Graph Theoretic Approach to the Wordle Problem

George Lyu

March 2023

1 Problem Statement

Wordle is a single-player online game created by Josh Wardle. The objective is to guess a target five-letter English word. The player sequentially guesses a five-letter word and wins if they guess the target word. If they guess incorrectly, the game indicates which of the guessed word's letters are shared by the target word, and the player uses this information to improve their future guesses. The player loses if they cannot guess the target word within six attempts. [1]

One strategy is to use the first five attempts to guess five words that do not share any letters with each other (we will refer to such words with no shared letters as *letter-disjoint*). These five words collectively cover 25 of the 26 English letters. Then, the player can easily deduce the letters that make up the target word for their sixth guess. The five letter-disjoint guess words can be predetermined, so this strategy can be employed regardless of what the target word is. [2] Unfortunately, this strategy is not foolproof because the target word may have duplicate letters or anagrams (there exist other words made of the same letters in a different order), in which case we cannot always guess the target word on the last attempt even if we know its letter composition.

The goal of this project is to identify five letter-disjoint words that cover 25 letters.

2 Graph Representation

There are 14,855 valid words that are accepted as guesses. [3] 3,715 words have redundant anagrams, 2,751 words have duplicate letters (and thus contain fewer than 5 unique letters), and 2,739 words have both. Ignoring these words leaves 5,650 remaining words.

We can formulate the problem of finding five letter-disjoint words using graph theory. Consider the simple graph $G = (V, E)$, where the nodes V are the valid words, and the edges E join two words if and only if they are letter-disjoint. See Figure 1.

Finding five letter-disjoint words is equivalent to finding five nodes that are all adjacent to each other. In other words, we want to find a complete subgraph on five nodes, which we call a *5-clique*. We can use the following algorithm to find a 5-clique. The idea behind the algorithm is that we can recursively find a k-clique in G by finding some $v \in V(G)$ such that $G[N_G(v)]$, the subgraph induced by the neighbors of v, contains a $(k-1)$-clique. Then, this $(k-1)$-clique, together with the original node v, form a k-clique in G. [2]

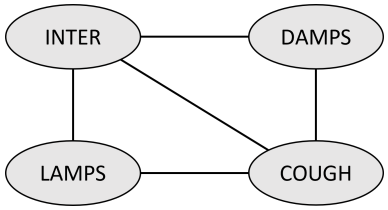

Figure 1: Example of the graph representation of the Wordle problem. The words [inter, cough, lamps, damps] are represented as nodes, where two words are connected if and only if they have no common letters. There are two 3-cliques: [inter, cough, lamps] and [inter, cough, damps].

Algorithm 1: FindClique

Input:
G, a nonempty simple graph
k, the size of the clique to find
Output:
Returns a set of k nodes of G that form a clique.
Returns *NULL* if no such k-clique exists.

```
1 if k = 1 then
2     return {v}, where v is any node in V(G)
3 for v ∈ V do
4     if N_G(v) ≠ ∅ then
5         C ← FindClique(G[N_G(v)], k − 1);
6         if C ≠ NULL then
7             return C ∪ {v} ;
8 return NULL ;
```

Our Python implementation is available at [4]. Calling **FindClique**$(G, 5)$ should yield a 5-clique, if one exists. However, applying this algorithm to the Wordle problem leads to a runtime exceeding 10 minutes. Rather than continuing to wait until the algorithm terminates, we can instead try some preprocessing tricks to reach a more reasonable runtime.

3 Presorting the Words

The for-loop in Step 3 of the algorithm examines the words in V in an arbitrary order. But if we sort V before running the algorithm, we can control the order in which Step 3 encounters the words in V. Then, we could first evaluate the words most likely to form k-cliques, hopefully leading to a faster runtime.

One possibility is to sort the words by the prevalence of their letters. Some letters, like "t", appear much more commonly than others, like "q". We can give each word w, de-

noted by its five letters as $w = l_1 l_2 l_3 l_4 l_5$, a *prevalence rating* defined by $r_w = \sum_{i=1}^{5} \mu(l_i)$, where $\mu(l_i)$ is the number of words in V that contain the letter l_i. Then, we order the words in V by their prevalence rating in ascending order. Words with lower prevalence contain rarer letters, so they are likelier to have the most neighbors in G from which they can form cliques.

A similar approach is to sort the words in V by their degree in descending order so that the nodes encountered first have the most neighbors.

4 Results and Discussion

We run the **FindClique** algorithm five times, once for each of the following word-presorting methods, to see if presorting affects the runtime. The "Alphabetic" presort is considered to be the experimental control. The algorithm is terminated at 300 seconds runtime regardless of progress. Results are shown in Table 1 and Figure 2. Our code is available at [4].

1. Sort V by prevalence, ascending.
2. Sort V by prevalence, descending.
3. Sort V by degree, ascending.
4. Sort V by degree, descending.
5. Sort V alphabetically (control).

Both the "Prevalence ascending" and "Degree descending" presorts successfully found a 5-clique within tenths of a second. Both discovered the same 5-clique: [glent, prick, waqfs, vozhd, jumby]. The other three presorts all failed to find a 5-clique within the 300s timeout. In fact, none of these three presorts even found a 4-clique within the timeout.

The "Prevalence ascending" presort performed best, finding the 5-clique about twice as fast as the "Degree descending" presort. However, degree is a more general metric than the prevalence rating and degree is easier to compute, so the "Degree descending" presort may be more useful for general max-clique problems.

Of the three presorts that did not find a 5-clique, the "Alphabetic" control presort performed best. This presort found a 3-clique before any of the other four presorts, and it found a 3-clique over 100 times faster than the "Prevalence descending" and "Degree ascending" presorts.

Therefore, prevalence and degree were both significant methods for presorting the words before calling **FindClique**. When the words were sorted by ascending prevalence or descending degree, the algorithm performed much faster than the control. When the words were sorted in the opposite way, the algorithm performed worse than the control.

5 Conclusion

We used a graph theoretic approach to find set of five words that covered 25 letters: [glent, prick, waqfs, vozhd, jumby].

We also found that it is useful to sort the order in which a graph's nodes are examined when looking for a clique. Evaluating the nodes in the order of ascending prevalence or descending degree may significantly reduce runtime.

Table 1: **FindClique** algorithm performance after different presorting methods. The "Alphabetic" presort is treated as the experimental control. The *Iterations* column counts the total number of calls to **FindClique**. The algorithm is terminated at 300 seconds runtime regardless of progress; this is denoted by "+" on the runtime and iterations.

Presort	Largest clique	Runtime (s)	Iterations
Prevalence asc.	5	0.0235	6
Prevalence desc.	3	300+	194,338+
Degree asc.	3	300+	194,269+
Degree desc.	5	0.0408	10
Alphabetic	3	300+	209,037+

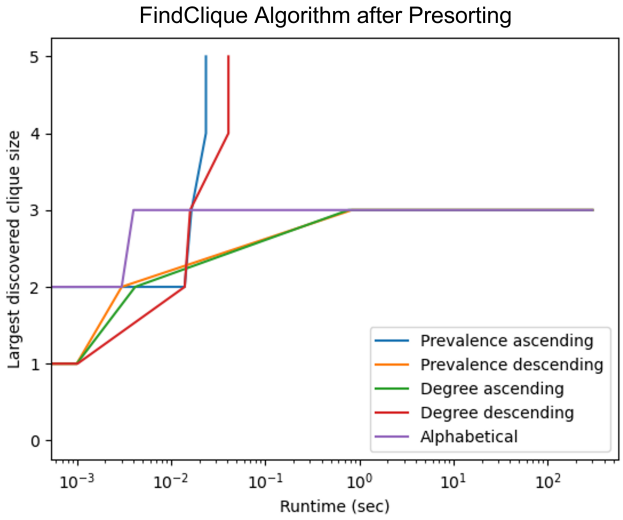

Figure 2: The size of the largest clique discovered by **FindClique** through the algorithm's runtime. The algorithm's performance is shown for several word-presorting methods.

References

[1] Josh Wardle & The New York Times. *Wordle*. [Online; accessed 14-March-2023]. 2023. URL: `https://www.nytimes.com/games/wordle/index.html`.

[2] Cornell University. *Developing a Fast Wordle Strategy Using Graph Theory*. [Online; accessed 12-March-2023]. 2022. URL: `https://blogs.cornell.edu/info2040/2022/09/29/developing-a-fast-wordle-strategy-using-graph-theory/`.

[3] tabatkins. *wordle-list*. [Online; accessed 12-March-2023]. 2022. URL: `https://github.com/tabatkins/wordle-list`.

[4] SeventhPrize. *WordleCover*. [Online; accessed 22-March-2023]. 2023. URL: `https://github.com/SeventhPrize/WordleCover`.