. This term is based entirely on the number of words in a value set of our processed dictionary map. This set seemingly will be of length $n!$ this means that we must do at most $n!$ comparisons to determine if a string of that composition is a word. This will not be the case because of two powerful implicit bounds on the dimension of human language. Our first powerful implicit bound is the limit of the length of words in human language. In all human language the longest word is a finite number. The longest word in the English language is 45 letters meaning we have $O(n!)\ with\ n \leq 45$ for english and $O(n!)\ with\ n \leq C$ for all of human language. The second powerful implicit bound is related to the frequency of words by length in human language. Figure 3.6.1 shows how the frequency of words in the English language increase as the length of the word approaches 5. The frequency of words drops off quickly as we approach 0 from 5 and as we approach 10 from 5. This is extremely significant because it shows that the probability of multiple words being the same letters as other words when n is large is extremely small. This means that we will likely only run into a large number of 'collisions' in our dictionary map when n is small (approx. $n < 7$). Stated more clearly:

$$where\ f(w_i)\ is\ the\ time\ complexity\ of\ algorithm\ 3.2.1\ for\ an\ input\ word\ w\ of\ length\ i\ and\ C\ is\ a\ natural\ number\ that\ is\ the\ maximum\ length\ of\ words\ in\ a\ language$$

$$\lim_{i \to C} P(f(w_i) = O(n!)) = 0, and \lim_{i \to 0} P(f(w_i) = O(n!)) = 0$$

This means that for all of our $n!$ terms related to algorithm f are $\Theta(\mu)$ where μ is the expected value of probability density function related to figure 3.6.1. With this information we can say that our $O(n!)$ terms run in constant time due to the bounded dimensions of linguistic dictionaries.

Figure 3.6.1 English Language Word Frequency by Word Length

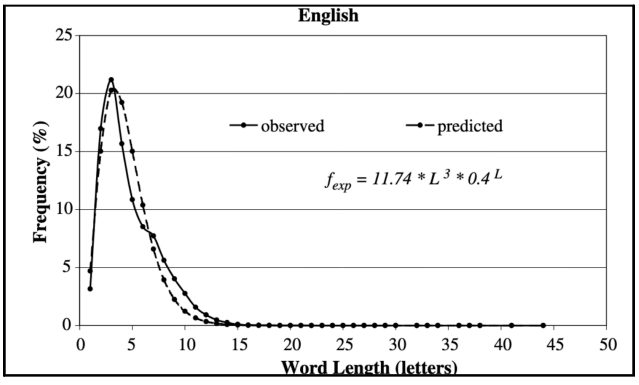

Figure 3.4.1 is taken from WORD LENGTH, SENTENCE LENGTH AND FREQUENCY – ZIPF REVISITED by Bengt Sigurd, Mats Eeg-Olofsson & Joost van de Weijer

4. Scrabble Game Description and Analysis

4.1 Game Structure Analysis and Bounding

To describe scrabble we use an $n \times n$ array where each element contains all relevant information about the i^{th} j^{th} space. The general method by which we will assess a gamestate will be driven by the nature of legal scrabble moves and how they coalesce around the 'island' (the collection of already played words on a board). There is one primary quality of scrabble moves that we will use to do gamestate analysis. All legal scrabble moves must have some intersection with either the center square or a space that is a member of the island where scrabble moves, with respect to intersection, are defined as the set of all spaces on which new words are created by a play.

4.2 Finding All Legal Moves

In order to find the maximal move we must use a greedy approach, not ruling out any option. This means that we must find every possible move for a given state. To do this we will use previously described algorithms to iterate through our $b \times b$ scrabble board and find every possible move on every space of the board. The algorithm used to do this is described below.

Figure 4.2.1, Finding All Legal Moves Algorithm:

```
find_all_moves(hand, game_state):
     LIST E # legal moves
     FOR (i,j) IN game_state:
          IF board[i][j].has_neighbors():
               A=find_all_neighboring_strings(board[i][j])
               B #possible words
               FOR string IN A:
                    B += get_all_words_with_substr(hand,
                                                   substr, dictionary)
               FOR word IN B: # worst case size b²n!(2ⁿ)
                     FOR l IN word:
                       IF are_all_words(intersecting_strings(l))
                         E.add(word)
     Return E
```

Figure 3.4.1 The 'find all moves' algorithm has two parameters: 'hand' which is a string type containing the letters in a player's hand and 'game_state' which is an object type that contains all relevant information about the current scrabble game state. This algorithm first initialized a list 'E' which will be the list of all legal moves for the game state given by 'game_state'. It then iterates through every space on the board associated with 'game_state'. For each space: it first finds all of the strings on the island that are adjacent to the current space. It then initializes B which is a tentative list of possible playable words. Then, for each adjacent string to the current space, it will search the dictionary associated with 'game_state' using the algorithm described in figure 3.5.1 for playable words containing those adjacent strings and add them to B. Then for each word in B and for each letter in each word, it will find all of the intersecting strings over the span of that created word and check if each created string is also a word. If all created strings are also words then this word will be added to the list of all playable moves. This algorithm runs in $O(b^2(n! \cdot 2^n + 2^n + n \cdot 2^n + n!^2(2^n))$ where n is the length of the 'hand' argument and b is the dimension of the board. We get the b^2 term from iterating through the entire board and it is multiplied over all other terms since all subsequent operations are done b^2 times. The $n! \cdot 2^n + 2^n + n \cdot 2^n$ term is the result of finding all words with substring at each space (algorithm 3.5.1). The final $n!^2(2^n)$ term is a result of the last nested for loop in the above algorithm: This loop must iterate through $n!$ (2^n) elements and perform $n!$ operations on each iteration when determining if the adjacent created strings of words in B are words (algorithm 3.2.1). As shown in section 6.1 the it is extremely probable that the n factorial terms will drop out giving us a time complexity of $O(b^2(2^n + 2^n + n \cdot 2^n + (2^n))$. For real implementation of scrabble n is also bounded at 7 and b is bounded at 15, giving us very computationally feasible complexity despite the unreasonable theoretical growth in complexity of this problem.

4.3 Finding the Maximal Scoring Move

To find the maximal scoring move we must simply find the maximal point value word in the list outputted by the function described in figure 4.2.1.. This total process will run first algorithm 4.2.1 giving $O(b^2(2^n + 2^n + n \cdot 2^n + (2^n))$ runtime where n is hand length. It must then iterate through the resulting list of all playable moves which is of length: $b^2 n!\, 2^n$, where b is the dimension of the board and n is the number of playable letters in the hand and at and adjacent to a given space. Therefore, our maximal scoring move algorithm will run in $O(b^2(2^n + 2^n + n \cdot 2^n + (2^n) + b^2 n!\, 2^n)$ time. Applying our bounding from section 5.6 we get that our maximal move algorithm runs in $O(b^2(2^n + 2^n + n \cdot 2^n + 2^n) + b^2 2^n)$ or $O(b^2(3 \cdot 2^n + n \cdot 2^n) + b^2 2^n) = O(b^2(2^n + n \cdot 2^n) + b^2 2^n) = O(b^2(n \cdot 2^n) + b^2 2^n)$ which gives us a final complexity of $O(b^2 2^n(n + 1))$

5 Program Structure

This section is meant to be a guide to understanding the code base associated with this project and is meant to be read alongside the **source code**. This project was written entirely in Python and uses only standard python libraries. This program consists of two main sections. The first section of the program is related to the dictionary and consists of two classes: dictionary_processor and dictionary_searcher. The second section of this program is related to the analysis of Scrabble game states and consists of two classes as well: move_analyzer and board processor.

There are also 3 demo classes named demo1 through demo3. These classes are simply preliminary classes used to exhibit the behaviours that the core classes of the program are capable of exhibiting.

To find the maximal move, the program takes in a gamestate represented by a string as an argument. It also takes a specific dictionary as an argument and an alphabet related to that dictionary as an argument. It then runs the preprocessing algorithm (figure 2.2.1, dictionary_processor: line 18) on the given dictionary and alphabet. It then creates a Game_State object (board_processor line 19) that describes the given string input describing a gamestate. It then uses both the created map describing the dictionary and the Game_State describing the board and its spaces to find the optimal move using the algorithm described in figure 4.2.1 (move_analyizer line 45). It then returns a Move object (move_analyizer line 7). This move object is the final output of this program and contains all relevant information about the maximally scoring move for a gamestate of scrabble.