Research on How Hash Functions Affect Programs

Jeong-Wook Kim
Artificial Intelligence Convergence Major, Sungkyunkwan University, Seoul, South Korea
jwjw0412@g.skku.edu

***Abstract*— In this research, we will present experimental results on how hash functions affect programs. Based on the hash part of the target program, Bigram Analyzer, we will place data in a hash table using various hash functions and find a way to distribute them as evenly as possible. Through this, we were able to confirm the results of how the use of an appropriate hash function affects program performance improvement. An appropriate hash function resulted in improved performance, and it was confirmed that the selection of an appropriate hash function is one of the important factors affecting program performance improvement.**

I. Introduction

Hashing is considered very important in today's computer science. Deciding how to do hashing, such as what data structure to use to form a hash table and what hashing function to use to store the data, is very important in most fields of computer science, regardless of field. Therefore, in this research, we will change the hashing method and see how hashing affects the program.

The target program to be used in this research is the well-known bigram analyzer for the n=2 case among n-grams. Bigram Analyzer is a program that takes a given file as input, breaks it into two-word pairs, and checks which word pairs occur frequently. This Bigram Analyzer provides the basis for understanding the structure of language through the analysis of word pairs, and can be used to predict and generate the next word, so it is used as one of the basic units used in language modeling in natural language processing today. This bigram analyzer has the advantage of being usable in a variety of fields, including document similarity analysis, keyword extraction and summary, and natural language processing model development.

This bigram analyzer consists of various detailed functions. Although there are many known program composition algorithms, we adopted the method of directly implementing the code by looking at the algorithm rather than using the code to modify the function and proceed with research. The algorithm for the program was available through a book.[1] In this Bigram Analyzer program, a hash table is used to store word pairs split into two words, and the location where they will be stored is determined through a hash function. In this study, we will aim to improve the performance of the program by modifying this part.

To evaluate the efficiency of the implemented hash function, the same input file should be given to each modified version of the program. In this research, we decided to provide the shakespeare.txt file, which contains Shakespeare's works converted into text files, as input. If the results are too short, it is difficult to analyze the results, so this 5.3MB file with over 120,000 lines was selected as it was expected to require sufficiently long processing time to analyze the file. The file exists in abundance on the Internet and was easily obtained.[2]

II. Design and Implementation

A. System Design

As explained in the introduction part, the data size is a 5.3MB text file with over 120,000 lines. This file consists of tens of thousands of characters, and its access pattern will only use read operations. The program will proceed by putting this file into a Bigram Analyzer, splitting the text into two-word pairs, storing them, measuring the frequency, and outputting the most frequent word pairs.

B. Implementation in Detail

When receiving an input file, the program reads line by line, splitting each line into words to create an array of words. Afterwards, these words are grouped into two-words and go through the process of creating word pairs.

Each pair of words is separated into two-word units and goes through the process of being expressed as a node. At this moment, the node contains information about two words, information about the frequency with which the word pair appears, and a pointer to point the next node. For this purpose, the node is declared through a structure in C language. The information on the word pair expressed in this way on a node basis is stored in a hash table through a hashing function, and the hash table consists of an array of node pointers with a bucket of HASH_SIZE. To prevent overflow during the storage process, hash tables basically use a chaining method in which the number of slots is variable.

After searching all nodes stored in the hash table and sorting the word pairs based on frequency, the top 7 word pairs with the highest frequency are output and the program ends.

III. Performance Evaluation

A. Experiment Setup

The environment settings are as follows. It was conducted on the Linux operating system using an Ubuntu-based Docker container, and program compilation was performed using gcc.

.

Text files are read line by line within the program, and the reading process ends when all are read.

B. *Analysis*

The size of the hash table is defined through HASH_SIZE defined through #define. The index of the hash table where the node will be entered is the value divided by HASH_SIZE after going through the hash function, and the hash function is defined as the sum of all ASCII code values of the two words. First, we gradually increased this value from a small number to a large number and compared the change in overall execution time.

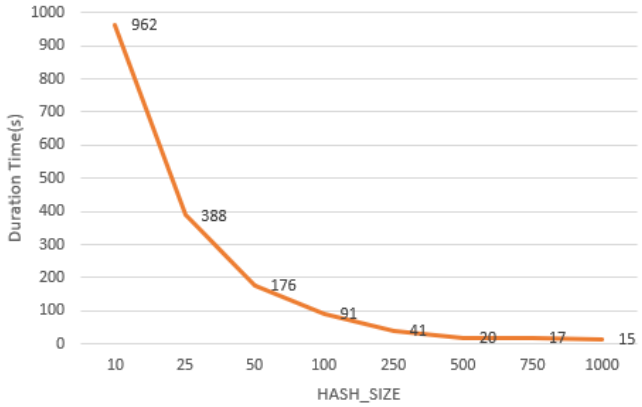

Fig. 1. Execution time graph according to hash table size.

As a result of changing the size of the hash table from a small value to a large value while keeping the hash function fixed, the execution time changed meaningfully. As the value of HASH_SIZE, the size of the hash table, increases, the execution time tends to gradually decrease. This decrease in time appears to be due to the fact that as the size of the hash table increases, the number of nodes assigned to one bucket decreases and the number of times searching and referencing addresses decreases.

And next, we changed the method of the hash function. Despite the increase in the number of hash tables, it was confirmed that there was an imbalance in nodes concentrating on specific buckets. Accordingly, it was decided to change the hash function to make uniform distribution. With the size of the hash table fixed, we decided to look at how the program execution time changes by changing the hash function.

For comparison, a total of four hash functions were created and compared. Hash function #1 is a function that adds up all the ASCII code values within two words, and hash function #2 is a function that calculates the average of the ASCII code values for each word and then adds the two values. Hash function #3 is a function that does not find all ASCII codes within two-words, but gives a value that adds only the ASCII code value of the first word of each word. Lastly, hash function #4 is a function that adds all the ASCII code values within two words and adds an additional number. Additional number is multiplication of a specific number and the length of the word (in here, the value of the number is set to 7).

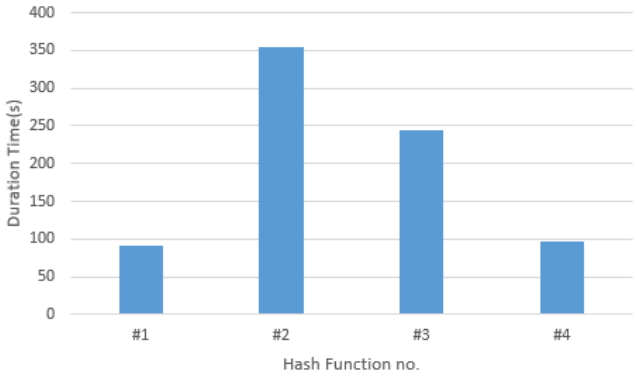

Fig. 2. Execution time graph according to hash function.

According to the resulting graph, it can be seen that the time has increased further compared to the #1 hash function. Rather, it was confirmed that the data imbalance had become more severe, and here it was confirmed that the existing hash function #1 provides the best data distribution.

IV. Conclusions

Through this research, we were able to determine how hash functions and hash tables affect program performance. It was found that the larger the hash table, the more the data is distributed, improving performance, and the use of an appropriate hash function that uniformly distributes the data leads to better program performance.

Improved performance was achieved by increasing the size of the hash table, but no hash function was found that showed better performance than the initial version. If we introduce a hash function that can produce a more uniform distribution, the performance of the program will be further improved.

Through this research, we found that the structure and arrangement of data are important factors that affect program performance. As we study computer science subjects in the future, we should not forget to always think about efficient data structures and improving performance.

Reference

[1] Randal E. Bryant, and David R, O'Hallaron, " Computer Systems: A Programmer's Perspective", 3rd Edition, PEARSON, 2016.

[2] MIT OpenCourseWare, “t8.shakespeare.txt”, January, 1994, Available: https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt