Poli Sci Gurus
John Morin, Poliana Martins de Santana, Alex Liu
Analysis of Donald Trump Rallies
GitHub Link: https://github.com/jmorin369/Project3.git
Video Link: https://youtu.be/zbCufHtOHmE

Extended and Refined Proposal

Problem: What problem are we trying to solve?

- The words of Donald Trump during his rallies inspire awe and admiration from all listeners. The uses of his vast vocabulary have yet to be truly explored to the fullest extent that they deserve. We attempted to remedy this issue by analyzing his varied vernacular and delving into his deft proficiency of the English language.

Motivation: Why is this a problem?

- As true analysts of Donald Trump's linguistic prowess, we endeavored to understand the deeper meaning behind his idioms. We hoped to look beyond the ocherous exterior and observe the enigmatic brain that powers it. Hopefully, you will gain a bit of wisdom by learning the true extent of which he uses popular words such as "China" and "Fake", in addition to how varied his vocabulary truly is.

Features Implemented: When do we know that we have solved the problem?

- We were able to accurately and quickly count how many times Donald Trump said each word in all of his speeches. In addition, we created a groovy graphic to display the contents in an easily digestible format as well as graphs that demonstrated the variety of Trump's words, the count of specific words, and time comparison of how long they took.

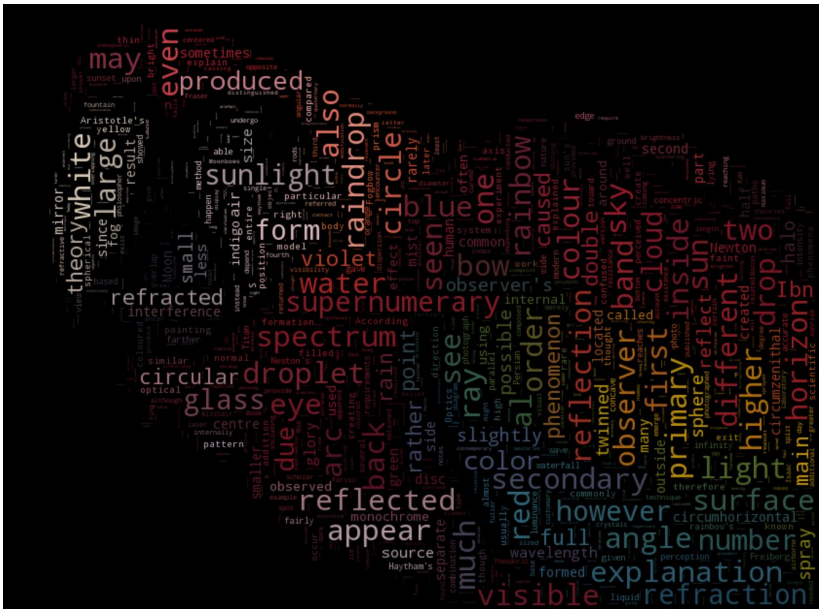
Description of Data: (Public data set we used and the link to the public data set)

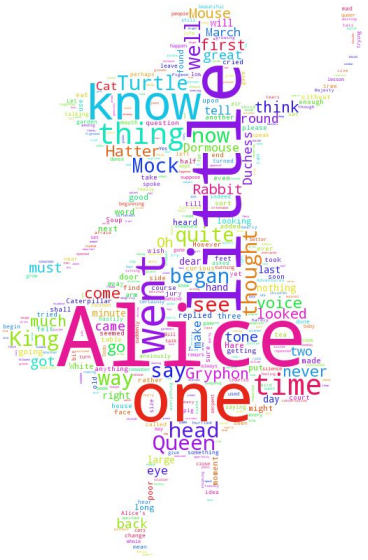
- The dataset we used contains transcripts from 35 of Donald Trump's campaign speeches given between 2019 and 2020. Each speech is given as a text file and is a full transcript. These speeches were given at rallies held in 19 different states.
- https://www.kaggle.com/christianlillelund/donald-trumps-rallies

Tools / Languages / APIs / Libraries: Programming languages or any tools/frameworks we used

- We used Python for our project
- Python gave access to a word cloud library that was created by Andreas Muller, and we used that as a visual representation of the frequency of words used. More information can be found through this link: https://python-graph-gallery.com/wordcloud/

- Word Cloud





Data Structures Implemented:

- Ordered Map (AVL Tree)
- Unordered Map (Hash table)

Data Structures Used:

- The Library used a dictionary (python version of unordered map) to parse and display the words
- We also used a dictionary to keep track of the count frequency to make graphing easier

Distribution of Responsibility and Roles:

- John: AVL Tree & Data Parsing / Filtering
- Poliana: UI, word cloud, data analysis and plotting.
- Alex: Hash Table implementation

Analysis

Changes and Rationale:

- The group chose not to implement any sorting algorithms as this was deemed unnecessary when these data structures already had their own methods to efficiently search words. Also, because the word cloud was implemented using a library instead of manually, we decided to add an additional plot that showed the frequency of certain word counts. This showed the variance in the words that donald trump used ie, the higher the number of 1's and 2's the more varied his speech and wider range of

vocabulary. In addition, we implemented a graph that shows multiple different word counts simultaneously and the searching performance of both structures.

Worst case time complexity of major functions:

Ordered Map:

Note: n is equal to the number of nodes in the structure

Search - O(log(n)): Only has to search through half the structure each time it goes down a layer, resulting in log2(n) time taken.

Add - O(log(n)): Has to search through the tree for where the node should be, so it has the complexity of the search time, plus a single rotation if necessary, resulting in O(log(n) + 1) which results in O(log(n)).

printTree - O(n): Needs to print through every node in the tree regardless of any optimization involved.

getCounts - O(n): Same with printTree. Needs to touch all of the nodes of the tree to make sure it updates count properly.

Unordered Map:

Note: n is the number of nodes stored at the index of the hash table (n will be less than 4, since n will only increase with each unwanted collision - an unwanted collision is one where two different words hash to the same key)

Insert - O(1) in average case. O(n) in worst case (where n < 4). The insert function is O(n) in the worst case since if there is a collision where different words hash to the same key in the hash table, then an additional node is stored at that index in the hash table. N is less than 4 because there is not expected to be more than 3 unwanted collisions.

getCount - O(1) in average case. O(n) in worst case (where n < 4). The getCount function is O(n) in the worst case since if there was a collision where two different words hashed to the same key in the hash table, then multiple nodes would exist at the hash table index. This means that the function must search through all of the nodes at that index.

Note: m is the number of non-null indices in the hash table

getCounts - O(m) in average case and O(m*n) in worst case (where n < 4). The getCounts function returns a map of all of the different counts and the number of times those counts exist in the hash table. To accomplish this, the getCounts function must iterate through each hash table index and each node at every hash table index. Hence, the time complexity is O(m*n).

User Interface / Parsing:

parse() - O(c(log(n1)+n2)): This function uses a nested for loop to cycle through every character (c) in the selected files. It uses two different for loops to insert into the ordered and unordered maps, where the ordered is O(log(n1)) and the unordered is worst case O(n2) (if it has a bunch of clashes), where n1 is the number of values in the data structure and n2 is the number of nodes at an index. The final result is the number of iterations / characters (c) times the worst case insertion time (n1 + n2).

build() - O(c(log(n1)+n2)+f): This function loops through f selected files O(f) and calls makeFile(), which also has O(f) complexity for f selected rallies. It then calls parse() (discussed above), increasing the complexity to O(c(n1+n2)+f).
printInfo() - O(w*(log(n1)+mn2))): This function calls parseSearch(), which has complexity O(w) for w words inserted into the search bar. Then, for each word retrieved, it calls each data structure's searchCount functions to get the count for that word. The AVL Tree search function increases the complexity by O(log(n1)) since the height of an AVL tree is always log(n1). Furthermore, while the Hash Table in the average case can retrieve the count in constant time, since our implementation used double hashing, the worst case would be O(n2), where n2 is the number of nodes stored at a particular index (n2 < 4).
freqPlot() - O(n1 + mn2): This function calls both of the *getCounts()* of each of the data structures, so the time complexity would be the addition of the two. The ordered map is O(n1) where n1 is every node and the hash table is O(mn2) where n2 is the number of nodes stored at a particular index and m is the total occupied indices.

Reflection

The overall experience as a group was positive even though there were definitely challenges along the way. As a whole, our goal was to analyze the words spoken by Donald Trump in his rallies. We accomplished that by implementing a simple search of the rallies and expanding on that to increase functionality and improve user experience. By dividing up the work individually, each member was able to focus on a specific section of the program. We were then able to combine these together and form the final product.

Many of the major challenges came from not understanding the scope of what we wanted to do. This was especially apparent when we thought we were on the same page, only to discover a week later that we had implemented different functions and needed to completely reorganize our code in order to be compatible with each other. This hampered our progress and caused more stress than if we had invested in communicating more. If we were to start over, we would definitely have set down a much clearer schedule from the start so that we would all be on the same page regarding the final product we wanted and the timeframe that we wanted it by.

Members Learned Experiences:

- John: I learned how to work with python in addition to working on a group using GitHub. I also became much more familiar with balancing trees as I had not done that efficiently and when I have thousands of nodes, the efficiency in adding becomes much more pronounced.
- Alex: Working on this project, I was able to learn a programming language I had never used before, python, as well as become more familiar with implementing and using hash tables. Both will be very helpful moving forward in classes, projects, and beyond.

- Poliana: I learned to code in python, which allowed me to become familiar with using multiple libraries and putting together a final product interconnecting all of them. I also learned how to use GitHub to work on a group project, which will certainly aid me in doing so in future classes, as well as my career. Lastly, it was fun to learn how to use tkinter to make a python GUI.

References

- AVL tree: Base structure was referenced from John Morin's Project 1 from the Data Structures & Algorithms course
- Word Cloud: https://github.com/amueller/word_cloud