COSC 320

*Analysis of Algorithms*

2022/2023 Winter Term 2

**Project Topic # 1**

**Keyword Replacement in a Corpus**

**Group Lead: Jeremy Adams**

**Group Members:**

**Jeremy Adams**

**Jolo Sesbreno**

**Yatharth Mathur**

**Project Proposal:**

**Problem Description:** In social media, abbreviated slang and acronyms come in a wide variety of ways. Not everyone has the time or access to learn and understand the meaning and context of these shortened versions; thus, this correcting algorithm can help turn modern slang into everyday English most people know and speak. For example, millions of Tweets on Twitter are sent out everyday containing slang both old and new, and remembering old ones while trying to learn the new ones can be difficult.To accomplish this, the algorithm being made will have to have an efficient way of storing many different abbreviations and the actual terms associated with them.

**Algorithm Examples:** An example of something in the real world that uses a similar algorithm is Google Search. Google is able to detect what you mean when you type an abbreviation into the search bar. It does this by having a very large amount of data on every abbreviation.

* Another example is Apple’s text replacement option for their IPhones. The user is able to input their own slang (or already established ones), then their device will translate it to the meaning also set by the user.

**Edge Cases:** A possible edge case would be if the tweets being analyzed have no abbreviations to replace.

* Another edge case would be if there are no tweets in the first place, i.e. the tweet array is empty.

**Expected Complexities:** The asymptotic worst case time complexity is O(n). This is because it has to increment through *n* amount of words. The replacement of any words should be linear and take O(1) time.

**Dataset Collection:** We plan to use Kaggle (<https://www.kaggle.com/>) to search for multiple potential datasets that can help run and test our algorithm.

**Programming Language:** For our choice of programming language, we’ve decided to use Python. This is mainly because it will probably be much easier to use Python in a group setting as it’s a lot more forgiving and less prone to problems compared to Java. It also has Dictionaries (the Python equivalent of Hash Tables), which we predict will be very useful for this project.

**Timeline:** For the timeline, each milestone is expected to be taken and be completed by 2.5 weeks.

**Task Separation:** Each member of the group will be given a chance to do all-around work and spread equal responsibility in each task. No one member will specialize in order to get a full feel of the project in the design and analysis aspect.

**Project Milestone 1:**

**Problem Formulation:** Given an input Array “words” of size n (n being the number of all words), and a dictionary of size m (m being the number of abbreviations/full phrase pairs), output the list of words with the abbreviations replaced into their full English counterpart.

**Pseudo Code:**

Load data from datasets

* Read tweet data into an array of strings for each word: “words”
* Read abbreviation data into a dictionary: “abb”. The keys should be the abbreviation and the values should be the real phrase associated with it.
* Create Array “output”
* For w in words:

If w in abb.keys() then:

add abb[w] to output

else:

add w to output

* return output

**Algorithm Analysis & Time Complexity:**

* Creating the array and dictionary will happen before the algorithm is run so the time complexity of that doesn’t matter.
* let size of “words” array be n.
* incrementing through the “words” array takes O(n).
* Checking if w is in abb.keys takes O(1).
* Adding to the output array takes O(1).

So overall, the time complexity of the algorithm is O(n).

**Proof of Correctness:**

* Loop Invariant: The size of the output array is always going to be equal to the iteration of the loop j.
* Initialization: Before the first iteration, j = 0., and the size of the output array is 0, so the loop invariant holds.
* Maintenance: For each iteration of the loop, the size of output, j, will be incremented by 1, and it will have added either the replaced word, or the original word. j is also incremented by 1, so the loop invariant holds.
* Termination: Once the loop has ended, j = n, and the output array will be of size n, so the loop invariant holds.

The loop invariant holds for each phase of the algorithm, so this proves that the algorithm is correct.

**Unexpected Cases/Difficulties:**

* Cases of two abbreviations having different words/meanings could confuse the algorithm. Solving it would require the context around the abbreviation.
* Abbreviations that do not have a direct translation could be misleading, but since our abbreviation dataset only contains ones with an English translation, then they will just be ignored.
* With high amounts of data in the library/dictionary any language used will use high amounts of time with results coming in.

**Project Milestone 2:**

**Dataset:**

* Along with the provided text dataset to test our algorithm, we used an external Kaggle dataset to fill our dictionary with slang and abbreviated words. ([Chat / Internet Slang | Abbreviations | Acronyms | Kaggle](https://www.kaggle.com/datasets/gowrishankarp/chat-slang-abbreviations-acronyms?resource=download))

**Implementation:**

* Our implementation is in a .ipynb file in our GitHub ([linked here](https://github.com/joloses/COSC320-Project)). Our pseudo-code is very similar to how we implemented it in addition to some neat python functions such as split() and zip(). We loaded our slang dataset and assigned it into a dictionary with keys and values. We then loaded the provided test run and took the review/tweet column, added it into a list and split each word between each space before adding it back into the array. Then simply run the two against each other.

**Results:**

* The plot of our first algorithm is what we expected as the complexity of the algorithm is linear, which matches up with what is shown. The implementation of the algorithm slightly affects the results as if there were more/less instructions in the code it would be slower or faster respectively by a constant amount. The choice of data structure affected the result because if we had chosen a slower data structure, the curve could have been logarithmic or exponential instead of linear.
* Chart, line chart

  Description automatically generated

**Unexpected Cases/Difficulties:**

* Coding using Matplotlib started off confusing, as we sifted through documentation to fit our needs.
* An unexpected case occurred when our data set confused the word ‘it’ with the abbreviation of ‘information technology’. As of now, there is no fix in the code but we will implement it to simply ignore that specific bit.

**Project Milestone 3:**

**Data Structure Rationale:**

We opted for a straight-forward/brute-force algorithm as a way to compare running time with our first algorithm. We chose this as we believe that our first dictionary/hashtable was an efficient algorithm and can not be improved upon, thus choosing a naive solution could help it stand out.

**Problem Formulation:**

Given an input Array “words” of size n (n being the number of all words), and two lists ‘phr’ and ‘abb’, (phrase and abbreviation) both of size m, output the list of words with the abbreviations replaced into their full English counterpart.

**Pseudo Code:**

* Read tweet data into an array of strings for each word: “words”
* Read the abbreviations column of data into a list ‘abb’
* Read the phrases associated with abbreviations into a list phr.    (phr and abb will have the same size and matching indexes)
* Create new array ‘output’
* for w in words:

for i in range(0, len(abb) - 1):

if (abb[i] == w):

add phr[i] to output

else: add w to output

    -  return output

**Algorithm Analysis & Time Complexity:**

* ‘phr’ and ‘abb’ lists are of size n (‘phr’ and ‘abb’ lists will be created before runtime).

* List time complexities:
  + Append: O(1)
  + Get Item: O(1)
  + Iterate: O(n)
  + Get index: O(n)

* Overall, the time complexity would be O(nm) where n accounts for the phrases and abbreviation lists and m accounts for the total words translated.

**Proof of Correctness:**

**-**     Loop Invariant: The size of the output array is always going to be equal to the iteration of the outside loop ‘j’. (the loop that increments through the words array)

-        Initialization: Before the first iteration, j = 0., and the size of the output array is 0, so the loop invariant holds.

-        Maintenance: For each iteration of the loop, the size of output will be incremented by 1, as it will have added either the replaced word, or the original word. Naturally, j is also incremented by 1 every loop so it can go to the next word, so the loop invariant holds.

-        Termination: Once the loop has ended, j = m, (m being the size of the ‘words’ array) and the output array will also be of size m, so the loop invariant holds.

The loop invariant holds for each phase of the algorithm, so this proves that the algorithm is correct.

**Unexpected Cases/Difficulties:**

* Cases of two abbreviations having different words/meanings could confuse the algorithm. Solving it would require the context around the abbreviation.
* Abbreviations that do not have a direct translation could be misleading, but since our abbreviation dataset only contains ones with an English translation, then they will just be ignored.
* Depending on the placement of certain phrases/abbreviations, iterating and getting indexes will result in long running times.

**Project Milestone 4:**

**Dataset:**

**-**     Along with the provided text dataset to test our algorithm, we used an external Kaggle dataset to fill our lists with slang and abbreviated words. ([Chat / Internet Slang | Abbreviations | Acronyms | Kaggle](https://www.kaggle.com/datasets/gowrishankarp/chat-slang-abbreviations-acronyms?resource=download))

**Implementation:**

**-**  Our implementation is in a .ipynb file called ‘Algo2’ in our GitHub ([linked here](https://github.com/joloses/COSC320-Project)). Our Implementation is essentially similar to our pseudo-code. We loaded our slang dataset, divided the abbreviation column and the translation column into two lists ‘abb’ and ‘phr’ respectively. We then loaded the provided test run and took the review/tweet column, added it into a list and split each word between each space before adding it back into the array ‘words’. We then ran the ‘words’ array against the ‘abb’ array and for every match: grabbed the index of the matching ‘abb’, search for the index in ‘phr’, take the translation in ‘phr’, and replace the string at the current index of ‘words’ with ‘phr’.

**Results:**

* The plot of the algorithm reflects the inefficiency of a naive solution as the line is so high it makes a linear complexity look almost constant. With more words and abbreviations, the time gets slower which is the cause of such a high line in the plot. With this, it is a useful comparison to our initial algorithm as that appears more efficient knowing a solution such as this exists.
* ***Chart, line chart

  Description automatically generated***

**Unexpected Cases/Difficulties:**

-        An unexpected case occurred when our data set confused the word ‘it’ with the abbreviation of ‘information technology’. As of now, we have added a band-aid fix by removing the translation altogether since additional context is needed to find the appropriate response.

-    As the word count increased, our algorithm’s time complexity rapidly increased as well. This resulted in extremely long wait times. We believe this is an unexpectedly good thing as we see how much more efficient our initial algorithm is.

**Conclusion:**

**Summary:**

* We tackled our chosen topic of key word replacement with two solutions: a naive brute-force algorithm (algorithm 2), and a simpler and more efficient dictionary/hash-table algorithm (algorithm 1). In comparison, our first algorithm was significantly better than our second algorithm- as intended. We designed the first algorithm as a simple, and efficient way to solve our problem; while we designed our second algorithm as naive as possible to get a contrast between the two when it comes to comparing complexities. Our first algorithm ran on average O(2n + n) while our second algorithm ran way beyond exponential.

**Plots (note that Algo1 is in ms and Algo2 is in s):**

Chart, line chart

Description automatically generated

***Chart, line chart

Description automatically generated***

**Work Distribution:**

* Jeremy was responsible for a majority of the coding process and ensuring that the proof of correctness in the documentation was accurate to the algorithms he was conducting.
  + Jeremy also implemented the plots that are seen in the documentation.
* Jolo was responsible for most of the documentation, and running the Github repository, ensuring that all related files and datasets are uploaded and working.
  + Jolo also found and implemented the Kaggle dataset that is used in our algorithms.
* Yatharth was responsible for presenting the project through the creation of an informative video.