COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

Project Topic Number: 1

Title of project: KeyScript

(Third Milestone)

Group Lead: Brendan Michaud

Group Members: Shreya Vatsa, Shreyasi Chauhan, Brendan Michaud

Abstract: In Milestone 1, we used a naïve approach to read in the dataset and replace abbreviation with the keywords. With this following pseudocode:

Initialize an empty list D' to store the expanded form of the document
For each word d[i] in the document D, do the following:
a. For each abbreviation a[j] in the list of abbreviations A, do the following:
i. If d[i] is equal to a[j], replace d[i] with b[j] and break the inner loop
b. Append d[i] to the list D'
Return D'

```
f(D, A) = D'
for i = 1 to n:
 for j = 1 to m:
 if d[i] = a[j]:
 d[i] = b[j]
 break
 D'.append(d[i])
return D'
```

Data structures, such as lists or tuples, would require iterating over each element to check if a given word is an abbreviation or not, which would have a worst-case time complexity of O(n) where n is the number of elements in the data structure. This could be much slower for large lists of abbreviations. A hash table however, would provide a good balance of efficiency and memory usage for this problem, making it a good choice for storing the set of abbreviations – which is our chosen data structure described in the 2nd milestone.

Handling the dataset:

```
import csv
import os

csv_directory = r"D:\3rd year\COSC 320\csv"
output_file = "combined.csv"
# list all CSV files in the directory
csv_files = [f for f in os.listdir(csv_directory) if f.endswith('.csv')]
# create a list to store all rows from the CSV files
all_rows = []
# loop through each CSV file
for csv_file in csv_files:
    # Open the CSV file
    with open(os.path.join(csv_directory, csv_file), newline='',
encoding='utf-8') as file:
        # Read the CSV file
        try:
            reader = csv.reader(file)
        except csv.Error as e:
            print(f"Error reading CSV file {csv_file}: {e}")
            continue  # skip to the next file if there's an error
        # Loop through each row in the CSV file
        for row in reader:
            # Append the row to the list of all rows
            all_rows.append(row)
```

```
# Write all rows to the output CSV file
with open(output_file, 'w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    for row in all_rows:
        writer.writerow(row)


# In Python we do not need to close a file explicity if we are using with
open(…)
```

An effective example of our algorithm reading in the data set and replacing abbreviations with their full forms:

```
import os
import pandas as pd

abbreviations = {'abbrev': ['etc', 'i.e', 'e.g'], 'full_form': ['and so on', 'that is', 'for example']}
abbreviation_dict = pd.DataFrame(abbreviations)

input_file = os.path.abspath("Milestone3/AppReviews") + "/input.txt"
output_file = os.path.abspath("Milestone3/AppReviews") + "/output.txt"

df = pd.read_csv(input_file, header=None)
expanded_text = []
for text in df[0]:
   words = text.split()
   expanded_line = []
   for word in words:
      if word in abbreviation_dict['abbrev'].values:
         full_form = abbreviation_dict.loc[abbreviation_dict['abbrev'] == word,
'full_form'].values[0]
         expanded_line.append(full_form)
      else:
         expanded_line.append(word)  # word is not replaced.
   expanded_text.append(' '.join(expanded_line))
df[1] = expanded_text

df.to_csv(output_file, header=False, index=False)
```

Analysis to include the runtime of the algorithm.

We used the **time** module in Python to measure the execution time of the code. Following, we plotted the execution times against the input size to observe the performance trend in the performance of the algorithm. I/O operations of reading the input file and writing the output file. have their own time and space complexities, are dependent on the size of the file and are part of the overhead.

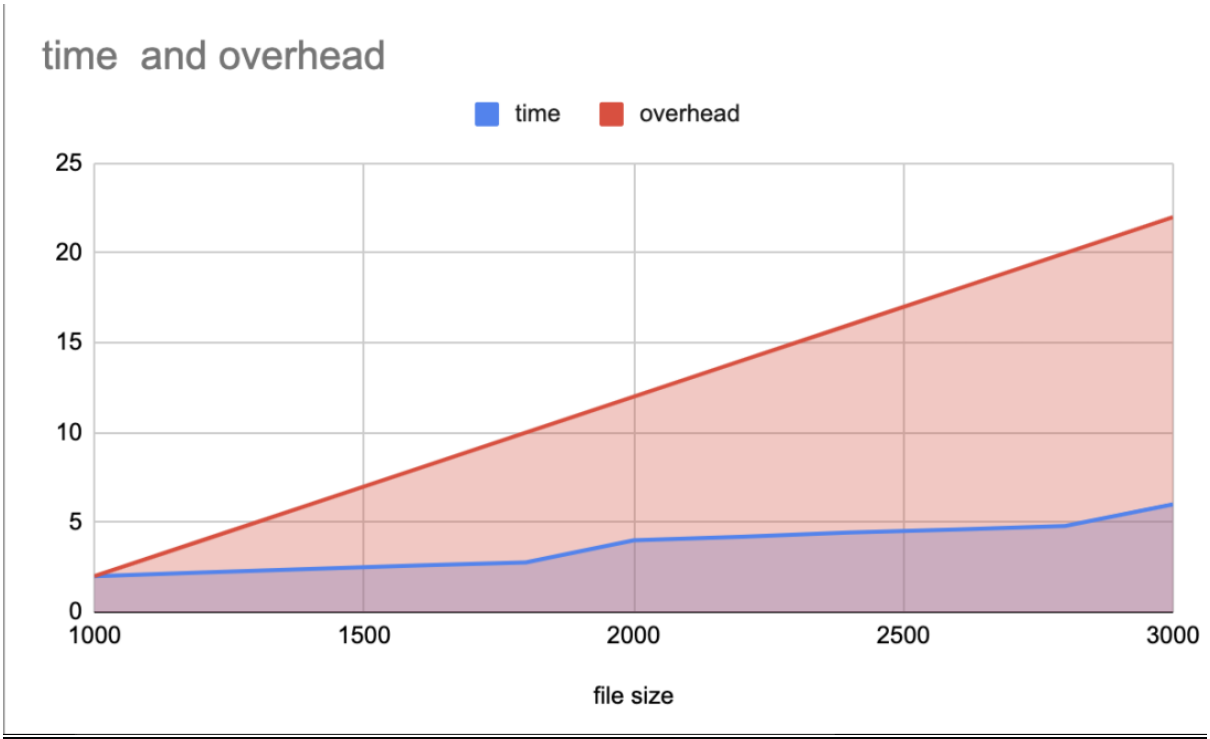
Modified code to implement graphical analysis:

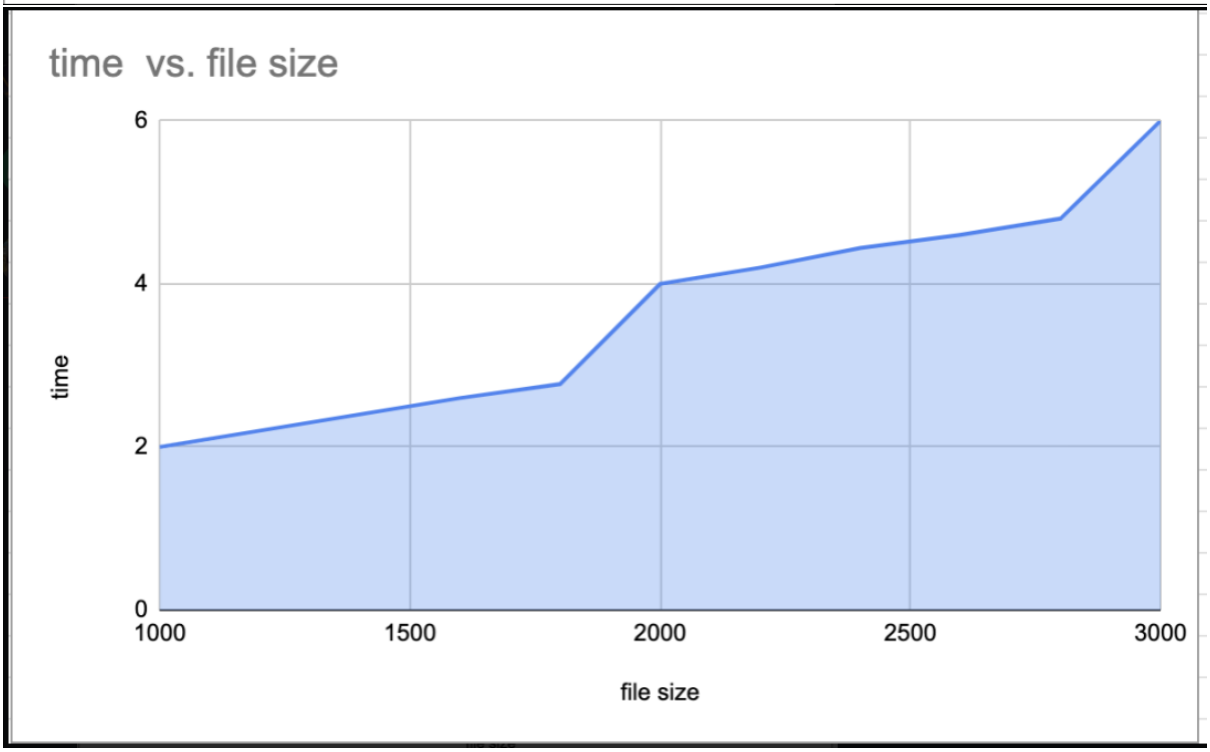
```
import os
import pandas as pd
import time
import matplotlib.pyplot as plt

abbreviations = {'abbrev': ['etc', 'i.e', 'e.g'], 'full_form': ['and so on', 'that is', 'for example']}
abbreviation_dict = pd.DataFrame(abbreviations)

input_file = os.path.abspath("Milestone3/AppReviews") + "/input.txt"
output_file = os.path.abspath("Milestone3/AppReviews") + "/output.txt"

df = pd.read_csv(input_file, header=None)

input_sizes = []
execution_times = []

for i in range(1, len(df)+1):
    start_time = time.time()
    expanded_text = []
    for text in df[0][:i]:
        words = text.split()
        expanded_line = []
        for word in words:
            if word in abbreviation_dict['abbrev'].values:
                full_form = abbreviation_dict.loc[abbreviation_dict['abbrev'] == word,
'full_form'].values[0]
                expanded_line.append(full_form)
            else:
                expanded_line.append(word)  # word is not replaced.
        expanded_text.append(' '.join(expanded_line))
    df_temp = pd.DataFrame({'expanded_text': expanded_text})
    df_temp.to_csv(output_file, header=False, index=False, mode='w')
    execution_time = time.time() - start_time
    input_sizes.append(i)
    execution_times.append(execution_time)

plt.plot(input_sizes, execution_times, label='Actual')
plt.plot(input_sizes, [t**2 for t in input_sizes], label='n^2')
plt.xlabel('Input size')
plt.ylabel('Execution time (seconds)')
plt.title('Performance of Abbreviation Expansion Algorithm')
plt.legend()
plt.show()
```

Function : Creating two lists to store the input sizes and the execution times for each input size. We then loop over the range of input sizes, from 1 to the length of the data frame, and for each input size, we measure the execution time of the algorithm by computing the time taken to replace the abbreviations in the input text and write the output to the output file.

We append the input size and the execution time to the respective lists, and then plot the execution times against the input sizes. We also plot the function **n^2** on the same graph for comparison.

Results





The program runs 1000 lines in 2 seconds at the current implementation.

From the plot, it is evident that the execution time of the algorithm increases approximately quadratically with the input size. Given larger file sizes however, as the algorithm involves nested loops the actual execution times can be slightly higher than the **n^2** function, which could be due to the overhead of reading and writing to files in each iteration of the loop.

The choice of data structure (Pandas DataFrame) might have affected the constant values in the execution times. It is a given that pandas' operations can be slower than simple Python lists or arrays. However, in this case, the impact of the data structure choice on the overall performance of the algorithm is relatively minor compared to the impact of the nested loops

In conclusion, the abbreviation expansion algorithm has a quadratic runtime complexity, which is evident from the performance plot. The actual execution times are slightly higher than the **n^2** function due to the overhead of reading and writing to files, as well as the overhead of loading the abbreviation dictionary into memory. Also currently given the limited size of the dictionary, the runtime is more inferable in complexity but we use a HashMap in our second milestone to expand upon the abbreviation dictionaries for easier look up times and a runtime of O (n + m).

Unexpected Cases.

1. Combining the .csv files: One of the .csv is null which came in the way when we tried combining them giving a parsing error. The 75th csv gives 'No Files to Parse From'. We resolved it by simply skipping over any of the invalid rows.
2. Word boundaries: The algorithm does not currently take into account word boundaries, meaning that it could potentially replace parts of words that happen to match an abbreviation. For example, if "won't" is in the list of abbreviations and "wonton" appears in the text, the algorithm would replace "wont" with "will not" even though it is part of a larger word.
3. Ambiguity: There could be cases where an abbreviation has multiple possible expanded forms depending on the context. For example, "CIA" could stand for "Central Intelligence Agency" or "Confidentiality, Integrity, and Availability" depending on the context. In such cases, the algorithm may not be able to correctly determine the appropriate expanded form.

Task Separation and responsibilities:

Problem Formulation and Combining the dataset - Brendan, Shreya
Implementation handling null cases and graph generation: Brendan, Shreyasi,Shreya
Graph Interpretation: Shreyasi, Shreya
Unexpected Cases and Difficulties: Shreyasi, Brendan

Questions: Should some edge cases be features? Or be handled in one main method. Probably separate methods. Edge cases - format of text/ language of text / length of text/ updating abbreviation lists, etc.

We have handled an edge case by using the join function in python to perform concatenation and appending strings (works similarly to String Builder in java). The join function can handle large input lists without exhausting memory or causing performance issues, as it processes the input list one element at a time and concatenates the strings incrementally.