Measuring Methods of Solving the NYT Spelling Bee

Rowan Hennessy

Abstract

An informal personal project exploring the properties of the New York Times game `Spelling Bee`. In this paper, our goal is to find patterns or distributions of solutions to the game, explore different methods of "solving" the puzzle, given different win conditions, and generally learn as much as possible about the application of algorithms to an inherently human game.

1 Introduction

Recently, I came across a video by `3Blue1Brown` about solving Wordle using Information Theory, and I thought it would be interesting to explore methods of solving a different New York Times game- the Spelling Bee. Unlike Wordle, there is no information "missing" at any stage of the game. Rather, the user is given a "hive" of characters, and is tasked with combining the middle character with any subset of the other characters to create as many words as possible.

The difference between efficient algorithms to solve this problem compared to a network flow or stable matching problem interested me, as we have specific bounds on n, and I thought looking deeper into this problem would be a good opportunity to practice Python, as well as see if there was anything interesting hidden in this simple game.

This paper began as simply examining how a few solving methods would measure against each other, seeking a single most-efficient implementation to solve the problem. While looking for ways to optimize solving the problem, however, I found analysis of the game itself interesting, which gave rise to section 3.

2 From Online Game to Programming

2.1 Game Explanation and Formalizing

If you're unfamiliar with the New York Times' Spelling Bee puzzle, the rules are quite simple. Opening the link displays something similar to Figure 1 below, with six outer characters and one highlighted inner character.

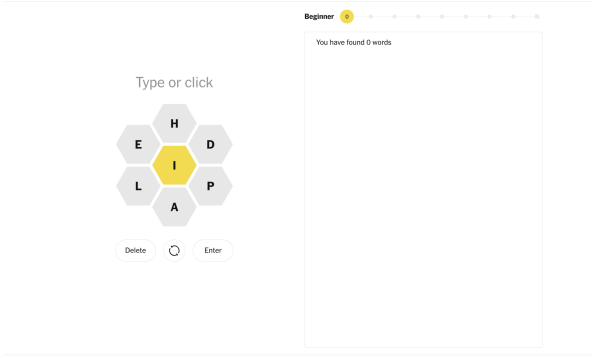

Figure 1:
The opening screen of Spelling Bee

The user inputs a word made of any combination of these characters, and if that word is valid, they are awarded the appropriate amount of points. A word ω is deemed valid if it has the following

properties:

1. The inner character of the hive is contained in ω
2. The length of ω is greater than 3, less than 20
3. ω is not obscure, hyphenated, a proper noun, or an expletive

Upon entering a valid word ω, the user's total score is increased by $f(\omega)$ points, where f is the following function:

$$A = \{\omega \mid \omega \text{ is a valid word and contains } \textit{every} \text{ character in the hive}\} \tag{1}$$

$$f(\omega) = \begin{cases} 1 & \text{if len}(\omega) = 4 \\ \text{len}(\omega) & \text{if len}(\omega) > 4 \text{ and } \omega \notin A \\ \text{len}(\omega) + 7 & \text{if len}(\omega) > 4 \text{ and } \omega \in A \end{cases} \tag{2}$$

Note that A represents the set of "pangrams", which are valid words that contain every character in the hive. It's given to us that each puzzle contains *at least one* pangram, which are worth an additional 7 points. Clearly, every valid word is either a pangram or not, and so any valid word ω has a (not necessarily distinct) point value $f(\omega)$.

The user's goal is to climb the "rankings", the boundaries of which are determined by a "percentage of possible points in a puzzle". The rankings, and the respective percent of total points necessary to achieve said rankings, are in Table 1, below. In the game, these percentages are rounded, and we will use the same metric moving forward.

Rank	% Necessary
Beginner	0
Good Start	2
Moving UP	5
Good	8
Solid	15
Nice	25
Great	40
Amazing	50
Genius	70
Queen Bee	100

Table 1: Rankings and Percentage of Total Points Necessary to Achieve Them

It's important to note the user does not have direct access to the amount of "possible points in a puzzle", but they *do* have access to the points necessary to reach each rank. Additionally, the "Queen Bee" rank is hidden from the user, and requires entering every word in the solution set.

The most obvious win condition, and the one used by most users, is reaching the "Genius" rank, requiring the user to get approximately 70% of the total possible points. With that being said, we'll define the user having "won" the game as when their set of input words $\Omega = \{\omega_1, ..., \omega_n\}$ has the following property:

$$\sum_{i=1}^{n} f(\omega_i) \geq 0.7 \cdot K \tag{3}$$

where K is the maximum number of points possible.

As an aside, I have *never* achieved the "Genius" rank independent of any help, and Professor Steven Strogatz of Cornell University jokingly told New York Times he gets to the genius level "right away", before letting his family mop up the easy words so he can gracefully come in last. Anecdotally, my experience has been that unlike Wordle, the Spelling Bee is a game where most users do not "win", under our win condition, but with Spelling Bee hidden behind a paywall, there's a distinct lack of data on that front.

With the game formalized, we now transition to the initial programming necessary to implement it.

2.2 Transferring the Game to Code

Unlike the New York Times, we're uninterested in designing a minimalist GUI for a user to play on, so the following programs will exclusively focus on implementing the game for different solving methods to use.

The Spelling Bee is difficult to "create" valid datasets of; see this MATLAB post for the complexity of algorithmically "creating" Spelling Bees. Additionally, with valid words words being described as "not obscure", there's still a bit of formalization missing. Is the Scrabble dictionary used? What makes a word "obscure" or not? While we will eventually create our own Spelling Bees in section 3.2, there are faults to doing so, and it's important to have an objective source of data directly from the website.

Luckily, the source code of the Spelling Bee contains two weeks' worth of Spelling Bees, including each day's center letter, outer letters, pangrams, and valid answers. So, with some quick scraping, we're able to create two `csv` files, `data.csv` and `solutions.csv`, where

$$\begin{aligned} \text{data}_{i,0} &= \text{day } i\text{'s central character} \\ \text{data}_{i,j} &= \text{day } i\text{'s } j\text{th outer character, } 1 \le j \le 6 \\ \text{solutions}_{i,j} &= \text{day } i\text{'s } j\text{th solution} \end{aligned}$$

To prevent algorithms from having benefits by sorting the solutions by length, we leave them in the psuedo-random order presented in the source code. This way, an algorithm that solves beginning with lower-length words is equally encumbered by searching `solutions.csv` as one that begins with higher-length words.

Now, we have 14 Spelling Bee data sets. This is, admittedly, a low number of data sets, but given this paper is meant to be an exploration rather than any formal writeup, we'll assume that's enough for testing.

All that's left to make the game playable by any search algorithm is to write the rules of the game. We do so in `game.py`, with the psuedocode:

```
def count_total_points(solution_list):
    possible_points = 0
    for i in solution_list:
        if length(i) == 4:
            possible_points += 1
        else:
            possible_points += length(i)
        if i is a pangram:
            possible_points += 7
    return possible_points

def get_rank_points(total_possible):
    rank_percentages = [0, 0.02, 0.05, 0.08, 0.15, 0.25, 0.40, 0.50, 0.70, 1]
    return [x * total_possible for x in rank_percentages]

*note precondition of word being valid*
def get_word_points(word):
    points = 0
    if length(word) == 4:
        return 1
    else:
        if word is a pangram:
            return 7 + length(word)
        return length(word)

def get_alg_solutions(alg, chars, soln, max_rank, *optional params):
    * Calculate amount of time it takes algorithm alg to complete
    each rank 0,...,max_rank independently, for characters chars*
    out: words_result, points_result, times_result
```

`get_alg_solutions` is intentionally left general for now, and will be specified in more concrete terms later. Important structural information is that:

$$\begin{aligned}\texttt{words\_result} &= [0, W_1, ..., W_{max-rank}]\\ \texttt{points\_result} &= [0, P_1, ..., P_{max-rank}]\\ \texttt{times\_result} &= [0, T_1, ..., T_{max-rank}]\end{aligned}$$

where W_i is the list of words algorithm `alg` used to reach rank i, P_i is the number of points algorithm `alg` achieved that "crossed" the boundary from rank $i-1$ to i, and T_i is the amount of time it took algorithm `alg` to reach rank i.

An important note is that the `get_alg_solutions` computes each rank independently. That is, $W_i \in W_{i+1}$ is not given, and $T_{i-1} = n$ and $T_{i+1} = m$ does not imply $T_i = m - n$. This was done for two reasons: With a lack of data sets, increasing randomness for each rank gives us a better picture, and with "Genius" rank not being the common human win condition, it allows us to examine algorithms under different win conditions. For instance, an algorithm A may have better results for lower win conditions than algorithm B, but better for higher win conditions, and allowing independence lets us examine each separately.

Now that our game rules are set up, we begin examining the information in `data.csv` and `solutions.csv` to see if we can gain information to use in our solving.

3 Data Analysis

An important step before writing algorithms is to do some analysis of the data. Below, we first examine the distribution of potential solutions' by length, and then by points earned. Analysis of these properties will help us write better algorithms, as well as give better results when analyzing the algorithms themselves.

3.1 Distribution of Potential Solution Word Length [INCOMPLETE]

Unlike typical algorithms, we have a clear upper bound on the length of possible solution words $\omega \in C$, that being $n < 20$. Intuitively, it seems like we can improve this bound. With all potential solution words having a maximum of 7 unique characters, it seems unlikely for many, or *any*, solution words to have a length of 19. Since we're interested in possible distributions of points as well, it's probably a good idea to examine specifically pangram lengths as well.

We'll first do a quick sanity check on our given solutions in `solutions.csv`, to get an idea if there could be any underlying distribution. In Figure 2, we see the solution words from the past two weeks of spelling bees appear to follow an exponential distribution! So, we'll continue with our analysis.

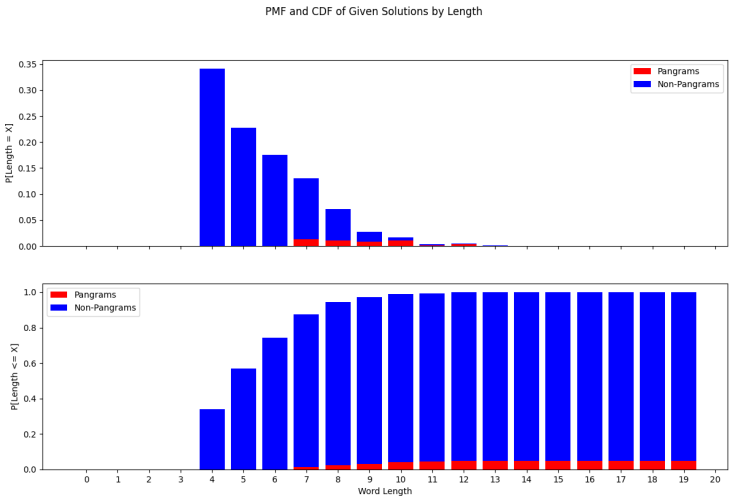

Figure 2: PDF and CDF of Given Solutions by Length

Word Length n	$\tilde{\mathbb{P}}[\text{len}(a) = n]$	$\tilde{\mathbb{P}}[\text{len}(a) \leq n]$	$\tilde{\mathbb{P}}[a \text{ is pangram} \mid \text{len}(a) = n]$
4	0.341256	0.341256	0.000000
5	0.227504	0.568761	0.000000
6	0.174873	0.743633	0.000000
7	0.130730	0.874363	0.103896
8	0.071307	0.945671	0.142857
9	0.027165	0.972835	0.312500
10	0.016978	0.989813	0.600000
11	0.003396	0.993209	0.500000
12	0.005093	0.998302	0.666667
13	0.001698	1.000000	0.000000
$\{14, ..., 19\}$	0.000000	1.000000	0.000000

Table 2: Figure 2 in table format

Using an English text dictionary that has been curated to contain over 100K words that *could* follow our limitations for *some* Spelling Bee and repeating the process, we see quite a different result, pictured in Figure 3:

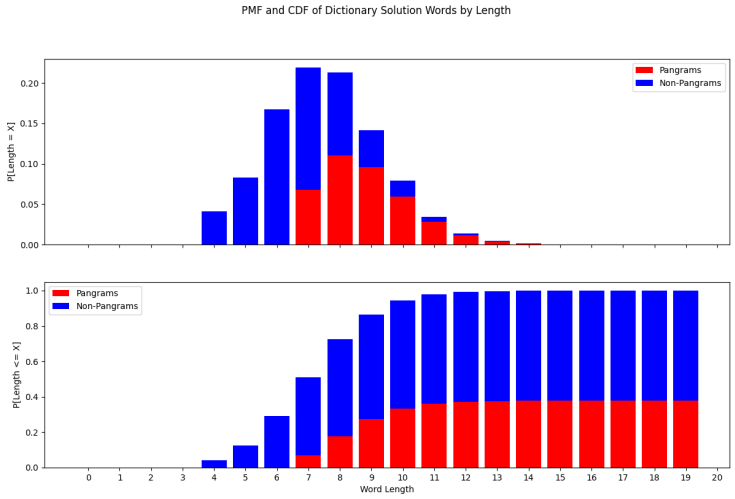

Figure 3: Dictionary Potential Solutions PDF and CDF

Word Length n	$\mathbb{P}[\text{len}(a) = n]$	$\mathbb{P}[\text{len}(a) \leq n]$	$\mathbb{P}[a \text{ is pangram} \mid \text{len}(a) = n]$
4	0.041383	0.041383	0.000000
5	0.083197	0.124580	0.000000
6	0.167370	0.291950	0.000000
7	0.219048	0.510998	0.309181
8	0.212973	0.723971	0.516349
9	0.141887	0.865858	0.675501
10	0.079358	0.945216	0.748677
11	0.034468	0.979684	0.822613
12	0.013635	0.993319	0.819484
13	0.004551	0.997871	0.884120
14	0.001543	0.999414	0.873418
15	0.000400	0.999814	0.878049
16	0.000127	0.999941	0.923077
17	0.000039	0.999980	0.750000
18	0.000010	0.999990	1.000000
19	0.000010	1.000000	1.000000

Table 3: Figure 3 in table form

What could lead to such severely different distributions? The most likely answer is the informal "obscurity" filter. In our dictionary, we have words like "zugzwang", which *is* a word, but isn't one an average human is likely to recognize. We'll attempt to find some level of "obscurity" that gives us more similarity.

Notice that the ratio of pangrams is significantly higher in our dictionary, and recognizing that any pangram defines 7 Spelling Bee games. Using this, and the assumption that Spelling Bee games are intended to be fun and knowing they ideally have around $30 - 45$ (from a Q & A session with the creator) solution words, we'll apply the following filter: Remove any words that are (1) Are pangrams with less than 2 or more than 3 vowels, or (2) Require such pangrams. These assumption holds for our given solutions, and eliminates a relatively minute small amount of our dataset.

3.2 Distribution of Potential Solution Points

From above, we know that the vast majority of potential solution words are of lower relative length. However, winning our game requires earning a certain *percentage of total possible points*. So, simply guessing all the small valid combinations of characters may not be the globally best solution possible. So, similarly to above, we begin by analyzing *just* `solutions.csv`, and potentially extending to `nltk`'s `wordnet`.

Examining the total potential points of each game in `solutions.csv` using our `count_total_points` function, we have a minimum of 89 potential points, and a maximum of 348 points. This difference is large enough to demand further analysis, as intuition suggested. The histogram of this data is seen in Figure 4, below:

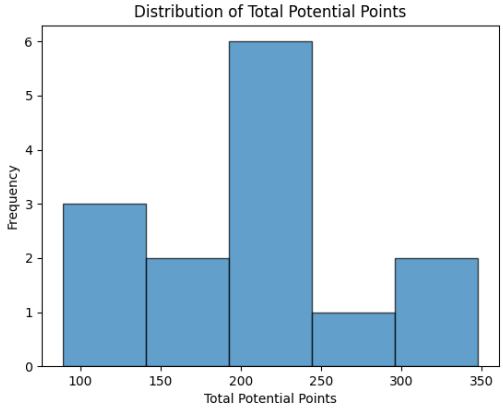

Figure 4: Histogram of Potential Points from `solutions.csv`

3.2.1 Attempting to Filter Potential Solutions

To get a more general histogram, we want to somehow create a large set of possible Spelling Bees. We'll begin unconcerned about the viability of these Spelling Bees, for now, and change our approach if difficulty arises. We first note that given a 7-tuple of characters $(c_0, ..., c_6)$, the only order that matters is c_0, i.e. $(c_0, c_1, .., c_6) = (c_0, c_6, ..., c_1)$, but neither are equivalent to $(c_1, c_0, ..., c_6)$. To find all unique combinations of the alphabet following this, we fix a first character c_0, and consider all subsets, size six, of the remaining 25 letters. This gives us

$$\begin{aligned} 26 \cdot \binom{25}{6} &= 26 \cdot \frac{25!}{6!(25-6)!} \\ &= 26 \cdot \frac{25 \cdot 24 \cdot 23 \cdot 22 \cdot 21 \cdot 20}{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2} \\ &= 26 \cdot 5 \cdot 23 \cdot 22 \cdot 7 \cdot 10 \\ &= 4,604,600 \end{aligned}$$

total combinations. With our end goal being a series of computations on *each* combination, this is an unreasonable amount. To reduce this to a more reasonable size, we'll make a few (trivially reasonable) assumptions about any given Spelling Bee, based on `data.csv`. First note the frequency graph of letters in `data.csv` in Figure 5, below.

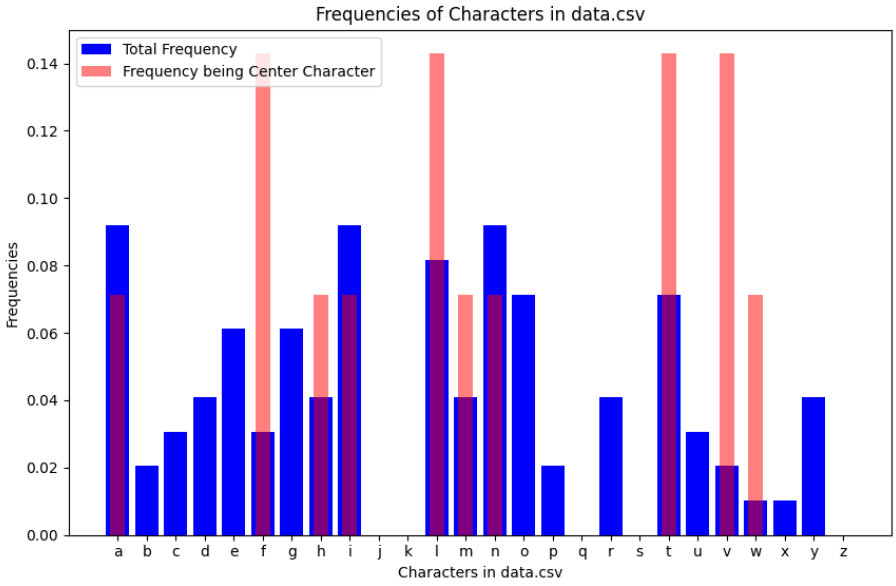

Figure 5: Frequency Map of `data.csv`, including frequency of being center character

1. There must be at least two vowels, not including y. This assumption holds for `data.csv`, and is in spirit of the Spelling Bee being a relatively easy game.
2. Similarly, there must be no more than three vowels, this time including y. This assumption also holds for `data.csv`
3. The overall frequency of any character c_i across multiple Spelling Bees follows some distribution sharing characteristics to the relative frequency of characters in the English Language.

Assumptions (1) and (2) are both obvious filters to apply, considering the nature of the Spelling Bee game- English words containing no vowels are primarily "obscure", with some examples being "crwth, cwm, and pfft". Similar logic holds for the number of vowels, and with `data.csv` following these conventions for all entries, these assumptions can be taken as relatively safe.

Assumption (3) is made by examining the frequency map of characters in `data.csv`. Assuming the distribution of characters were uniform and independent across days gives us the following probability of any one character not being in one puzzle:

$$\begin{aligned}\mathbb{P}[a_i \notin c] &= \frac{\binom{25}{6} + (25 \cdot \binom{24}{5})}{26 \cdot \binom{25}{6}} \\ &= \frac{7}{26}\end{aligned}$$

from there being $\binom{25}{6}$ combinations where a_i is first, and $25 \cdot \binom{24}{5}$ where a_i is not first. Using our assumption of independence across days, this gives the probability for any letter a_i to not be among 14 puzzles as

$$\begin{aligned}\mathbb{P}[a_i \notin c^{14}] &= (\mathbb{P}[a_i \notin c])^{14} \\ &= 1.0513 \times 10^{-8}\end{aligned}$$

an astronomically small number. In our sample `data.csv`, there are *five* letters not appearing once: [j, k, q, s, z] four of which are in the minimum five relative frequency letters [z, q, j, x, k]. The final missing letter, s, is actually deliberately missing- the editor of Spelling Bee, Sam Ezersky, said on a NYT Q&A in 2021 that he purposefully keeps s from being a character because "if every other word is a plural, it can make for tedious solving". Thus, our third assumption leads to eliminating s from our alphabet entirely.

This is a tremendous result! Not only does it reduce our entire *alphabet* moving forward by a letter, which will be important for our solving algorithms, but also reduces our set of "possible spelling bees" by a large amount. Going back to our discussion of the number of unique Spelling Bees, but now with s removed from the alphabet:

$$\begin{aligned} 25 \cdot \binom{24}{6} &= 25 \cdot \frac{24!}{6!(18!)} \\ &= 25 \cdot \frac{24 \cdot 23 \cdot 22 \cdot 21 \cdot 20 \cdot 19}{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2} \\ &= 25 \cdot 23 \cdot 22 \cdot 7 \cdot 2 \cdot 19 \\ &= 3,364,900 \end{aligned}$$

Next, we'll apply our assumptions to reduce this number further. A combination of seven characters contains less than 2 vowels if it contains exactly 1 vowel or exactly 0 vowels. A combination can contain exactly 1 vowel two ways: (1) the first character is a vowel, or (2) the first character is a consonant and there is one vowel in the rest. So, we can express the number of combinations containing exactly 1 vowel as $A + B$, where $A = 5 \cdot \binom{20}{6}$ is the number of combinations with one of five vowels at the beginning, and all consonants after, and $B = 20 \cdot \binom{19}{5} \cdot 5$ is the number of combinations beginning with a consonant, then 5 more consonants, then one of 5 vowels, since order of the final six characters doesn't matter. Next, the only way to express a combination containing zero vowels is to have $20 \cdot \binom{19}{6}$, with 20 possible consonants in first, and then the remaining 19 consonants comprising all remaining 6 characters.

With this in mind, the number of unique combinations following our rules (including removing s from the alphabet) that break our assumption (1) is:

$$\begin{aligned} \#(1) &= 5 \cdot \binom{20}{6} + 20 \cdot \binom{19}{5} \cdot 5 + 20 \cdot \binom{19}{6} \\ &= 1,899,240 \end{aligned}$$

Next, we follow a similar procedure for the upper bound on number of vowels, but this time we *do* consider y to be a vowel. With only 6 vowels, those being a, e, i, o, u, and y, we don't need to consider the case with 7 vowels, as that combination does not exist. Following the same logic as above, we find the number of unique combinations following our rules that break our assumption (2) is:

$$\begin{aligned} \#(2) = 6 \cdot \binom{19}{3} \cdot \binom{5}{3} + 19 \cdot \binom{18}{2} \cdot \binom{6}{4} + 6 \cdot \binom{19}{2} \cdot \binom{5}{4} + \\ 19 \cdot \binom{18}{1} \cdot \binom{6}{5} + 6 \cdot \binom{19}{1} \cdot \binom{5}{5} + 19 \cdot \binom{18}{0} \binom{6}{6} = 109,516 \end{aligned}$$

So, after removing s from the alphabet, and applying assumptions (1) and (2), we have a remaining

$$3,364,900 - 1,899,240 - 109,516 = 1,356,144$$

possible combinations. We've now reduced our original $4,604,600$ combinations by a factor of over 70%, but this number of combinations is still not quite manageable. It's possible that more filters could reduce this to a manageable size, but further assumptions will bring us further from objective data, so we label this sub(sub)section as a fun adventure in combinatorics and move on.

3.2.2 Creating Bees as Sam Ezersky Intended [INCOMPLETE]

3.3 Summary of Analysis and Consequences

In section 3.1, we examined the distribution of all potential solution words using `wordnet`. Surprisingly, we found that unlike our scraped data set, the larger set of potential solution words from `wordnet` follows an approximately normal distribution, with the positive skewness arising from potential solution words having a minimum length of 4.

The following result of is that there is a relative zero number of potential solution words of length greater than 17, and greater than 97% of potential solution words have length less than 11. Consequently, when building solving methods moving forward, prioritizing solving lower-length words first

is most likely optimal, as the relative abundance of lower-length words outweighs the higher score of longer-length words. This result is further extended later.

In 3.2, we examine examine points, rather than words. Despite the lack of direct results in 3.2.1, we learned that s is never a character in the hive, center or otherwise. This is a very helpful piece of information for building our solving methods later- eliminating s from any building method will help us find the most efficient algorithm possible.

In **??**, we used `nltk` and `wordnet` to generate our own Spelling Bees similarly to the creator of the game, using our assumptions from 3.2.1 to try and avoid "obscure" words. After eliminating data that was clearly outside the bounds of a normal Spelling Bee game, including removing s from any games, we found the distribution of total potential points to be approximately normal, centered around ~ 175. Next, we found the average *percent* of total points generated by *all* words of each length, which gave more evidence to our "lower (but not lowest) length words first" idea from 3.1. We found that, similar to our result from 3.1, not only are *all* solution words of length 4 for a given game, on average, worth relatively less than the same for lengths $5 \leq n \leq 8$, but a single solution word of length 4 has relative worth of, on average, only 20% of a single word of length 5.

As we move forward to building methods to solve a Spelling Bee instance, we'll focus on applying these results; namely:

1. Prioritize finding all words of length $5, 6, 7$, and potentially 8, then work back to words of length 4
2. Ignore the letter s if iterating through all possibilities (and plurals)
3. Effective solving algorithms must work well when the total potential points are in $[175, 200]$
4. If close to winning, use the average individual percent of potential points to determine the smallest word length that will most likely "win" the puzzle.

Now, we move on to two most obvious approaches for solving the puzzle.

4 Methods of Solving the Puzzle, and Problems Therein

In this section, we investigate using brute force and dictionary filtering to solve our problem. We'll see the benefits and drawbacks from each of these methods, which will help us in our final formulation of an optimal algorithm.

We'll restrict solving methods as follows: given the characters of the bee `chars`, the list of solutions `solns`, and a goal number of points to reach `gp`, formulate full words via any method possible using only `chars`, and check if they are solutions by using `solns`.

While blatant errors may arise from having access to `solns`, we prevent them by treating it exclusively as a "checker" of sorts, similar to how a human player may be unsure if a word is valid and try to submit it regardless. If the reader requires further justification for this, consider a function $g(\omega)$ such that

$$g(\omega) = \begin{cases} 1 & \text{if } \omega \in V \\ 0 & \text{otherwise} \end{cases}$$

We use `solns` exclusively to build such a function g from the set of all possible character combinations C to the solution set V, which a user of Spelling Bee clearly has access to.

We begin with the most obvious, naive method of solving our puzzle.

4.1 Brute Force Method

The brute force method is the most obvious algorithm to implement. Our first iteration will ignore the results found above (3.3), and simply cycle through all potential solution words, beginning with length 4. Since we are given `chars`, we'll exclusively cycle through all combinations of characters that follow our definition of "potential solution words", similar to a user of Spelling Bee. This will give us a good benchmark for our four quantifiers, which will be (1) Average Length of Word Found, (2) Average Number of Words Needed, (3) Average Points Earned Per Rank, and (4) Average Time Taken to Reach Rank.

4.1.1 Brute Force Results

Running the most naive solving method possible on `data.csv` and `solutions.csv` gives us the Figure 6, below. As expected, the average submitted word length is 4, and the time needed to complete each rank grows exponentially.

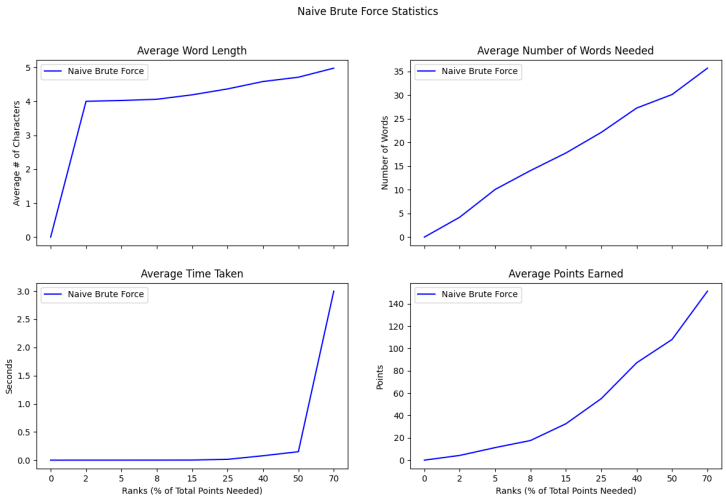

Figure 6: Unoptimized Brute Force Algorithm Results

Let's attempt some basic optimization of the brute force method. Our first optimization will be changing our word length priority to $5 > 6 > 7 > 8 > 4 >$ rest, as we found in 3.3 to be the supposed optimal priority. Examining the results depicted in Figure 7, we see somewhat expected results. The optimized BF has a strictly larger average word length due to prioritizing 4 later, which obviously gives a lower average number of words needed. With the time to find a word using brute force increasing by a factor of 7 for every increase in n, the Optimized BF takes slightly more time for high ranks than the Naive BF. Interestingly, the average points earned grew very similarly, implying both have similar margins when "crossing" a rank boundary.

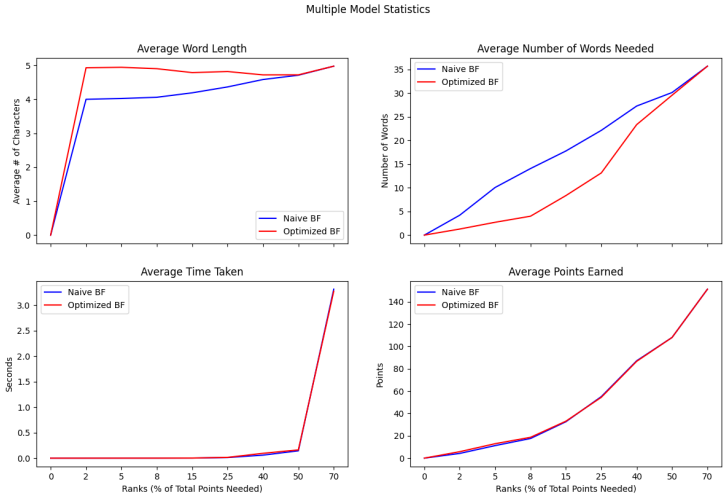

Figure 7: Comparing Naive Brute Force and Optimized Brute Force

Of course, different priorities may lead to different results, but we would begin to overfit to our small dataset. So, we'll stick to the priority list we found in 3.3, to avoid finding the best model for just our 14 Spelling Bees.

Regardless of optimization, any variation of brute forcing solutions to this puzzle are both uninteresting and inefficient, so we'll move on to basic analysis of the algorithm.

4.1.2 Analysis of Brute Force

This method is trivially correct and terminating, due to the nature of attempting every possible valid combination of characters $(c_0, ..., c_6)$.

With the method finding every word of length n before moving on to the next highest priority length, we examine the computational complexity of finding all solution words of length n. For any Spelling Bee game instance $(c_0, ..., c_6)$, we know all potential solution words will contain at least one c_0, and the rest of the letters may be any c_i, $0 \leq i \leq 6$. Using this, we find the number of potential solution words of length n to be:

$$1 \times \underbrace{7 \times ... \times 7}_{\text{n - 1}} = 7^{n-1}$$

Using the naive setup of "find all of length n, then restart and find all of length $n+1$" (rather than memoization or a lookup table), finding all potential solution words of length $< \alpha$ is thus $\sum_{j=4}^{\alpha-1} 7^{j-1}$.

Let $p = [p_0, ..., p_{14}]$ be our priority list, such that every word length $n \in [4, 20)$ is uniquely contained in p, and p_i is of strictly greater priority than p_{i+1} for any i. Then, fix $i \in [0, 14]$ such that by finding all words of length $p_0, ..., p_{i-1}$, we will reach 70% of potential points at some word of length p_i. Using this, we find the number of solutions requiring calculation, K, follows the inequalities below:

$$\sum_{\ell=0}^{i-1} 7^{p_\ell - 1} < K \leq \sum_{j=0}^{i} 7^{p_j - 1}$$

Using basic algebra, we find the interval that K lies in to be of size $7^{p_i - 1}$.

Thus, we can formulate a computational complexity for our optimized brute force algorithm. Let p be an arbitrary priority list, and define N as the index of p such that we will achieve 70% point coverage during some word of length $p[N]$. Then, using Big-Oh Notation, we'll find the Big-Oh number of iterations of the optimized brute force algorithm, `OBF`, as follows, with the eventual goal being a formulated computational complexity:

$$\begin{aligned}
\texttt{OBF} &\leq O(\sum_{j=0}^{N} 7^{p_j - 1}),\ p_j \leq 20 \\
&\leq O(\sum_{j=0}^{N} 7^{19}) \\
&= O(N(7^{19})) \\
&= O(N),\ N \leq 14 \\
&= O(1)
\end{aligned}$$

This result is interesting, although not unexpected. It effectively depicts the problem of using Big-Oh notation on algorithms with upper-bounded growth factors, which was one of the reasons I began this project to begin with. With the maximum number of calculations known to be $\sum_{i=3}^{18} 7^i$, there is no growth under Big-Oh notation, as the worst case is known. To find a more effective answer, we'll examine the expected runtime under unknown probability distribution $\mathbb{P}$.

Begin with the same setup as above, that being defining p and N the same. Then, our expected

runtime is

$$= O(\mathbb{E}[\sum_{j=0}^{N} 7^{p_j - 1}])$$
$$= O(\sum_{j=0}^{\mathbb{E}[N]} \mathbb{E}[7^{p_j - 1}])$$

Let p_k be the maximum priority s.t. $k \in [0, \mathbb{E}[N]]$

$$\leq O(\sum_{j=0}^{\mathbb{E}[N]} \mathbb{E}[7^{p_k - 1}])$$
$$= O(\mathbb{E}[N]\mathbb{E}[7^{p_k - 1}])$$
$$= O(\mathbb{E}[N]7^{\mathbb{E}[p_k] - 1})$$

With the overall distribution of the Spelling Bee game unknown, this is as far as we can simplify. Despite that, this formula is much more informative- we now see that the expected computational complexity is linearly dependent on how far through our priority list we have to go to achieve 70% coverage, as well as exponentially dependent on the maximum length priority up to and including that depth. Intuitively, we see the exponential behavior expected of a brute force algorithm such as this.

For completions' sake, we'll explicitly state here that we assume the following computations are constant time: individual combinations of the characters regardless of length, all verification of possible solutions, and all `python` functions regarding sets. As long as we stay consistent in our use of these throughout all methods, these are safe assumptions. Thus, our overall expected complexity of Optimized Brute Force is:

$$O(\mathbb{E}[N]7^{\mathbb{E}[p_k] - 1}) \tag{4}$$

For our second obvious method of solving, we'll focus less on analysis, as the creation and use of English dictionaries in `python` adds a high level of uncertainty. Instead, we'll focus on the concrete plots.

4.2 Dictionary Filtering

Our alternative obvious method of solving is a common method of finding specific words, and one I utilized in a class on Natural Language Processing, as well as above in **??**. Still a relatively simple algorithm, but can be extremely fast, depending on what precomputations we allow.

The obvious problem is that this does not technically provide a guaranteed win, since it's reliant on your chosen dictionary containing enough of the solution words to get $\geq 70\%$ of the total potential points. While this *technically* makes this algorithm incorrect, the chances of there being a statistically significant number of non-obscure English words missing from all available dictionaries is incredibly minor. For reference, the first link I saw when Googling "English Dictionary Text File" was a link to a public github repo containing a `words.txt` file with 497K unique "English" words.

4.2.1 Dictionary Filtering Results

We begin with basic dictionary filtering: taking every word in our corpus, we eliminate words that cannot be solutions to the given Spelling Bee, and submit every word that remains at the end. We'll standardize our five levels of precomputing as follows, with each level adding a new level of filtration done before running the algorithm:

(0) Do nothing prior to running the algorithm

(1) Precompute dictionary, filter nothing

(2) Filter all strings containing non-ascii characters

(3) Filter all strings containing characters not in the given Spelling Bee instance

(4) Filter all strings not containing the required character

Each level of precomputation after (1) filters the dictionary given to our algorithm further, until finally at precomputation level (4) the algorithm just has to check the length of words in the given dictionary. Obviously, for a genuine solving algorithm, the best precomputation possible would be restricted to level (2), but our goal is to examine the difference in performance for each of these levels.

With each level of precomputation completing the same steps, using the same methods, we expect all metrics but time to be approximately the same, excepting for possible randomness from using `sets`. Examining our results in Figure 8 below, we see those exact results. Obviously, taking more of the filtering steps off of the program decreases the time necessary to complete the filtration, giving us better running time for higher levels of precomputing, with the largest decrease being simply giving our algorithm the dictionary.

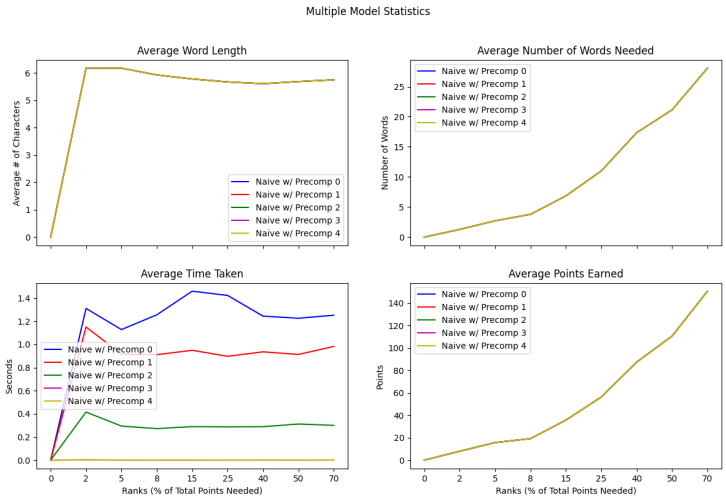

Figure 8: Naive Dictionary Filtering Results

Comparing our results to Figure 7 above, we see the benefits of dictionary filtering: the average time to solve our Spelling Bees is significantly less at higher ranks, even restricting ourself to the lowest level of precomputation. Along with this, the average word length is higher than the naive brute force method, due to the average word length of our dictionary being higher.

The primary drawback of the filtering method are the computational complexity dependent only on how filtered the dictionary is before running the algorithm. This gives the dictionary filtering method significantly higher runtime for lower ranks than the brute force methods above. The secondary drawback is what we mentioned above- we're dependent on an outside source providing a robust enough dictionary that we can solve any given Spelling Bee. While this may be true for our purposes, it's still an unsatisfying result.

The flaws of dictionary filtering are not due to lack of optimization, but rather the premise of the solving method itself. With that in mind, we will skip any attempt at optimization and move on.

4.3 Leaving the Spelling Bee Behind

Despite the length of analysis, solving the Spelling Bee itself isn't actually very difficult. Any Unix machine can use `/usr/share/dict/words` and filter words similarly to above, which gives an extremely low runtime with a reasonable amount of precomputing, and is as close to guaranteeing all of the solution words are in a dictionary as possible. The single line regular expression below can find all solutions in any dictionary:

```
^[c_1c_2c_3c_4c_5c_6]*c_0[c_1c_2c_3c_4c_5c_6]*$
```

and in the most extended case, you can both guarantee success and have a low expected runtime by using a massive dictionary textfile then cleaning up any missing points with a brute-force approach, focusing on lower-length words.

A quick Google of "NYT Spelling Bee Algorithms" gives a litany of approaches, but every one I saw was dependent on dictionaries and lists of English words. Despite multiple people going so far as to use bitvectors and lower-level programming languages to reduce runtime, every method I saw shared that one flaw. Looking to rectify that, we'll move to the next section with the hope of finding a polynomial-time algorithm that can solve the generalized Spelling Bee, with no reliance on dictionaries or outside resources.

5 Generalization of the Problem [INCOMPLETE]

The previous section was somewhat unsatisfying. Not only did the most time-efficient methods rely on using outside resources to supply dictionaries to filter from, but we're still unable to provably guarantee our algorithms terminate. In this section, we generalize the problem to intentionally eliminate distributions of any parameters, preventing reliance on dictionaries, and attempt to find a polynomial-time algorithm to solve the generalization.

At it's most general, our Spelling Bee puzzle can be described as follows:

Inputs: A an arbitrary set s.t. $|A| = 25$

$c_0 \in A$

$\{c_1, ..., c_6\} \subset A$

$V \subseteq \bigcup_{i=4}^{19} \{c_0, ..., c_6\}^i$ (restricted to only containment operation)

$$f : \bigcup_{i=4}^{19} \{c_0, ..., c_6\} \to \mathbb{Z}_+ \text{ defined as } f(\omega) = \begin{cases} 1 & \text{if } \text{len}(\omega) = 4 \\ \text{len}(\omega) + 7 & \text{if } \text{len}(\omega) > 4 \text{ and } \forall i \in [0, 6], c_i \in \omega \\ \text{len}(\omega) & \text{otherwise} \end{cases}$$

$k \in \mathbb{Z}_+ = \text{round}(0.7 \cdot \sum_{v \in V} f(v))$

$\exists \omega \in V$ s.t. $c_0, ..., c_6 \in \omega$

Output: $G \subseteq V$ s.t. $\sum_{\omega \in G} f(\omega) \geq k$ and $\forall \omega \in G,\ c_0 \in \omega$

At first glance, this problem looks nearly impossible to reduce to polynomial runtime. With V in principle being up to cardinality 7^{19}, we cannot afford to enumerate all of V, and without any idea of expected size of any $v \in V$, we cannot reduce the problem to a smaller size. We'll now attempt to determine if this problem actually *is* impossible to solve in polynomial time, and if so prove it.

Our first observation is that the scoring function $f(\omega)$ relies only on (1) the length of ω, and (2) whether or not ω contains all of $c_0, c_1, ..., c_6$. The insight we gain here is that, since we only care about reaching threshold k, and not finding a maximum value G, we don't need to attempt many *different* structural patterns, but rather a "small" family of candidate values.

First, observe that since $\text{len}(\omega) \in \{4, ..., 19\}$, we only have 23 possible values of $f(\omega)$. If ω has length 4, $f(\omega) = 1$, if ω has length ≥ 5, and does not contain all of $\{c_0, ..., c_6\}$, then $f(\omega) = \text{len}(\omega) \in \{5, ..., 19\}$, and if ω has length ≥ 7 and does contain all of $\{c_0, ..., c_6\}$, then $f(\omega) = \text{len}(\omega) + 7 \in \{14, ..., 26\}$. So, the possible values of $f(\omega)$ are $\{1\} \cup \{5, 6, ..., 19\} \cup \{14, 15, ..., 26\} = \{1, 5, 6, ..., 26\}$, which has 23 total possible values.

Expanding on this idea, we'll categorize any potential $\omega \in_? V$ as follows: ω is type $(i, 0)$ for each $i \in \{4, ..., 19\}$ if it does *not* contain all seven $c_0, ..., c_6$, and ω is type $(i, 1)$ for each $i \in \{7, 8, ..., 19\}$ if it *does* contain all seven $c_0, ..., c_6$. Finding all unique types is simple: $(4, 0), ..., (19, 0)$, $(7, 1), ..., (19, 1)$. In total, there are $16 + 13 = 29$ total types. Now, we note that some types can be paired by their $f(\omega)$ values, for example any ω of types $(14, 0)$ and $(7, 1)$ both give value 14. In total, there are 6 pairs of this type, which, if we redefine these matching pairs as being the same, gives us a nicely matching 23 total unique types.

Now, observe that the fact that we never need more than $\lceil \frac{k}{f_\tau} \rceil = \lceil \frac{0.7 \cdot \sum_{v \in V} f(v)}{f_\tau} \rceil$ of any type τ to achieve success, since

$$\lceil \frac{0.7 \sum_{v \in V} f(v)}{f_\tau} \rceil f_\tau \geq 0.7 \sum_{v \in V} f(v) = k$$