Sentence Similarity

Andey Robins Tuttle

1. Execution Instructions

For more complete execution information, see the `README.md` file in the root of the submitted zip file. Use the following command to build and run the distance 0 (i.e. duplicate detection) for the file specified in the `-in` flag.

```
$ go run main.go -k 0 -in sentence_files/100.txt -size 100
File 'sentence_files/100.txt' has 2 duplicate lines.
Finished in 1.061792ms
```

For optimal performance, first build an executable with:

```
$ go build -o sentences main.go
$ ./sentences -k 0 -in sentence_files/100.txt -size 100
File 'sentence_files/100.txt' has 2 duplicate lines.
Finished in 711.042µs
```

All of the examples presented above were captured on an M2 Macbook Pro 2022 running macOS Ventura 13.0.1 with 16GB RAM under average, workday load. Execution information for the numbers presented below were captured on the same device, but with a minimal load where the only running applications in user space were iTerm2 and the Go runtime.

2. Hashtable Implementation and Hash Functions

As was presented in the assignment, a custom hashtable was built using separate chaining to provide an alternative to the `map` structure from Go. Separate chaining was used as opposed to linear probing since we anticipate meaningful numbers of collisions and don't want the lookup for a value to need to read the entire hashtable to test for similarity. Furthermore, a number of hash functions from Campbell et al. [1] were tested to find the ones which yielded the most efficient computation of similar sentences. The hashing function labeled "Hash 5" in their paper (and referred to as Campbell5 in this document) was the one they found to be most effective in identifying similar sentences, a result mirrored by the efficiency calculations of this work.

3. Execution and Results

To assist in timing, the program assumes that the parsing of command line arguments is not important to our results, and so begins a timer after the initial execution of the program parses any user supplied input and before opening any of the sentence files. Timing ends after the number of detected similarities or copies is output to the standard output stream. The total execution time is then also written to output. While early programs made use of the built in `time` utility, since this timing behavior was desired every

Data

File	Distance 0 Time	Distance 0 Count	Distance 1 Time	Distance 1 Count	-size
tiny.txt	81.042µs	2	723.167µs	0	100
small.txt	101.916µs	1	8.733083ms	0	100
100.txt	251.791µs	2	24.666458ms	0	100
1k.txt	2.074583ms	34	137.512375ms	0	1000
10k.txt	20.184375ms	488	1.405221042s	25	10_000
100k.txt	175.108875ms	7124	21.876438208s	1186	100_000
1M.txt	1.83640875s	79902	9m25.496216417s	38098	1_000_000
5M.txt	9.21365425s	516778	1h57m52.556996041s	353343	5_000_000
25M.txt	46.974499541s	4432935	12h53m51.515688709s	1976565	25_000_000

Figure 1: The execution time for the program with values of $k \in \{0, 1\}$. Also the number of identified identical and similar sentences.

time we executed the program, it was integrated into the program.

Execution made use of the `-exp` flag, which enumerates every file and a hashtable size to use for the input. The timing and identified counts for these executions are presented in Figure 1. More in depth analysis and human readable graphics are presented further.

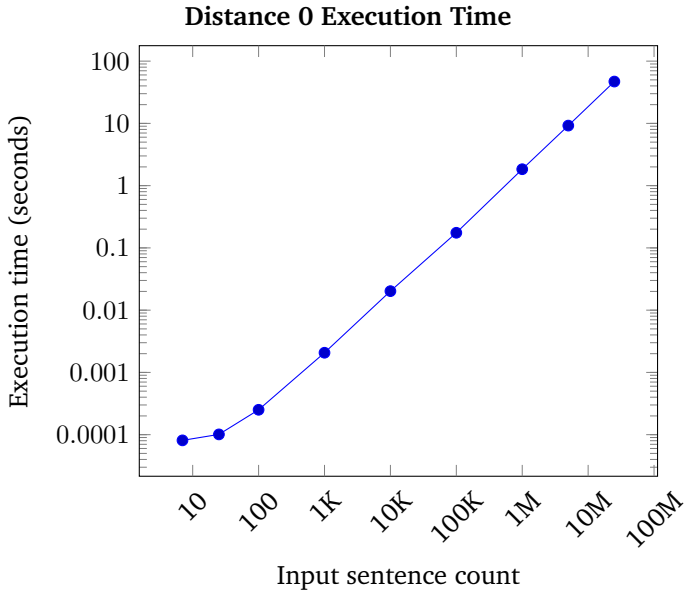

Figure 2: The runtime of the program for distance of $k = 0$. Both axes are log scale, demonstrating the linear growth rate of time as the input size grows while still showing the relationship between points at lower values. In other words, showcases the $O(n)$ runtime.

The distance 0 calculations were implemented in linear time trivially. For each line in the file, it was hashed with the SHA256 version of SHA-3 and stored into the hashtable. If the hash was already occupied, and the sentences matched, then a duplicate was found, counted, and discarded. Since a hashtable has a lookup time bounded by $O(1)$, and reading the file takes $O(n)$, we anticipate seeing an overall runtime of

$O(n)$. This is showcased in Figure 2. Both axes are On the low end, we observe a less than linear execution time for files with less than 100 lines, a product of the time requirements associated with starting and running the application itself. The response time of the computer itself become the issue when the entire programming is running in 81 micro seconds.

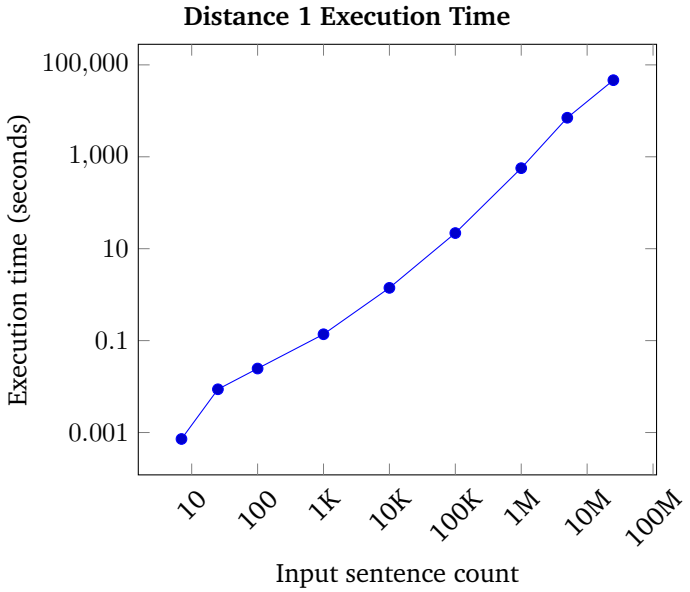

Figure 3: The runtime of the program for distance of $k = 1$. Both axes are log scale, demonstrating the linear growth rate of time as the input size grows while still showing the relationship between points at lower values. In other words, showcases the $O(n)$ runtime. Unlike the $k = 0$ case, slightly more variance as a result of the underlying processing is seen for this case. The R^2 value for these points is 0.9729.

The distance 1 calculations were implemented with far less triviality in linear time by finding and making use of the Campbell5 hash function [1]. In the process of loading each sentence into the hashtable, the frequency of each 4-gram in the sentence was calculated. This slowed the total execution time significantly as instead of using the highly optimized implementation of SHA-3, it made use of my implementation of the Campbell5 hash. This series of points are less obviously linear in their relationship, so further analysis calculated the correlation coefficient (0.9863) and R^2 value (0.9729), two analyses which support the assertion that the points are linearly related and thus the runtime of the program is $O(n)$.

References

[1] Douglas M Campbell, Wendy R Chen, and Randy D Smith. “Copy detection systems for digital documents”. In: *Proceedings IEEE Advances in Digital Libraries 2000*. IEEE. 2000, pp. 78–88.