시스템 프로그램
Assignment-03

수 강 과 목	시스템프로그램 (SWE2001_42)
담 당 교 수	엄영익 교수님
학 과	전자전기공학부
학 번	2020313218
이 름	배민준
제 출 일	2023/11/28
과 제 명	Assignment-03

CONTENTS

Part 1. Program Specification & Overview

해당 과제의 Target Program으로 SP-03 교안에서 학습한 [Bigram Analyzer]를 선정하였다. 본 과제에서는 Linux 환경에서 제공하는 Profiling tool인 gprof를 활용하여 교재에서 제공한 초기 스펙의 Bigram Analyzer 프로그램을 최적화하는 실습을 수행한다.

• Target Program의 기능

Target program인 bigram analyzer는 n-gram 통계 프로그램의 일종이다. n-gram이란 텍스트 파일이나 음성 녹음 파일 등 언어 데이터를 포함하는 자료 내에서 n개의 인접한 단어 배열을 의미한다. n에 따라 gram에 접두어를 붙여 명명하는데 1개의 단어의 경우 uni-gram이라 하며, 2개의 단어 배열은 bi-gram으로 부른다.

직접 제작을 수행한 Bigram analyzer는 텍스트 문서에 대해 인접한 2개의 배열(Bigram)을 모두 추출하고, 각각의 bigram이 전체 텍스트 문서에서 몇 번 등장하는지 통계를 내는 프로그램이다.

코드는 C 언어 기반으로 작성되었으며, linked list를 사용하여 데이터를 구조화하고 정렬 및 검색 작업을 수행한다. 탐색 시간 관리를 위해 해당 프로그램은 hash table을 사용하여 bigram의 출현 빈도수를 관리한다.

해당 프로그램을 테스트하기 위한 텍스트 문서는 William Shakespeare의 『A Midsummer Night's Dream』을 선정하였다. 파일 내부에는 849,933개 가량의 단어가 존재하며, 4.32MB의 file size를 갖는다. 해당 텍스트 파일의 출처는 Python 기반 bigram 및 trigram 분석 프로그램의 github public repository에서 테스트용 text file을 업로드한 것을 가져왔다.[1)]

수업에서 활용하는 교재(교안)에서는 Bigram Analyzer의 초기 스펙에 대해 명시하고 있는데, 해당 사항은 다음과 같다.

• Bigram Analyzer의 초기 스펙

다음은 교재 『Computer Systems A Programmer's Perspective 3rd Edition』 기준 p.601에 제시된 Bigram analyzer의 초기 스펙이다.

---

1) 테스트 text file의 출처
『bigram-trigram-python』. (2018, May 21). Github.
https://github.com/Adrianogba/bigram-trigram-python/tree/master

① 각 단어는 파일로부터 읽어들이며, 소문자로 변환하여 처리한다. 소문자 변환에는 다음의 lower1 함수를 사용한다.

```c
void lower1(char *s) {
    long i;

    for (i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}
```

< Fig 1 > 초기 lowercase convert function

② 문자열 hashing function은 bucket의 수를 s라고 가정하면, 문자열로부터 0 ~ s-1 사이의 숫자를 생성한다. hash function의 초기 버전은 단순하게 문자열의 ASCII code를 모두 더한 값에 modulo 연산을 수행하여 구한다.

③ 각 hash bucket은 linked list로 구성된다. 파일에서 추출한 특정 bigram에 대하여 기존 list에 존재하면 frequency를 증가시키고, 존재하지 않는다면 새로운 노드를 추가하는 형식으로 구현된다.

④ hash bucket에 table이 모두 형성되면, 모든 bigram element를 frequency를 기준으로 내림차순 정렬한다. 초기 버전엔 삽입 정렬을 적용한다.

• Bigram Analyzer의 초기 버전 (v1)

위의 초기 스펙을 기준으로 프로그램을 작성하여 version 1을 완성하였다. 프로그램의 주요 기능을 바탕으로 코드를 간략하게 정리하였다.

1. main()

```c
#define BUCKET 500

int main(int argv, char**argc) {
    clock_t start =clock();

    char *filename =NULL;
    if (argv >1) {
        filename =argc[1];
```

```
    } else {
        filename = "shakespeare.txt";
     }

    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("[ERROR] File \"%s\" is invalid.\n", filename);
     }

    // Make Bucket
    List *bucket[BUCKET] = { 0 };
    for (int i = 0; i < BUCKET; i++) {
        bucket[i] = (List *)malloc(sizeof(List));
        list_init(bucket[i]);
    }

    printf("\n-----------------------------\n");
    printf("[INFO] Mining Bigrams...\n");
    printf("-----------------------------\n");

    // Extract bigram from the text file and renew the bucket
    parse_bigram(file, bucket);

    // Merge all list that seperated to hash map(bucket)
    List *merged_list = bucket[0];
    for(int i = 1; i < BUCKET; i++) {
        merged_list =merge_list(merged_list, bucket[i]);
    }

    // Do insertion sort & print info
    sort_list(merged_list);
    print_list(merged_list, 1);

    // Clear all resources
    fclose(file);
    clear_list(merged_list);

    // Program running time check
    clock_t end = clock();
    printf("\n----------------------\n");
    printf("Running time: %.3lfs\n", (double)(end-start) /CLOCKS_PER_SEC);
    printf("----------------------\n");
    return 0;
}
```

위의 코드는 Bigram_v1.c의 main 함수의 코드이다. command line argument로 원하는 텍스

트 파일명을 제공하여, 다양한 문서에 대해 실험을 할 수 있다. 기본적으로 셰익스피어의 소설을 기반으로 테스트를 진행하였기 때문에, default로는 "shakespeare.txt"를 filename의 인자로 넘겨준다.

text file을 open한 뒤에는 Linked list 자료구조에 대한 포인터를 element로 하는 배열을 선언한다. 해당 구조가 교재에서 bucket이라고 말하는 hash map을 구현한 것이다.

```c
typedef struct __BigramData {
    char *fword;
    char *bword;
    int   freq;
} Bigram;

typedef struct _node {
    Bigram *bigram;
    struct _node *next;
    struct _node *before;
} Node;

typedef struct _likedList {
    Node *head;
    Node *cur;
    Node *tail;
    int num_of_data;
} List;
```

< Fig 2 > Linked List의 구조

```c
void list_init(List *plist) {
    plist->head = (Node*)malloc(sizeof(Node));
    plist->tail = (Node*)malloc(sizeof(Node));

    plist->head->bigram = NULL;
    plist->tail->bigram = NULL;

    plist->head->next = plist->tail;
    plist->tail->next = NULL;

    plist->head->before = NULL;
    plist->tail->before = plist->head;

    plist->cur = plist->head;

    plist->num_of_data = 0;
}
```

< Fig 3 > Linked List 초기화 함수

Linked list는 향후 정렬을 할 때, 각 bucket에 저장된 리스트를 병합하기에 편리하도록 Dummy head와 Dummy tail 노드를 삽입하여, dummy head의 뒤쪽에 새로운 노드를 추가하는 형태로 프로그래밍하였다.

main 함수에서 bucket에 대한 initialize를 수행하는 것 외에는 기능을 모두 함수로 정의했다.

```c
// Extract bigram from the text file and renew the bucket
parse_bigram(file, bucket);

// Merge all list that seperated to hash map(bucket)
List *merged_list = bucket[0];
for(int i = 1; i < BUCKET; i++) {
    merged_list = merge_list(merged_list, bucket[i]);
}

// Do insertion sort & print info
sort_list(merged_list);
print_list(merged_list, 1);
```

< Fig 4 > main 함수의 뒷부분

기본적으로 bigram의 추출과 bucket의 구성에 대한 모든 기능은 parse_bigram 함수에 모두 포함되어 있다. 그리고 list의 병합, 정렬, 출력에 대한 함수를 각각 정의하여, main에서는 merge_list(), sort_list(), print_list() 함수를 순서대로 호출한다.

따라서, 본 프로그램의 주 기능은 다음의 4가지 함수이고 각 함수에 대해서 설명할 것이다.

- parse_bigram()
- merge_list()
- sort_list()
- print_list()

2. parse_bigram()

```
void parse_bigram(FILE *file, List **bucket) {
    char fword[20] = { 0 };
    char bword[20] = { 0 } ;
    int hash_idx = 0;
    int dup_flag = 0;

    char buffer[3000] = {0};
    char *fp =NULL;

    while (fp = fgets(buffer, 3000, file)) {
        if (buffer[0] =='\n') {
            continue;
        }

        lower1(buffer);

        while (1) {
            extract_bigram(buffer, fword, bword);
            if (bword[0] == '\0') {
                break;
            }

            hash_idx =hash(fword, bword);
            dup_flag =search_bigram(bucket[hash_idx], fword, bword);
            if (dup_flag == -1) {
                Bigram *bigram =init_bigram(fword, bword);
                append_2_head(bucket[hash_idx], bigram);
            }
```

```
        }
    }
}
```

위 함수는 bigram을 추출하는 핵심적 기능을 한다. 수행하는 기능은 다음과 같다.

[기능 - ①] 파일에서 문자열 한 line을 읽어와 모든 대문자를 소문자로 치환

fgets()로 '\n'을 만날 때까지 한 line을 읽어 buffer에 저장한다. 그리고 Fig 1에서 언급한 lower1() 함수를 사용하여 문자열 내 모든 대문자를 소문자로 변경한다.

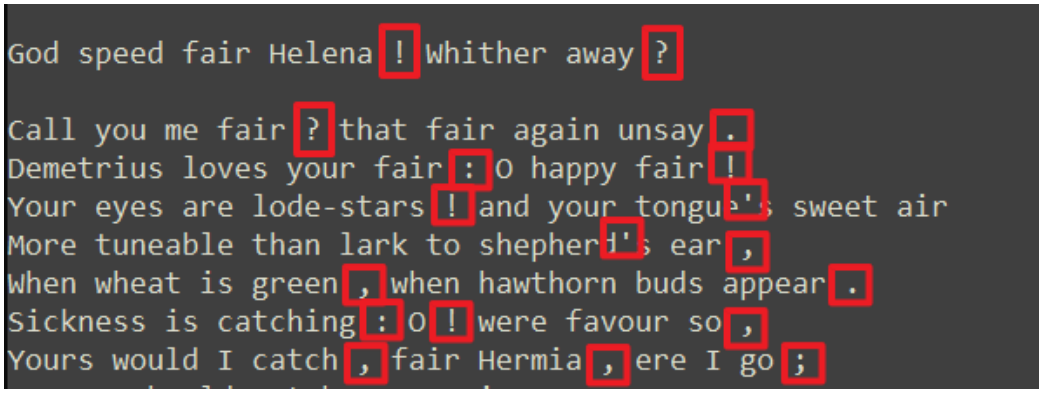
[기능 - ②] 문자열에서 bigram을 추출하여 front-word와 back-word를 추출

extract_bigram() 함수로 Bigram의 앞에 위치한 단어(front-word)와 뒤쪽에 위치한 단어(back-word)를 추출한다. extract_bigram() 함수는 내부적으로 static char *형 변수를 하나 선언해서 단어를 추출한 마지막 위치를 저장하고, strtok() 함수의 이용과 비슷하게 함수를 반복적으로 호출하여 연속적인 bigram 추출이 가능하도록 구현했다.

```c
void extract_bigram(char *buffer, char *fword, char *bword) {
    static char *g_address = NULL;
    int offset = 0;
    int n = 0;
    int __trash_n = 0;
    char __trash[20] = { 0 };

    if (g_address == NULL) {
        g_address = buffer;
    }

    if (fword[0] == '\0') {
        sscanf(g_address, "%[^\n\r,; !.:?_'\"-]%n", fword, &n);
        offset += n;

        n = 0;
        sscanf(g_address + offset, "%[\n\r,; !.:?_'\"-]%[^\n\r,; !.:?_'\"-]%n", __trash, bword, &n);
        offset += n;

    } else if (fword[0] != '\0' && bword[0] != '\0') {
        strcpy(fword, bword);
        sscanf(g_address, "%[\n\r,; !.:?_'\"-]%n", __trash, &__trash_n);
        offset += __trash_n;

        sscanf(g_address + offset, "%[^\n\r,; !.:?_'\"-]%n", bword, &n);

        if (__trash[__trash_n - 1] == '\n') {
            strcpy(bword, "");
            g_address = NULL;
            return;
        }
        offset += n;
    } else if (fword[0] != '\0' && bword[0] == '\0') {
        sscanf(g_address, "%[\n\r,; !.:?_'\"-]%n", __trash, &__trash_n);
        offset += __trash_n;

        sscanf(g_address + offset, "%[^\n\r,; !.:?_'\"-]%n", bword, &n);
        offset += n;
    }

    g_address = g_address + offset;
    return;
}
```

< Fig 5 > extract_bigram()

위의 Fig 5의 코드를 보면 해당 함수는 g_address라는 정적 변수를 기준으로 read 한 byte만큼 offset을 더해가며 문자열을 읽도록 구성하였다.



< Fig 6 > 『한여름 밤의 꿈』 소설 중 일부 발췌 - 특수문자 강조

한편, 소설 파일을 확인해 보면 짧은 문단 내에도 Fig 6처럼 기본적으로 많은 문장 부호들이 섞여 있다. 따라서 Fig 5의 초록색 박스처럼 정규식을 이용해 필터링을 수행하고, 특수문자가 아닌 알파벳만을 읽을 수 있도록 parsing을 수행했다.

[기능 - ③] front-word와 back-word를 기반으로 hashing하여 얻은 정수로 bucket indexing

추출한 두 단어를 가지고 ASCII 코드의 총합을 구한 뒤, 그 값에 modulo 연산으로 hashing을 수행한다. 해당 hashing 로직은 교재에서 제공한 초기 specification 중 하나이다.

```c
int hash(char *fword, char *bword) {
    int ascii_sum = 0;
    char ch = fword[0];

    for (int i = 1; ch != '\0'; i++) {
        ascii_sum += ch;
        ch = fword[i];
    }

    ch = bword[0];
    for (int i = 1; ch != '\0'; i++) {
        ascii_sum+= ch;
        ch = bword[i];
    }

    return (ascii_sum % BUCKET);
}
```

< Fig 7 > hash 함수

[기능 - ④] 기존 Linked list에 추출한 Bigram이 존재하는지 확인
[기능 - ⑤] 존재한다면 frequency를 1 증가, 존재하지 않는다면 새로운 노드 추가

위의 두 기능은 Fig 8의 search_bigram 함수가 담당한다.

```
int search_bigram(List *plist, char *fword, char *bword) {
    set_first(plist);
    int idx = 0;
    for (; idx < plist->num_of_data; idx++) {
        if (!strcmp(plist->cur->bigram->fword, fword) && !strcmp(plist->cur->bigram->bword, bword)) {
            plist->cur->bigram->freq++;
            break;
        }
        next_node(plist);
    }

    if (idx == plist->num_of_data) {
        idx = -1;
    }

    return idx;
}
```

< Fig 8 > search_bigram()

list에 추출한 bigram과 동일한 노드가 이미 존재하는지 순차적으로 검색하는 코드이며, 동일한 노드를 발견하면 해당 bigram의 frequency를 1만큼 증가시킨다. 만약 list 전부를 검색해도 동일한 bigram이 검색되지 않는다면, -1을 반환하여 Fig 9 코드의 dup_flag가 -1이 되도록 한다.

dup_flag가 -1이면 새로운 Bigram 구조체를 fword와 bword로 초기화하여 list의 head 쪽에 새롭게 노드를 추가한다.

```
dup_flag = search_bigram(bucket[hash_idx], fword, bword);
if (dup_flag == -1) {
    Bigram *bigram = init_bigram(fword, bword);
    append_2_head(bucket[hash_idx], bigram);
}
```

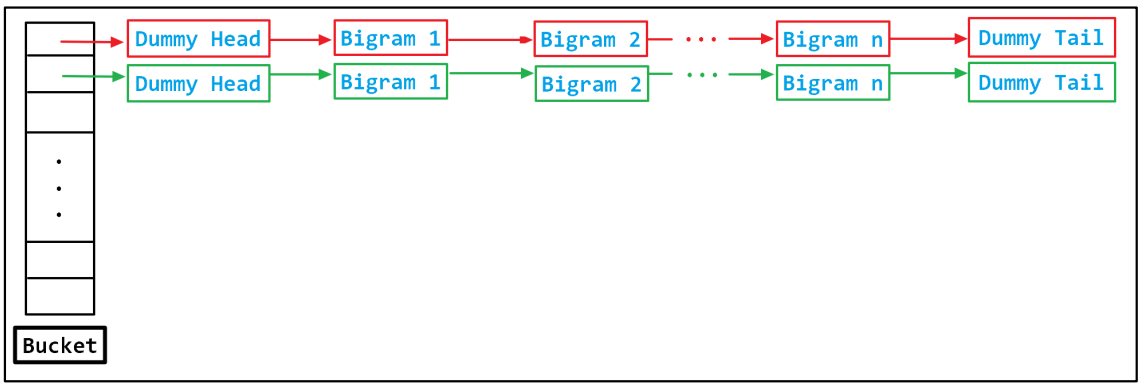
< Fig 9 > parse_bigram()의 뒷부분

3. merge_list()

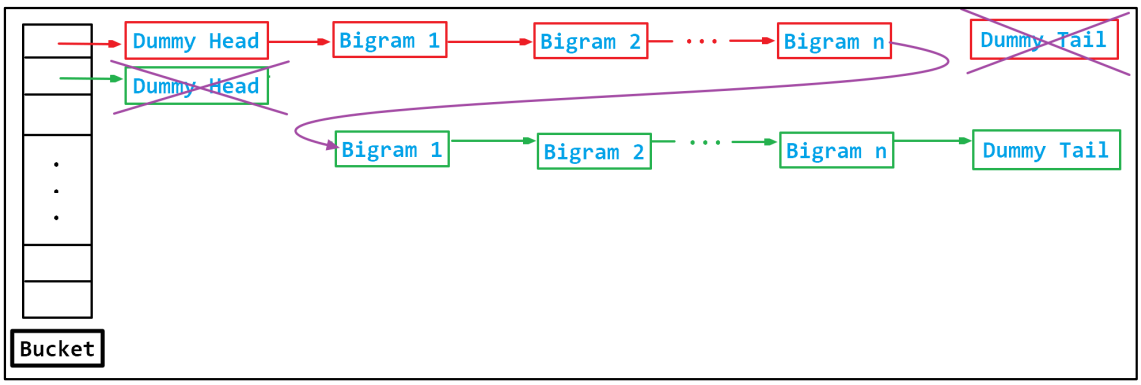
```
List* merge_list(List *plist1, List *plist2) {
    plist1->tail->before->next =plist2->head->next;
    plist2->head->next->before =plist1->tail->before;
```

```
    // Free dummy tail of plist1
    free(plist1->tail);
    // Free dummy head of plist2
    free(plist2->head);

    plist1->num_of_data += plist2->num_of_data;
    plist1->tail = plist2->tail;

    // Merge two list at plist1 and Return
    return plist1;
}
```

병합 이전의 bucket과 병합 이후의 bucket의 상황을 보여주는 그림을 통해 merge_list()의 동작을 설명하면 다음과 같다.



< Fig 10 > 병합 이전의 Bucket



< Fig 11 > 병합 과정을 설명하는 figure

각 bucket의 element로 존재하던 linked list들을 하나로 묶어주는 기능을 하는 것을 Fig 11의 그림을 통해 간단하게 보일 수 있다.

4. sort_list() & print_list()

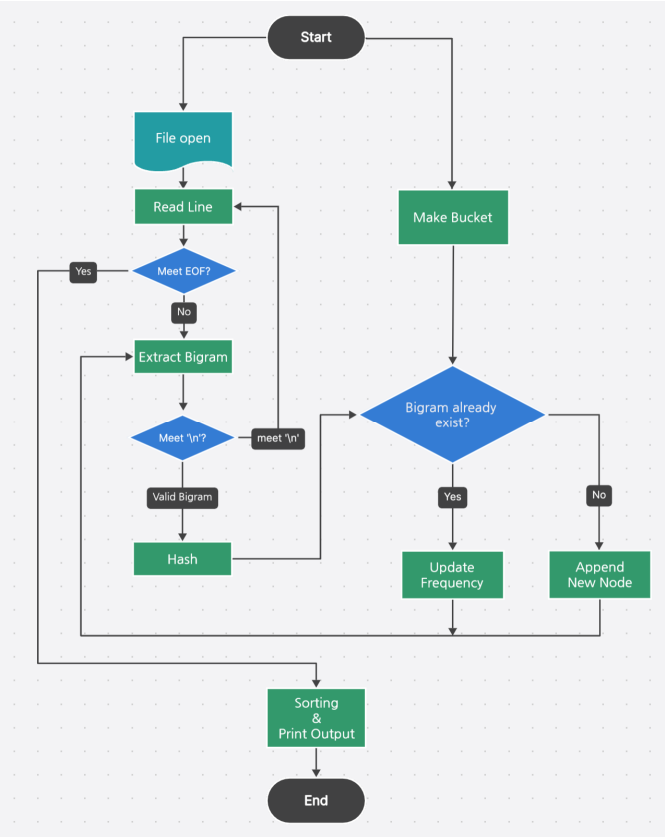
```c
void sort_list(List *merged_list) {
    int i, j;
    Node *key_node = NULL;
    int standard = 0;
    int cmp = 0;

    for(i = 1; i < merged_list->num_of_data; i++){
        key_node = pop_node(merged_list, i);
        standard = key_node->bigram->freq;

        for( j = i - 1; j >= 0; j--){
            cmp = check_node_freq(merged_list, j);
            if (standard < cmp) {
                break;
            }
        }

        insert_node(merged_list, key_node, j + 1);
    }
}
```

< Fig 12 > sort_list()

```c
int print_list(List *plist, int start) {
    set_first(plist);
    int idx = start;

    for (int i = 0; i < plist->num_of_data; i++) {
        printf("\n-+-+-+-[%d] Bigram-+-+-+-\n", idx);
        printf("Front word : %s\n", plist->cur->bigram->fword);
        printf("Back word  : %s\n", plist->cur->bigram->bword);
        printf("Frequency  : %d\n", plist->cur->bigram->freq);
        printf("--------------------------\n");
        next_node(plist);
        idx++;
    }

    return idx;
}
```

< Fig 13 > print_list()

위의 두 함수는 병합된 list를 정렬하고 출력하기 위한 목적의 함수이다. 정렬의 경우 초기 스펙으로 언급된 바와 같이 삽입 정렬을 구현하였다.



< Fig 14 > 프로그램의 전체적 흐름도

프로그램의 기능에 대한 설명을 하나의 흐름도로 정리하면 Fig 14와 같이 표현할 수 있다.

- **초기 버전에 대한 gprof test**

이제 완성한 bigram analyzer 프로그램을 컴파일하고 테스트를 진행해보았다.

```
bmj@linux:~/v1_bigram$ gcc -Og -pg -m64 Bigram.c -o bigram
bmj@linux:~/v1_bigram$ ./bigram test.txt > result_v1.txt
bmj@linux:~/v1_bigram$ gprof ./bigram > gprof_v1.txt
bmj@linux:~/v1_bigram$
```

< Fig 15 > gcc 컴파일 및 gprof 적용 command

Fig 15에 명시된 것처럼 컴파일 옵션으로는 **-Og, -pg, -m64**를 사용하였다. 본 프로젝트를 진행하면서 모든 최적화 단계는 **-Og**으로 통일하여 직접 적용한 최적화 전략이 어떠한 영향을 미치는지 확인해 보았다.

command에 실행시키고 싶은 파일명(Fig 15에서는 test.txt)을 입력한다면, 원하는 파일을 프로그램의 command line argument로 전달할 수 있다. 전달하지 않고 기본적으로 실행한다면, shakespeare.txt를 default로 고려한다. 프로그램의 결과는 redirection을 통해 txt 파일에 출력하는 형태로 실행을 시켰다.

```
bmj         3347  0.0  1.3 382956 27136 ?        Sl   12:04   0:00 /us
bmj         3369  0.0  2.8 726548 57804 ?        Ssl  12:04   0:00 /us
bmj         3420  0.0  0.2  10500  5376 pts/0    Ss   12:04   0:00 bas
bmj         3942 99.9  1.9  41056 39552 pts/0    R+   12:05 381:23 ./B
bmj         3949  0.0  1.2 418164 25860 ?        Sl   12:05   0:00 upd
root        4407  0.0  0.0      0     0 ?        I    14:09   0:00 [kw
root        4482  0.0  0.0      0     0 ?        I    14:54   0:00 [kw
root        4485  0.0  0.0      0     0 ?        I    14:54   0:00 [kw
root        4604  0.0  0.0      0     0 ?        I    15:39   0:01 [kw
root        4804  0.0  0.0      0     0 ?        I    17:28   0:00 [kw
root        4866  0.0  0.0      0     0 ?        I    17:44   0:00 [kw
root        4868  0.0  0.0      0     0 ?        I    17:52   0:00 [kw
root        4871  0.0  0.0   3208  1664 ?        S    18:04   0:00 sle
```

< Fig 16 > 초기 버전에 Shakespeare 소설 전문을 입력으로 준 경우

그런데 초기 bigram analyzer의 경우 Shakespeare 소설 전문을 입력으로 제공했을 때, 프로그램이 끝나지 않고 무한히 실행되는 현상이 발생했다. Fig 16에서 확인할 수 있듯이 ps -aux 명령으로 프로세스 실행시간을 확인해 본 결과, 6시간 반 정도 실행되고 있음에도 불구하고 프로그램이 종료되지 않았다.

이러한 이유로 소설 전문을 사용하는 것이 아니라, 1,200줄 정도를 발췌하여 비교적 짧은 문서로 테스트를 수행해 보았다.

▶ v1 - gprof test

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 82.13     38.85    38.85 3966373442     0.00     0.00  next_node
 11.79     44.44     5.58 18976801     0.00     0.00  check_node_freq
  6.06     47.30     2.87 18998154     0.00     0.00  set_first
  0.02     47.31     0.01        1     0.01    47.31  sort_list
  0.00     47.31     0.00     9097     0.00     0.00  extract_bigram
  0.00     47.31     0.00     8176     0.00     0.00  hash
  0.00     47.31     0.00     8176     0.00     0.00  search_bigram
  0.00     47.31     0.00     6589     0.00     0.00  append_2_head
  0.00     47.31     0.00     6589     0.00     0.00  data_free
  0.00     47.31     0.00     6589     0.00     0.00  init_bigram
  0.00     47.31     0.00     6588     0.00     0.00  insert_node
  0.00     47.31     0.00     6588     0.00     0.00  pop_node
  0.00     47.31     0.00      921     0.00     0.00  lower1
  0.00     47.31     0.00      500     0.00     0.00  list_init
  0.00     47.31     0.00      499     0.00     0.00  merge_list
  0.00     47.31     0.00        1     0.00     0.00  clear_list
  0.00     47.31     0.00        1     0.00     0.00  parse_bigram
  0.00     47.31     0.00        1     0.00     0.00  print_list
```

< Fig 17 > 1,200줄 문서를 입력으로 준 경우 flat profile

```
-+-+-+-[6588] Bigram-+-+-+-
Front word : thou
Back word  : bully
Frequency  : 1
----------------------------

-+-+-+-[6589] Bigram-+-+-+-
Front word : creeping
Back word  : fowler
Frequency  : 1
----------------------------

----------------------
Running time: 275.644s
----------------------
```

< Fig 18 > 따로 측정한 실행시간

Fig 17은 v1의 gporf 테스트 결과이다. 그런데 프로그램 내부에서 main의 시작 부분과 return 직전 부분 사이의 시간 차를 직접 구해본 결과 Fig 18에서 확인할 수 있듯이 4분 35초의 실행 시간이 측정되었다. gprof에서 cumulative seconds에서 확인할 수 있는 47.31초의 결과는 직접 측정한 시간과 너무 큰 오차가 존재했다.

```c
int next_node(List* plist) {
    if (plist->cur->next == plist->tail) {
        return 0;
    } else {
        plist->cur = plist->cur->next;
        return 1;
    }
}
```

< Fig 19 > next_node()

```c
int check_node_freq(List* plist, int idx) {
    int freq = 0;
    set_first(plist);

    for (int i = 0; i < idx; i++) {
        next_node(plist);
    }

    freq = plist->cur->bigram->freq;
    return freq;
}
```

< Fig 20 > check_node_freq()

flat profile에서 call 횟수가 상위권인 next_node() 함수와 check_node_freq() 함수는 linked list를 다루기 위한 함수로 순전히 코드 가독성을 높이기 위해 사용하였다. 실질적으로 이 next_node() 함수의 경우 현재 노드가 마지막 노드인지 비교 연산을 수행하고, 포인터 값 하나를 옮기는 것이 전부이기에 실행시간이 매우 짧다.

check_node_freq()도 loop 반복 시간에 편차가 존재하긴 하더라도, function prologue와 epilogue를 고려하여도 실행 시간이 5~10ms를 넘지 않는다. 따라서 0.01초 단위로 샘플링을 수행하는 gprof 툴의 특성상 위 두 함수가 실행되고 있을 때, 샘플링을 수행할 확률이 높지 않다.

그러한 원인 때문에 Fig 17과 같이 비정상적인 실행 시간 측정이 이루어진 것으로 판단하였다. 아래의 Fig 21, 22를 확인해보면 next_node()는 ms 단위로 측정이 안되었고, check_node_freq()는 0 ~ 4ms 사이의 값이 랜덤으로 측정되는 것으로 측정 오차에 대한 원인을 판단할 수 있다.

```
----------------------------------------
next_node() running time: 0.000ms
----------------------------------------

----------------------------------------
next_node() running time: 0.000ms
----------------------------------------

----------------------------------------
next_node() running time: 0.000ms
----------------------------------------

----------------------------------------
next_node() running time: 0.000ms
----------------------------------------
```

< Fig 21 > next_node() 실행 시간

```
----------------------------------------
check_node_freq() running time: 0.000ms
----------------------------------------

----------------------------------------
check_node_freq() running time: 0.000ms
----------------------------------------

----------------------------------------
check_node_freq() running time: 0.303ms
----------------------------------------

----------------------------------------
check_node_freq() running time: 0.033ms
----------------------------------------
```

< Fig 22 > check_node_freq() 실행시간

하지만, 호출 관계를 보여주는 call graph를 확인해 보면, 정확한 시간 측정이 되지 않더라도 전반적인 프로그램의 문제를 파악할 수 있다. 다음 Fig 23은 v1에 대한 call graph 화면이다.

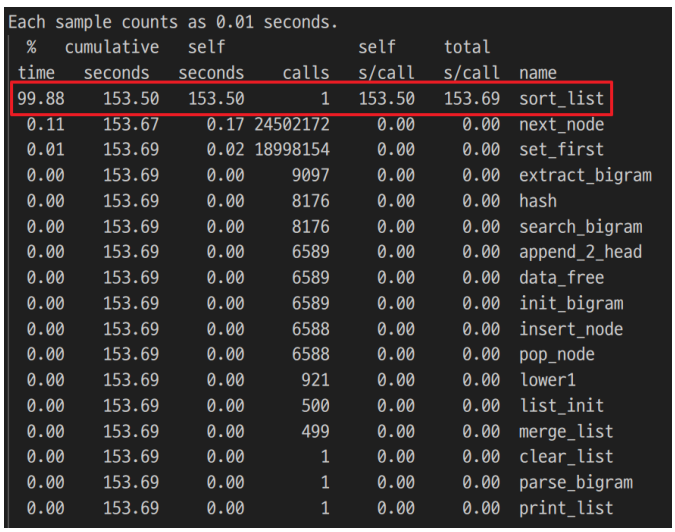
```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00   47.31                 main [1]
                0.01   47.30       1/1           sort_list [2]
                0.00    0.00       1/1           parse_bigram [9]
                0.00    0.00       1/1           print_list [10]
                0.00    0.00     500/500         list_init [17]
                0.00    0.00     499/499         merge_list [18]
                0.00    0.00       1/1           clear_list [19]
-----------------------------------------------
                0.01   47.30       1/1           main [1]
[2]    100.0    0.01   47.30       1         sort_list [2]
                5.58   41.48 18976801/18976801     check_node_freq [3]
                0.00    0.21    6588/6588        pop_node [6]
                0.00    0.03    6588/6588        insert_node [7]
-----------------------------------------------
                5.58   41.48 18976801/18976801     sort_list [2]
[3]     99.5    5.58   41.48 18976801         check_node_freq [3]
               38.61    0.00 3941871270/3966373442     next_node [4]
                2.86    0.00 18976801/18998154     set_first [5]
-----------------------------------------------
                0.00    0.00    6589/3966373442     print_list [10]
                0.00    0.00   57494/3966373442     search_bigram [8]
                0.03    0.00 2733923/3966373442     insert_node [7]
                0.21    0.00 21704166/3966373442     pop_node [6]
               38.61    0.00 3941871270/3966373442     check_node_freq [3]
[4]     82.1   38.85    0.00 3966373442         next_node [4]
-----------------------------------------------
```

< Fig 23 > 1200줄 문서를 입력으로 준 경우 call graph

call 횟수가 제일 많은 함수인 next_node()는 빨간색 박스에서 확인할 수 있듯이, 대부분 check_node_freq() 함수로부터 호출되고 있었다. 그리고 초록색 박스 부분을 체크해보면, check_node_freq() 함수는 sort_list() 함수에서 모든 호출을 수행함을 알 수 있다.

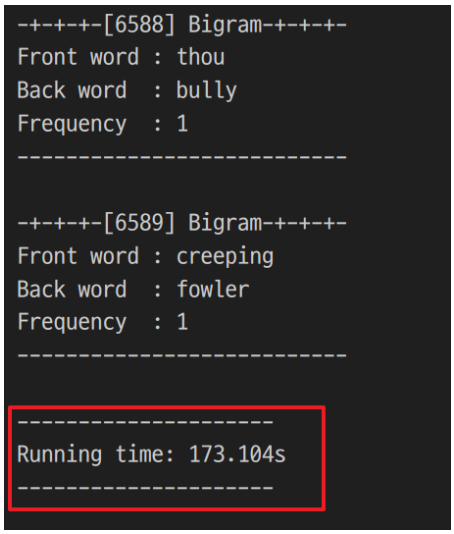
마지막으로 파란색 박스로 강조한 sort_list()를 확인해보면, 실행시간의 100%를 차지하는 것으로 통계를 확인할 수 있다. call graph를 확인해보면 sort_list() 함수가 가장 먼저 최적화해야 할 대상임을 판단 가능하다.

gprof의 실행 시간 측정 오차 문제를 해결하기 위해 부가적으로 실행 시간이 짧은 두 함수를 sort_list() 함수에 병합시킨 뒤 gprof 테스트를 수행해 보았다. 그 결과는 다음과 같다.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.88    153.50   153.50        1   153.50   153.69  sort_list
  0.11    153.67     0.17 24502172     0.00     0.00  next_node
  0.01    153.69     0.02 18998154     0.00     0.00  set_first
  0.00    153.69     0.00     9097     0.00     0.00  extract_bigram
  0.00    153.69     0.00     8176     0.00     0.00  hash
  0.00    153.69     0.00     8176     0.00     0.00  search_bigram
  0.00    153.69     0.00     6589     0.00     0.00  append_2_head
  0.00    153.69     0.00     6589     0.00     0.00  data_free
  0.00    153.69     0.00     6589     0.00     0.00  init_bigram
  0.00    153.69     0.00     6588     0.00     0.00  insert_node
  0.00    153.69     0.00     6588     0.00     0.00  pop_node
  0.00    153.69     0.00      921     0.00     0.00  lower1
  0.00    153.69     0.00      500     0.00     0.00  list_init
  0.00    153.69     0.00      499     0.00     0.00  merge_list
  0.00    153.69     0.00        1     0.00     0.00  clear_list
  0.00    153.69     0.00        1     0.00     0.00  parse_bigram
  0.00    153.69     0.00        1     0.00     0.00  print_list
```

< Fig 24 > sort_list()로 병합 후의 flat profile



```
-+-+-+-[6588] Bigram-+-+-+-
Front word : thou
Back word  : bully
Frequency  : 1
----------------------------

-+-+-+-[6589] Bigram-+-+-+-
Front word : creeping
Back word  : fowler
Frequency  : 1
----------------------------

---------------------
Running time: 173.104s
---------------------
```

< Fig 25 > 직접 측정한 실행시간

아직도 오차가 존재하긴 하지만 flat profile 상에서 직접 측정한 시간과의 오차가 상당히 개선되었다. 이것으로 gprof 툴을 사용함에 있어서 너무 실행 시간이 짧은 함수가 많은 call이 되는 경우 큰 측정 오차를 발생시키는 문제 요인이 될 수 있음을 실험적으로 확인하였다.

하지만, call graph를 잘 활용한다면, 오차가 있더라도 최적화를 위한 우선 대상을 추적하는 것에 있어서는 문제가 없기에 call graph의 호출 관계를 잘 확인하고 분석하는 것도 중요한 과정임을 실감했다.

이제는 명백하게 sort_list()가 가장 실행 시간을 많이 차지하는 부분으로 드러났고, 해당 함수를 제일 먼저 최적화해야 하는 것을 확인할 수 있었다.

Part 2. Optimization

교재 5장에서 학습한 다양한 최적화 전략들을 이용하여, 제작한 Bigram Analyzer에 대한 최적화를 수행하고 개선 정도를 정리하였다. Part 1.에서 초기 버전에 대한 gprof test로 가장 급선무로 최적화해야 할 대상이 sort_list() 함수인 것을 확인하였다. Part 2.에서는 sort_list()에 대한 최적화부터 시작하여 단계별로 프로그램의 최적화를 수행해 보았다.

• sort_list - 정렬 방식 개선하기 (v2)

기존의 sort_list() 함수는 삽입 정렬 방식을 사용하고 있었다. 삽입 정렬은 잘 알려진 바로 최악의 경우 $O(n^2)$의 시간복잡도를 갖는다. 1,200줄의 문서에서는 6,589개의 bigram이 추출되었고, Shakespeare 소설 전문에는 306,847개의 bigram이 추출된다. 단순 정렬 대상인 bigram의 수가 46.57배 차이가 나는데, 이는 $O(n^2)$의 시간복잡도 하에서 대략 2,168배의 실행시간 차이가 발생할 것으로 예상할 수 있다.

1,200줄의 문서를 기준으로 3 ~ 4분의 시간이 소요되는데, 이것의 2,168배라면 프로그램의 실용성이 없어진다. 따라서 최적화를 위해 평균적으로 $O(n \log n)$의 시간복잡도를 갖는 Quick sort로 정렬 방식을 교체하였다.

▶ 퀵 정렬 방식으로 변경한 함수 - sort_quick()

```
void sort_quick(int num, Node *Left, Node *Right, Node *dummy_head) {
    int nhigh = 1;
    int nTotal = num;
    int nlow = nTotal - 1;

    // Terminate the recursive function if there is one node to sort
    if (nlow <= nhigh)
        return;

    Node *Key_node = Right;
    Node *left_node = Left;
    Node *right_node = Right->before;

    while (1) {
        // Explore left partition until there is a value less than
        // the frequency of the key_node
        while ( left_node->bigram->freq >Key_node->bigram->freq) {
```

```
            left_node = left_node->next;
            nhigh++;
        }

        // Explore left partition until there is a value greater than
        // the frequency of the key_node
        while ( right_node->bigram->freq < Key_node->bigram->freq) {
            if(nlow <= 1 || right_node->before == dummy_head)
                break;
            right_node = right_node->before;
            nlow--;
        }

        // Terminate condition of while loop
        if (nhigh >= nlow)
            break;

        // Swap the nodes discovered in the two while loops above
        swap_node(left_node, right_node);

        left_node = left_node->next;
        nhigh++;
        right_node = right_node->before;
        nlow--;
    }

    // Swap key_node and left_node(nhigh)
    swap_node(Key_node, left_node);

    // Recursive function call to left partition
    sort_quick(nhigh - 1, Left, left_node->before, dummy_head);

    // Recursive function call to right partition
    sort_quick(nTotal - nhigh, left_node->next, Right, dummy_head);
}
```

해당 함수는 재귀 기반의 퀵 정렬을 구현한 것이다. index 접근이 어려운 linked list 자료구조의 특성상 함수의 파라미터로 Node* 형을 받고 그것을 갱신해 가며 정렬을 수행하는 방식을 사용하였다. 무작위의 hash map으로 분류되어 정렬이 거의 되어있지 않은 list이므로 quick sort를 사용함에 따른 개선 효과가 상당할 것으로 예상했다.

▶ v2 - gprof test

1,200줄 정도의 텍스트를 가지고 테스트를 수행한 v1과 비교를 하기 위해서 똑같은 텍스트를 v2의 입력으로 제공하였다. 다음의 Fig 27, 28은 v2에 대한 profiling 결과이다.

```
bmj@linux:~/v1_bigram$ gcc -Og -pg -m64 Bigram.c -o bigram_v2
bmj@linux:~/v1_bigram$ ./bigram_v2 test.txt > result_v2_short.txt
bmj@linux:~/v1_bigram$ gprof ./bigram_v2 > gporf_v2_short.txt
bmj@linux:~/v1_bigram$
```

< Fig 26 > gprof test command

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00    64083     0.00     0.00  next_node
  0.00      0.00     0.00    35155     0.00     0.00  swap_node
  0.00      0.00     0.00     9097     0.00     0.00  extract_bigram
  0.00      0.00     0.00     8177     0.00     0.00  set_first
  0.00      0.00     0.00     8176     0.00     0.00  hash
  0.00      0.00     0.00     8176     0.00     0.00  search_bigram
  0.00      0.00     0.00     6589     0.00     0.00  append_2_head
  0.00      0.00     0.00     6589     0.00     0.00  data_free
  0.00      0.00     0.00     6589     0.00     0.00  init_bigram
  0.00      0.00     0.00      921     0.00     0.00  lower1
  0.00      0.00     0.00      500     0.00     0.00  list_init
  0.00      0.00     0.00      499     0.00     0.00  merge_list
  0.00      0.00     0.00        1     0.00     0.00  clear_list
  0.00      0.00     0.00        1     0.00     0.00  parse_bigram
  0.00      0.00     0.00        1     0.00     0.00  print_list
  0.00      0.00     0.00        1     0.00     0.00  sort_quick
```

< Fig 27 > v2의 1,200-line text에 대한 flat profile

```
-+-+-+-[6588] Bigram-+-+-+-
Front word : thorny
Back word  : hedge
Frequency  : 1
----------------------------

-+-+-+-[6589] Bigram-+-+-+-
Front word : wildest
Back word  : hath
Frequency  : 1
----------------------------


----------------------
Running time: 2.478ms
----------------------
```

< Fig 28 > v2 실행시간

기본적으로 프로그램을 실행시키자마자 프로세스가 종료되는 것을 확인할 수 있었다. 173.104s가 걸렸던 프로그램이 2.478ms 만에 완료되어 69,856배 성능이 개선되었다. 적은 용량의 text에 대해서는 충분히 좋은 성능을 보이는 것을 알 수 있다. 이제 더욱더 최적화를 수행하기 위해 Shakespeare 소설 전문을 입력으로 제공하여 gprof를 다시 테스트해 보았다.

▶ v2 - Shakespeare 소설 입력 후 gprof test

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 96.39      7.21     7.21   838792     0.00     0.00  search_bigram
  1.34      7.31     0.10                             _init
  0.67      7.36     0.05 266402589     0.00     0.00  next_node
  0.67      7.41     0.05        1     0.05     0.05  sort_quick
  0.53      7.45     0.04        1     0.04     0.04  print_list
  0.13      7.46     0.01   838792     0.00     0.00  hash
  0.13      7.47     0.01    93166     0.00     0.00  lower1
  0.13      7.48     0.01        1     0.01     0.01  clear_list
  0.00      7.48     0.00  2453895     0.00     0.00  swap_node
  0.00      7.48     0.00   931958     0.00     0.00  extract_bigram
  0.00      7.48     0.00   838793     0.00     0.00  set_first
  0.00      7.48     0.00   306847     0.00     0.00  append_2_head
  0.00      7.48     0.00   306847     0.00     0.00  data_free
  0.00      7.48     0.00   306847     0.00     0.00  init_bigram
  0.00      7.48     0.00      500     0.00     0.00  list_init
  0.00      7.48     0.00      499     0.00     0.00  merge_list
  0.00      7.48     0.00        1     0.00     7.28  parse_bigram
```

< Fig 29 > v2의 소설 전문에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : hunter
Back word  : every
Frequency  : 1
----------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : horses
Back word  : swift
Frequency  : 1
----------------------------


----------------------------
Running time: 11482.324ms (11.482s)
----------------------------
```

< Fig 30 > v2 실행시간

정렬 알고리즘을 수정한 프로그램은 이제 소설 전체에 대해서도 정상적으로 작동하고, 11.48s 정도의 실행시간이 측정되었다. Fig 29를 확인해 보면 정렬을 수행하는 sort_quick()는 전체 프로그램 실행 시간의 100%를 차지하던 것에서 0.67% 정도를 차지하는 함수로 성능 개선이 완료된 것을 확인할 수 있었다.

이제 Fig 29 flat profile의 상위에 있는 함수는 search_bigram()이다.

```
int search_bigram(List *plist, char *fword, char *bword) {
    set_first(plist);
    int idx = 0;
    for (; idx < plist->num_of_data; idx++) {
        if (!strcmp(plist->cur->bigram->fword, fword) && !strcmp(plist->cur->bigram->bword, bword)) {
            plist->cur->bigram->freq++;
            break;
        }
        next_node(plist);
    }

    if (idx == plist->num_of_data) {
        idx = -1;
    }

    return idx;
}
```

< Fig 31 > search_bigram()

해당 함수는 문자열 비교를 통해 List 안에 중복되는 bigram이 존재하는지를 판단하고 처리하는 함수이다. 따라서 성능을 담당하는 요인은 비교하는 횟수이다. 초록색 박스 안의 비교를 많이 수행하면 할수록 메모리에 대한 접근과 뺄셈 연산 횟수가 증가할 것이다.

해당 루프의 반복 횟수는 list에 존재하는 num_of_data에 의해 결정된다. 즉, 하나의 bucket에 hash 값에 따라 분류된 노드의 개수가 search_bigram에서 비교 횟수와 비례하게 된다. bucket의 개수를 늘려 bigram을 분산시키면 비교 횟수를 효과적으로 줄일 수 있을 것으로 판단하였고 다음 최적화 방법으로 채택하였다.

• search_bigram - bucket 개수 증가 (v3)

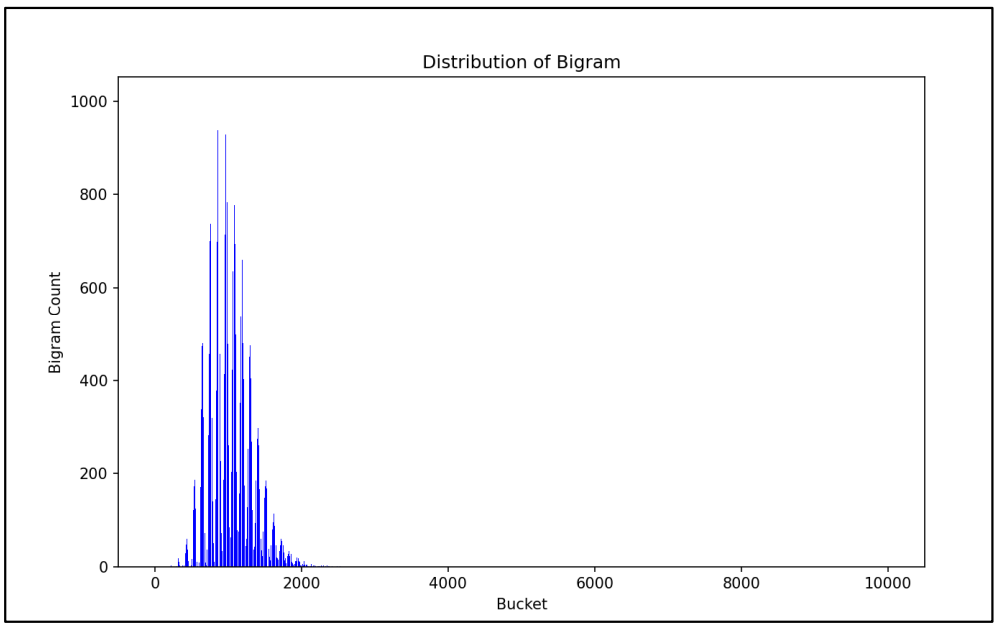
v3에서는 bucket의 개수와 성능의 관계성을 살펴보기 위해 bucket의 개수를 다양하게 조정해 보며 실행시간을 테스트해 보았다. #define directive로 정의해 둔 BUCKET은 지금까지 500으로 설정해두고 있었다. 이것을 5,000 ~ 200,000까지 변경해 가며 gprof를 테스트하고 다음과 같이 정리하였다.

▶ BUCKET의 개수에 따른 성능 개선 정도

BUCKET	gprof test	self test	초기값 대비 성능 개선
500	7.21s	11.482s	-
5,701	2.84s	5.978s	1.92x
8,039	3.60s	7.055s	1.63x
10,007	3.26s	6.762s	1.70x
21,649	2.46s	5.755s	2.00x
41,023	3.97s	7.201s	1.59x
88,397	3.22s	6.670s	1.72x
128,033	2.76s	6.159s	1.86x
150,847	3.32s	6.663s	1.72x
183,499	3.74s	6.708s	1.71x
200,003	2.56s	6.369s	1.80x

테스트 결과 버킷의 수를 늘리는 것이 성능의 개선에 영향을 미치는 것은 확인할 수 있다. 하지만, 버킷의 수와 성능 개선 정도가 단순 비례 관계에 있는 것은 아닌 것을 확인할 수 있었다. 왜 Bucket의 증가가 search_bigram() 함수의 성능 개선으로 직결되지 않는지 분석을 수행했다.

Bucket의 수를 10,007개로 지정하고, 각 bucket에 몇 개의 bigram이 존재하는지 분포도를 확인해 보면 다음과 같은 결과가 나온다.



< Fig 32 > 버킷에 따른 bigram 분포 (v3 - BUCKET_10,007)

Fig 32의 분포를 확인하면 주로 0 ~ 2000번의 bucket 쪽에 거의 모든 데이터가 몰려있으며, 2000번보다 큰 bucket들엔 거의 분포하고 있지 않다. 결국 bucket을 늘려도 이러한 분포에 대한 문제는 해결하지 못하고, 저조한 성능의 개선을 보이는 것이다.

이러한 분석의 결과로부터 다음 최적화 대상은 hash 함수를 지목하였고, bucket에 bigram을 균등하게 분포시키는 방향으로 개선을 수행하였다.

• hash - 균등 분배 (v4)

```c
int hash(char *fword, char *bword) {
    int ascii_sum = 0;
    char ch = fword[0];

    for (int i = 1; ch != '\0'; i++) {
        ascii_sum += ch;
        ch = fword[i];
    }

    ch = bword[0];
    for (int i = 1; ch != '\0'; i++) {
        ascii_sum+= ch;
        ch = bword[i];
    }

    return (ascii_sum % BUCKET);
}
```

< Fig 33 > 기존 hash()

```c
int improved_hash(char *fword, char *bword) {
    uint64_t hash = 0;

    int len_fword = strlen(fword);
    int len_bword = strlen(bword);

    for (int i = 0; i < len_fword; i++) {
        hash = 31 * hash + fword[i];
    }

    for (int i = 0; i < len_bword; i++) {
        hash = 31 * hash + bword[i];
    }

    int check = (hash % BUCKET);

    return (hash % BUCKET);
}
```

< Fig 34 > 개선한 improved_hash()

Fig 34에 있는 improved_hash는 다음의 수식을 코드로 구현한 것이다. arr은 hashing을 수행하고자 하는 문자열을 의미하고 n 값은 strlen(arr)의 반환 값, 문자열의 길이를 의미한다.

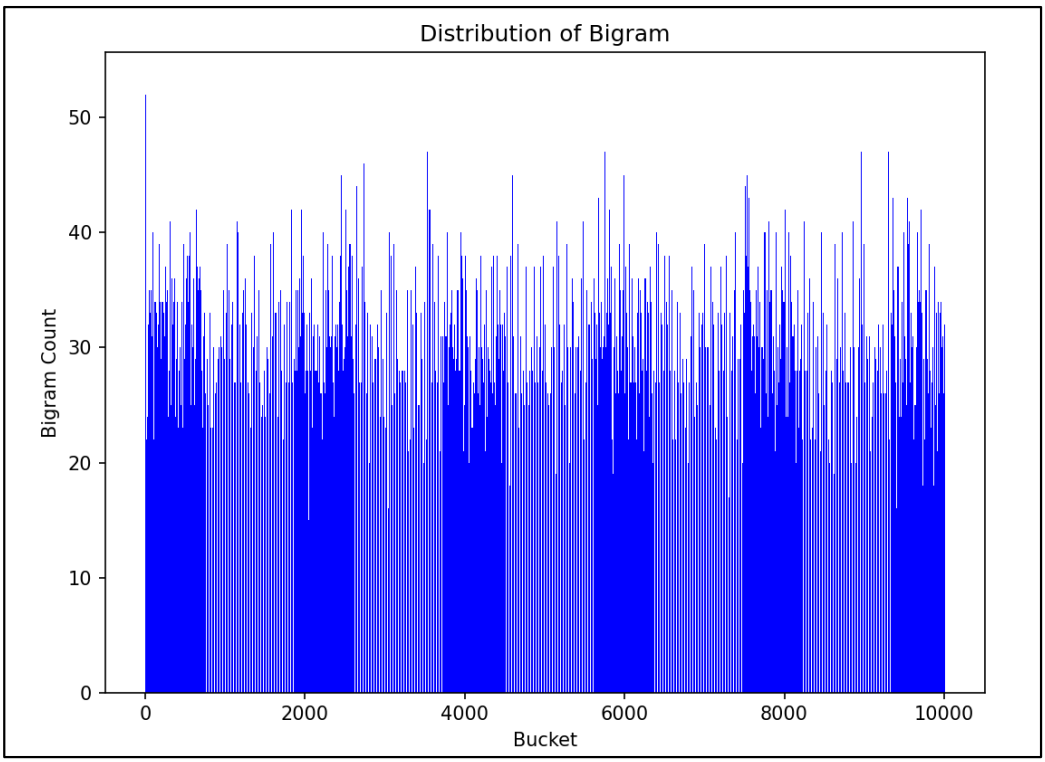
$$hash = (arr[0] * 31^{n} + arr[1] * 31^{n-1} + \cdots + arr[n-1] * 31^{0}) \bmod BUCKET$$

다항식 구조를 갖는 식은 상당히 큰 수를 반환하기에 inttypes.h에서 uint64_t 자료형을 활용하여 overflow에 대한 영향을 최소화하고, 균등한 분포로 hash 값을 반환하도록 함수를 작성하였다.

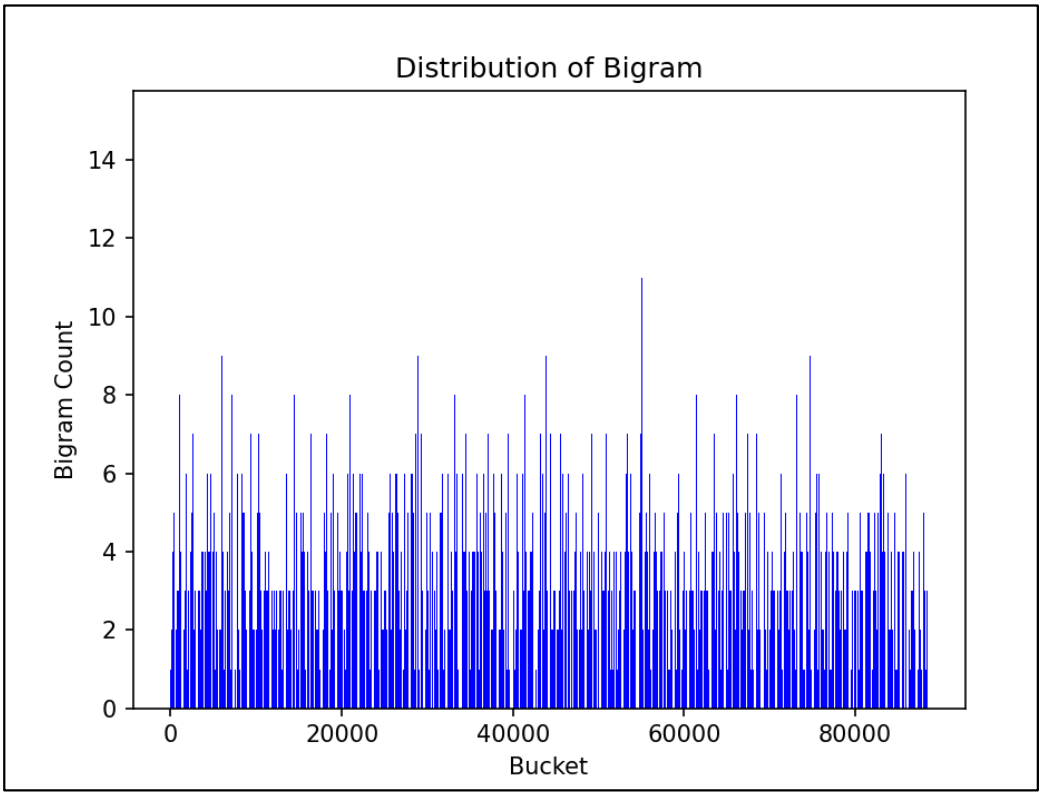
이런 식의 구조를 갖는 improved_hash() 함수를 바탕으로 v3에서 BUCKET을 조정하며 성능 개선 정도를 확인하는 과정을 동일하게 반복하였다.

▶ BUCKET의 개수에 따른 성능 개선 정도

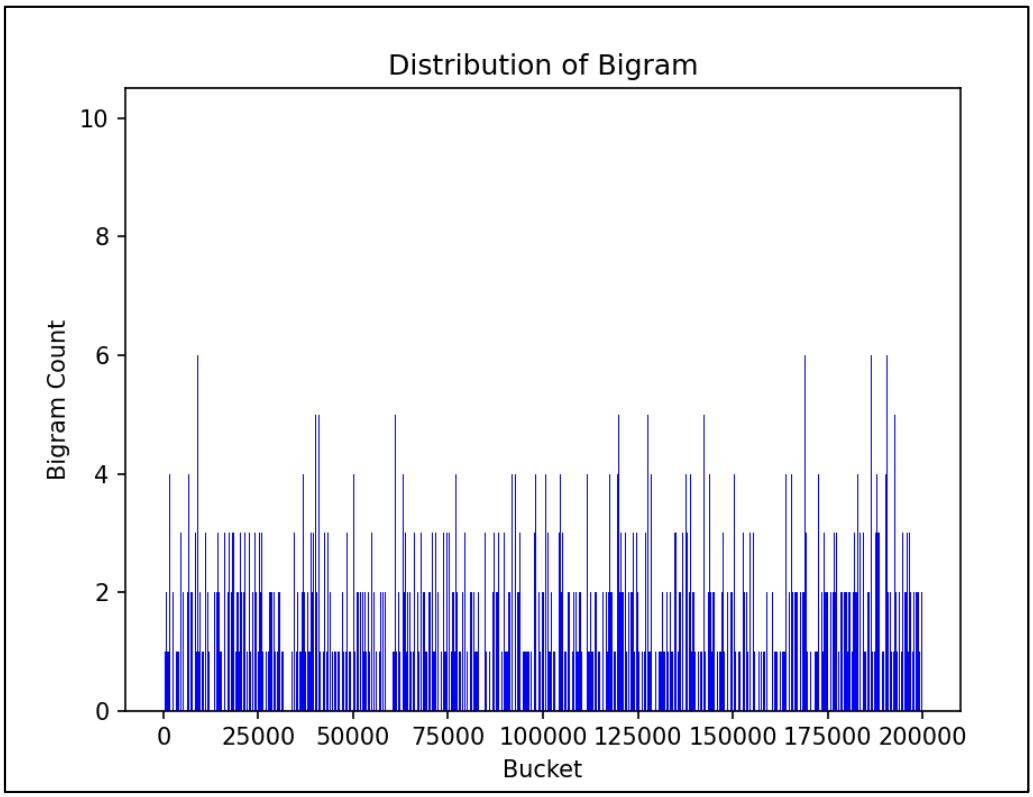
BUCKET	gprof test search_bigram()	self test running time	초기값 대비 성능 개선	v3 동일 bucket과 비교
500 [기존 hash() 함수]	7.21s	11.482s	-	-
500	6.25s	10.559s	1.09x	1.09x
5,701	0.59s	1.326s	8.66x	4.51x
8,039	0.32s	0.979s	11.73x	7.21x
10,007	0.41s	0.937s	12.25x	7.22x
21,649	0.17s	0.741s	15.50x	7.77x
41,023	0.10s	0.610s	18.82x	11.80x
88,397	0.06s	0.534s	21.50x	12.49x
128,033	0.06s	0.495s	23.20x	12.44x
150,847	0.10s	0.530s	21.66x	12.57x
183,499	0.06s	0.486s	23.62x	13.80x
200,003	0.07s	0.500s	22.96x	12.74x



< Fig 35 > 버킷에 따른 bigram 분포 (v4 - BUCKET_10,007)



< Fig 36 > 버킷에 따른 bigram 분포 (v4 - BUCKET_88,397)



< Fig 37 > 버킷에 따른 bigram 분포 (v4 - BUCKET_200,003)

Fig 35 ~ 37로부터 새롭게 개선한 improved_hash 함수가 이전보다 다양한 bucket에 골고루 bigram을 분배하는 것을 확인할 수 있다. 새로운 해시 함수를 적용하면서 BUCKET을 증가시킴에 따라 8 ~ 20배 가량 성능 개선 효과를 얻을 수 있었다. 이제 공간 복잡도와 연계하여 자원을 적게 소비하면서 많은 성능 개선도 얻을 수 있는 BUCKET 수는 무엇일지 분석해 보았다.

위의 성능 개선 정도 표에서 초기값 대비 성능 개선은 BUCKET이 10만 개 정도에 다가가면 20~22배 정도로 머물게 되고, 그 이후로는 그렇게 커다란 성능 개선을 보이지는 않는다. 그 원인은 소설의 전체 Bigram 수인 306,847개로 bucket의 수가 근접해 가기 때문이다.

Bigram 수 대비 bucket의 수가 적은 상황에서는 bucket의 수를 조금 올려도 극적인 성능 상승을 보일 수 있다. 하지만, 전체 bigram과 bucket의 수가 비슷한 scale을 가져가는 상황에서는 bucket의 수를 올리면 성능이 올라가긴 하나, 사용하는 메모리에 대비한 성능 이득은 그렇게 크지 않다고 판단된다.

따라서 v3 ~ v4의 최적화 과정을 거쳐 bucket의 수를 100,003으로 지정하고 다음의 최적화 과정을 수행하기로 하였다.

▶ v4 - BUCKET = 100,003일 때 gprof test

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 22.73      0.05     0.05   838792     0.00     0.00  search_bigram
 22.73      0.10     0.05        1    50.00    50.00  print_list
 18.18      0.14     0.04        1    40.00    40.00  sort_quick
 13.64      0.17     0.03   838793     0.00     0.00  set_first
  9.09      0.19     0.02   306847     0.00     0.00  data_free
  9.09      0.21     0.02        1    20.00   110.00  parse_bigram
  4.55      0.22     0.01    93166     0.00     0.00  lower1
  0.00      0.22     0.00  2450649     0.00     0.00  swap_node
  0.00      0.22     0.00  1550811     0.00     0.00  next_node
  0.00      0.22     0.00   931958     0.00     0.00  extract_bigram
  0.00      0.22     0.00   838792     0.00     0.00  improved_hash
  0.00      0.22     0.00   306847     0.00     0.00  append_2_head
  0.00      0.22     0.00   306847     0.00     0.00  init_bigram
  0.00      0.22     0.00   100003     0.00     0.00  list_init
  0.00      0.22     0.00   100002     0.00     0.00  merge_list
  0.00      0.22     0.00        1     0.00    20.00  clear_list
```

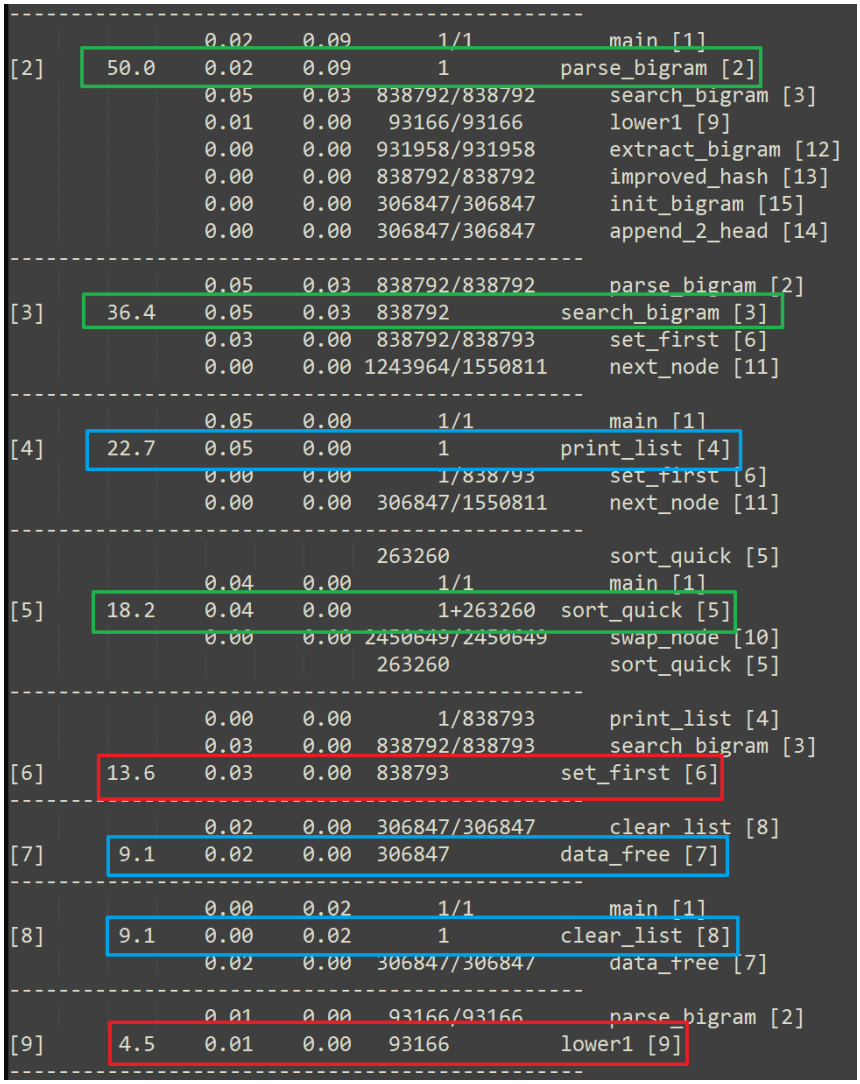
< Fig 38 > v4에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : will
Back word  : rain
Frequency  : 1
----------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : then
Back word  : feeling
Frequency  : 1
----------------------------

----------------------------
Running time: 590.891ms (0.591s)
----------------------------
```

< Fig 39 > v4 실행 시간

Fig 39의 실행 시간을 보면 590.89ms가 소요되었고, 위의 Fig 30에서 v2의 실행 시간은 11.482s가 소요되었다. v3와 v4의 최적화 과정을 거쳐 19.4배만큼 성능이 개선되었다. 이제 flat profile과 call graph를 확인해 보면, Part 1.에서 소개했던 주요 기능 함수들의 시간 소요가 비슷해져 독단적으로 실행 시간을 차지하는 요소는 상당 수준 제거한 것으로 판단할 수 있다.

이제 call_graph의 상위 항목들을 살펴보면 다음과 같다.

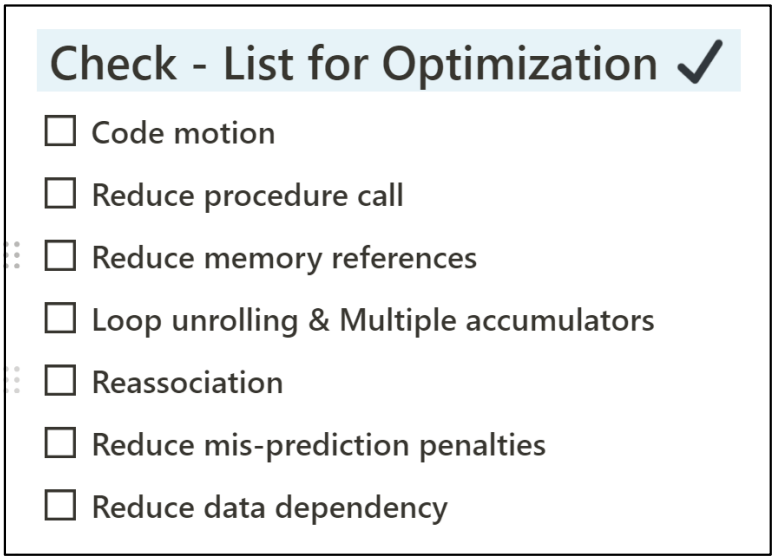


< Fig 40 > v4에 대한 call graph

Fig 40에서 초록색으로 강조한 부분은 지금까지의 최적화 과정을 거치면서 성능을 개선한 함수이다. 그리고 파란색으로 강조된 부분은 결과 출력과 메모리 할당 및 반환과 관련된 함수로 더 이상 최적화가 불가능하다.

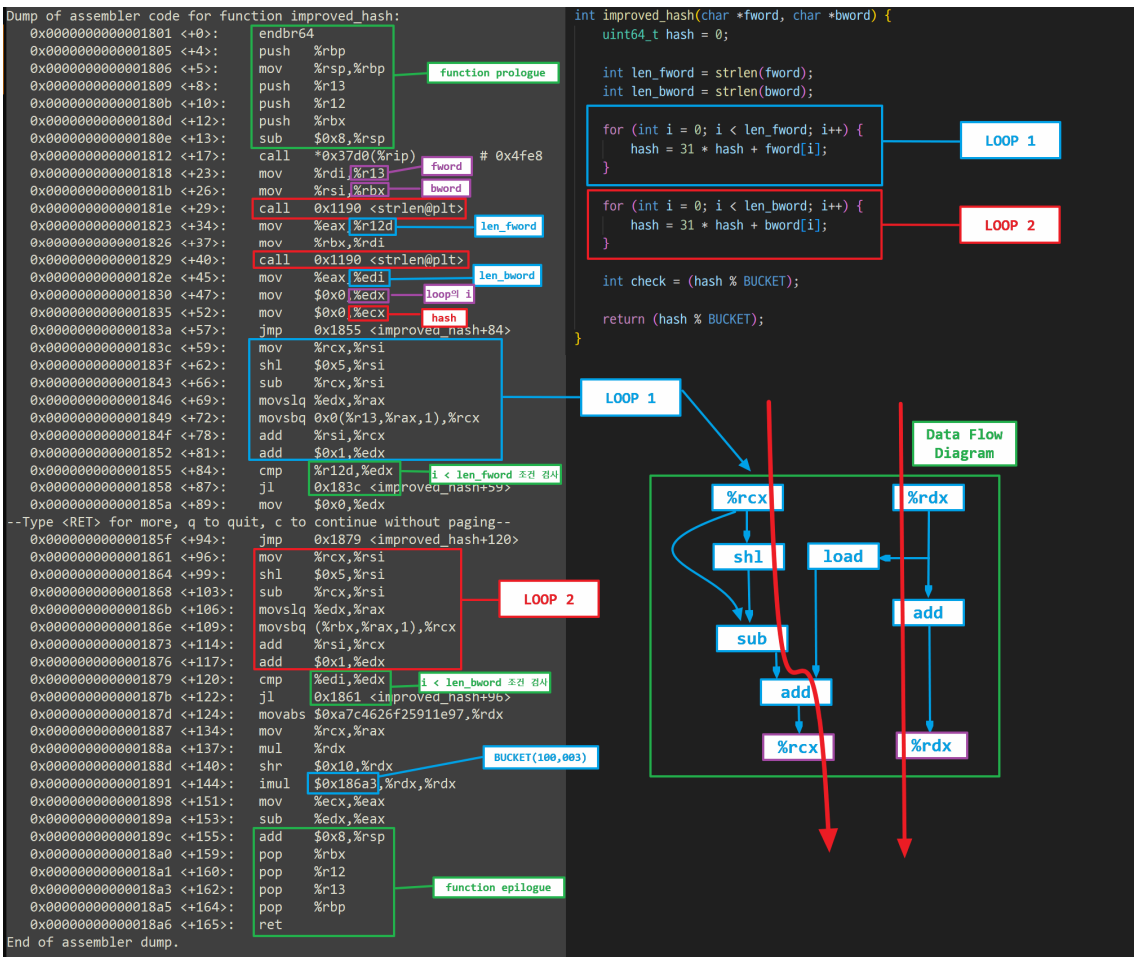
위의 두 경우를 제외하면 빨간색과 같이 프로그램 전체에서 크게 비중을 차지하지 않는 함수들만 남는다. 따라서 최적화의 마무리 단계에 접어들었다고 판단했고, 전체 프로그램에 영향을 적게 미치더라도 최적화가 가능한 요소의 탐색을 수행하였다.

다음 최적화 단계부터는 교재 5장에서 학습한 전략들을 check-list화 한 뒤, 각각의 함수에 적용할 만한 최적화 요소가 존재하는지 1:1로 확인해 가며 마무리 단계에 들어갈 것이다.



< Fig 41 > 최적화 요소 리스트

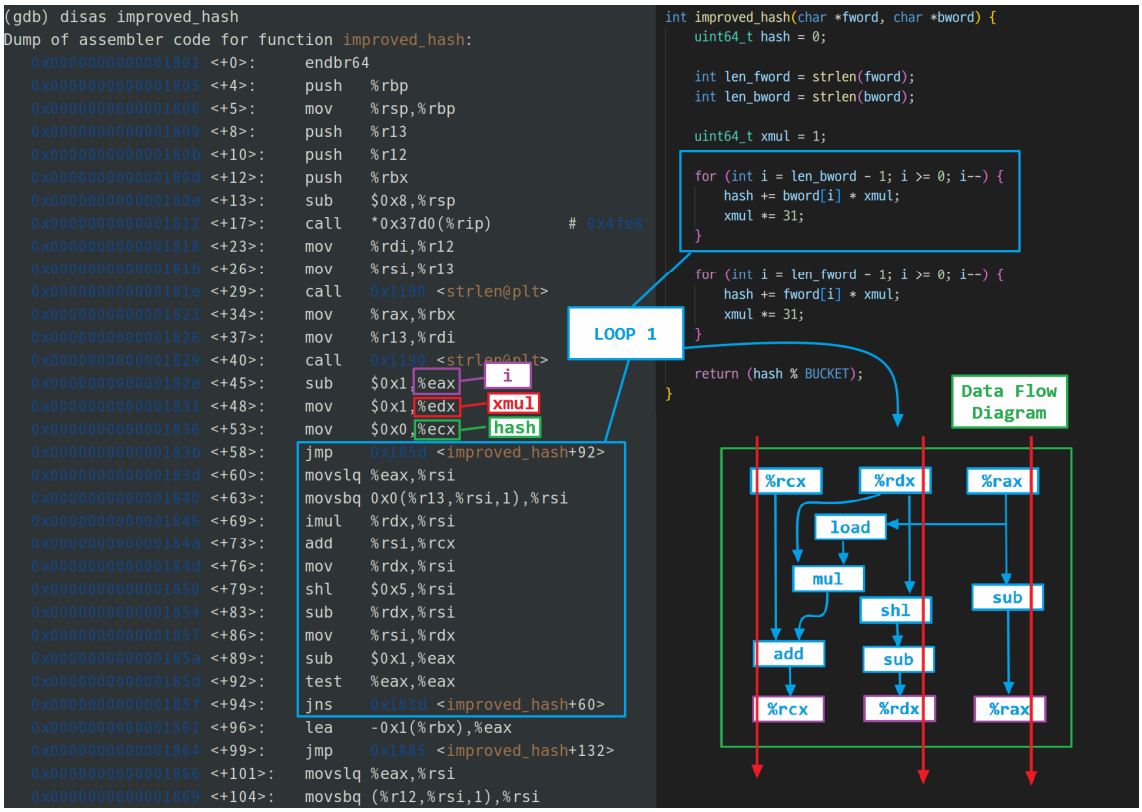
- **improved_hash - 데이터 의존성 (v5)**



< Fig 42 > gdb로 확인한 improved_hash()의 어셈블리 & Data Flow Diagram

improved_hash() 함수에 대해 어셈블리를 조사하고, loop register에 대한 Data flow diagram을 그려 Fig 42처럼 정리하였다. Diagram을 확인해 보면 %rcx 레지스터(hash 변수)는 31을 곱하는 연산으로써 [shift left + sub] 연산과 add 연산을 수행하여 다시 %rcx에 저장하는 구조를 갖추고 있다. 이러한 구조 속에서 add 연산은 shift와 sub 연산이 수행되고 나서야 수행할 수 있는 의존성을 갖는다.

해당 의존성을 해결하고자, 변수를 2개로 분리하여 다음과 같이 수정하였다.



< Fig 43 > 수정한 improved_hash 함수의 어셈블리 & Data Flow Diagram

hash 변수 하나에 31을 곱한 뒤 새로운 문자의 ASCII 코드를 더하는 방식에서, xmul이라는 accumulator를 하나 두어 문자마다 곱해줄 31^x을 계산하여 곱하는 방식으로 치환하였다. 수정된 hash 함수는 수정 이전의 함수와 동일한 hash 값을 반환하는 것을 확인했다.

수정된 improved_hash() 함수에 대한 data flow diagram을 Fig 43처럼 그려보면, chain은 3개가 생기지만, critical path(%rdx chain) 상에는 shift 연산과 sub 연산만 존재하고 add 연산은 다른 chain으로 분리되었다. shift, add, sub 연산이 하나의 chain으로 연관되어 있던 구조에서 add를 분리하면서 cpu의 parallelism을 활용한 성능 최적화가 가능할 것으로 판단하였다. 이러한 방식이 정말 성능을 개선하는지 테스트를 수행해 보았다.

▶ v5 gprof test

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 28.57      0.06     0.06        1    60.00    60.00  print_list
 23.81      0.11     0.05   838793     0.00     0.00  set_first
 19.05      0.15     0.04   838792     0.00     0.00  search_bigram
  9.52      0.17     0.02        1    20.00   120.00  parse_bigram
  9.52      0.19     0.02        1    20.00    20.00  sort_quick
  4.76      0.20     0.01   931958     0.00     0.00  extract_bigram
  4.76      0.21     0.01   306847     0.00     0.00  data_free
  0.00      0.21     0.00  2450649     0.00     0.00  swap_node
  0.00      0.21     0.00  1550811     0.00     0.00  next_node
  0.00      0.21     0.00   838792     0.00     0.00  improved_hash
  0.00      0.21     0.00   306847     0.00     0.00  append_2_head
  0.00      0.21     0.00   306847     0.00     0.00  init_bigram
  0.00      0.21     0.00   100003     0.00     0.00  list_init
  0.00      0.21     0.00   100002     0.00     0.00  merge_list
  0.00      0.21     0.00    93166     0.00     0.00  lower1
  0.00      0.21     0.00        1     0.00    10.00  clear_list
```

< Fig 44 > v5에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : will
Back word  : rain
Frequency  : 1
---------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : then
Back word  : feeling
Frequency  : 1
---------------------------


---------------------------
Running time: 472.138ms (0.472s)
---------------------------
```

< Fig 45 > v5 실행 시간

가장 먼저 주목할 점은 Fig 44에서 search_bigram(), parse_bigram(), sort_quick(), extract_bigram()의 주요한 함수들이 출력 함수인 print_list()보다 실행 시간이 적어진 것이다. 프로그램의 총 실행 시간은 직접 계산했을 때, 472.14ms가 측정되었는데, v4에 대해 측정한 시간인 590.89ms 대비 1.25배만큼 약간 개선되었다.

다음 최적화 대상으로는 Fig 44에서 print_list()를 제외하고 제일 상위에 있는 set_first() 함수를 지정했다.

• set_first & next_node - reduce call (v6)

```c
int set_first(List* plist) {
    if (plist->head->next == plist->tail) {
        return 0;
    } else {
        plist->cur = plist->head->next;
        return 1;
    }
}


int next_node(List* plist) {
    if (plist->cur->next == plist->tail) {
        return 0;
    } else {
        plist->cur = plist->cur->next;
        return 1;
    }
}
```

< Fig 46 > set_first & next_node

```c
int search_bigram(List *plist, char *fword, char *bword) {
    set_first(plist);
    int idx = 0;
    for (; idx < plist->num_of_data; idx++) {
        if (!strcmp(plist->cur->bigram->fword, fword) && !strcmp(plist->cur->bigram->bword, bword))
            plist->cur->bigram->freq++;
            break;
        }
        next_node(plist);
    }

    if (idx == plist->num_of_data) {
        idx = -1;
    }

    return idx;
}
```

< Fig 47 > set_first를 사용하는 search_bigram

Fig 47을 확인해 보면 이미 list에 접근하는 횟수는 plist의 num_of_data 항목으로 제어하고 있다. set_first 함수는 이런 상황에서는 불필요한 function prologue로 소요되는 시간만 증가시킬

것으로 판단하였고, 코드를 callee function에 옮겨넣었다. next_node도 set_first와 마찬가지로 수정했다.

▶ v6 gprof test

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 47.62      0.10     0.10   838792     0.00     0.00  search_bigram
 19.05      0.14     0.04        1    40.00    40.00  print_list
 14.29      0.17     0.03        1    30.00    30.00  sort_quick
  9.52      0.19     0.02   306847     0.00     0.00  data_free
  4.76      0.20     0.01        1    10.00    30.00  clear_list
  4.76      0.21     0.01        1    10.00   110.00  parse_bigram
  0.00      0.21     0.00  2450649     0.00     0.00  swap_node
  0.00      0.21     0.00   931958     0.00     0.00  extract_bigram
  0.00      0.21     0.00   838792     0.00     0.00  improved_hash
  0.00      0.21     0.00   306847     0.00     0.00  append_2_head
  0.00      0.21     0.00   306847     0.00     0.00  init_bigram
  0.00      0.21     0.00   100003     0.00     0.00  list_init
  0.00      0.21     0.00   100002     0.00     0.00  merge_list
  0.00      0.21     0.00    93166     0.00     0.00  lower1
```

< Fig 48 > v6에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : will
Back word  : rain
Frequency  : 1
--------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : then
Back word  : feeling
Frequency  : 1
--------------------------


--------------------------
Running time: 487.736ms (0.488s)
--------------------------
```

< Fig 49 > v6 실행 시간

v6는 예상과 다르게 실행 시간에 있어서 개선되는 결과는 보이지 않았고, Fig 49에서 보이듯 경우에 따라 v5보다 더 많은 실행 시간이 걸린 경우도 존재했다. 따라서 v6에서 수행한 최적화는 유의미한 최적화는 아닌 것으로 확인했다.

• lower1 - code motion (v7)

```c
void lower1(char *s) {
    long i;

    for (i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}
```

< Fig 50 > lower1()

```c
void lower2(char *s) {
    long i;
    long len = strlen(s);

    for (i = 0; i < len; i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a') ;
        }
    }
}
```

< Fig 51 > lower2

v7에서는 문자열을 소문자로 치환해 주는 함수인 lower1을 최적화하였다. Fig 50에서는 strlen이 동일한 문자열에 대해서 매번 같은 값을 반환해 주지만, 매 loop마다 새롭게 계산을 수행하고 있다. strlen() 함수의 호출을 loop 밖으로 옮기는 code motion을 수행하여 Fig 51처럼 수정하였다.

수정한 lower2를 search_bigram() 함수에 적용하고, gprof 테스트를 수행하면 다음과 같다.

▶ v7 gprof test

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 29.17      0.07     0.07   838792     0.00     0.00  search_bigram
 20.83      0.12     0.05   838793     0.00     0.00  set_first
 20.83      0.17     0.05        1    50.00    50.00  print_list
 20.83      0.22     0.05        1    50.00    50.00  sort_quick
  4.17      0.23     0.01   306847     0.00     0.00  data_free
  4.17      0.24     0.01        1    10.00   130.00  parse_bigram
  0.00      0.24     0.00  2450649     0.00     0.00  swap_node
  0.00      0.24     0.00  1550811     0.00     0.00  next_node
  0.00      0.24     0.00   931958     0.00     0.00  extract_bigram
  0.00      0.24     0.00   838792     0.00     0.00  improved_hash
  0.00      0.24     0.00   306847     0.00     0.00  append_2_head
  0.00      0.24     0.00   306847     0.00     0.00  init_bigram
  0.00      0.24     0.00   100003     0.00     0.00  list_init
  0.00      0.24     0.00   100002     0.00     0.00  merge_list
  0.00      0.24     0.00    93166     0.00     0.00  lower2
  0.00      0.24     0.00        1     0.00    10.00  clear_list
```

< Fig 52 > v7에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : will
Back word  : rain
Frequency  : 1
----------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : then
Back word  : feeling
Frequency  : 1
----------------------------

----------------------------
Running time: 478.328ms (0.478s)
----------------------------
```

< Fig 53 > v7 실행 시간

ms 단위로 측정한 프로그램의 실행 시간은 이제 유의미한 개선이 관찰되지 않는 수준에 도달했다.

• merge_list - loop unrolling (v8)

```
// Merge all list that seperated to hash map(bucket)
List *merged_list = bucket[0];
for(int i = 1; i < BUCKET; i++) {
    merged_list = merge_list(merged_list, bucket[i]);
}
```

< Fig 54 > hash map의 모든 list를 하나로 병합하는 루프

위 Fig 54는 main 함수에서 각 bucket에 저장된 모든 list를 하나의 list로 합치는 for loop이다. merge_list 함수는 기본적으로 memory load와 store를 바탕으로 하는 memory access 연산을 주로 수행한다. 교재 기준 5.12절에 의하면, write/read 의존성은 같은 메모리 위치에 대해 연산이 수행되어 두 처리의 순서가 지켜져야 하는 경우에 발생한다.

기본적으로 같은 메모리에 대한 load와 store 연산이 아니라면, load와 store 연산 간 병렬 수행이 가능할 것으로 판단하였다. 따라서 위의 loop에 대해 accumulator의 개념의 List*를 여러 개 선언하고 한 번의 iteration에 2개의 list를 병합하는 구조로 loop unrolling을 시도해 보았다.

다음의 Fig 55는 accumulator를 2개를 사용한 코드이고, Fig 56은 accumulator를 3개를 사용한 코드이다.

```c
List *merged_list = NULL;
List *acc1 = bucket[0];
List *acc2 = bucket[1];
int index = 0;
for(index = 2; index < BUCKET - 1; index += 2) {
    acc1 = merge_list(acc1, bucket[index]);
    acc2 = merge_list(acc2, bucket[index + 1]);
}

for(; index < BUCKET; index++) {
    acc1 = merge_list(acc1, bucket[index]);
}

merged_list = merge_list(acc1, acc2);
```

< Fig 55 > accumulator 2개

```c
List *merged_list = NULL;
List *acc1 = bucket[0];
List *acc2 = bucket[1];
List *acc3 = bucket[2];
int index = 0;
for(index = 3; index < BUCKET - 2; index += 3) {
    acc1 = merge_list(acc1, bucket[index]);
    acc2 = merge_list(acc2, bucket[index + 1]);
    acc3 = merge_list(acc3, bucket[index + 2]);
}

for(; index < BUCKET; index++) {
    acc1 = merge_list(acc1, bucket[index]);
}

merged_list = merge_list(acc1, acc2);
merged_list = merge_list(merged_list, acc2);
```

< Fig 56 > accumulator 3개

▶ v8 gprof test - accumulator 2개

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 30.00      0.06     0.06   838792     0.00     0.00  search_bigram
 30.00      0.12     0.06        1    60.00    60.00  sort_quick
 20.00      0.16     0.04        1    40.00    40.00  print_list
 10.00      0.18     0.02   838793     0.00     0.00  set_first
  5.00      0.19     0.01   838792     0.00     0.00  improved_hash
  5.00      0.20     0.01   306847     0.00     0.00  data_free
  0.00      0.20     0.00  2439584     0.00     0.00  swap_node
  0.00      0.20     0.00  1550811     0.00     0.00  next_node
  0.00      0.20     0.00   931958     0.00     0.00  extract_bigram
  0.00      0.20     0.00   306847     0.00     0.00  append_2_head
  0.00      0.20     0.00   306847     0.00     0.00  init_bigram
  0.00      0.20     0.00   100003     0.00     0.00  list_init
  0.00      0.20     0.00   100002     0.00     0.00  merge_list
  0.00      0.20     0.00    93166     0.00     0.00  lower2
  0.00      0.20     0.00        1     0.00    10.00  clear_list
  0.00      0.20     0.00        1     0.00    90.00  parse_bigram
```

< Fig 57 > v8-2x2 loop unrolling에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : unburdens
Back word  : with
Frequency  : 1
----------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : our
Back word  : attribute
Frequency  : 1
----------------------------


----------------------------
Running time: 494.205ms (0.494s)
----------------------------
```

< Fig 58 > v8-2x2 실행 시간

▶ v8 gprof test - accumulator 3개

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 28.57      0.06     0.06   838792     0.00     0.00  search_bigram
 23.81      0.11     0.05   838793     0.00     0.00  set_first
 23.81      0.16     0.05        1    50.00    50.00  print_list
  9.52      0.18     0.02   306847     0.00     0.00  data_free
  9.52      0.20     0.02        1    20.00    20.00  sort_quick
  4.76      0.21     0.01        1    10.00   120.00  parse_bigram
  0.00      0.21     0.00  2439584     0.00     0.00  swap_node
  0.00      0.21     0.00  1550811     0.00     0.00  next_node
  0.00      0.21     0.00   931958     0.00     0.00  extract_bigram
  0.00      0.21     0.00   838792     0.00     0.00  improved_hash
  0.00      0.21     0.00   306847     0.00     0.00  append_2_head
  0.00      0.21     0.00   306847     0.00     0.00  init_bigram
  0.00      0.21     0.00   100003     0.00     0.00  list_init
  0.00      0.21     0.00   100002     0.00     0.00  merge_list
  0.00      0.21     0.00    93166     0.00     0.00  lower2
  0.00      0.21     0.00        1     0.00    20.00  clear_list
```

< Fig 59 > v8-3x3 loop unrolling에 대한 flat profile

```
-+-+-+-[306846] Bigram-+-+-+-
Front word : unburdens
Back word  : with
Frequency  : 1
----------------------------

-+-+-+-[306847] Bigram-+-+-+-
Front word : our
Back word  : attribute
Frequency  : 1
----------------------------


----------------------------
Running time: 484.975ms (0.485s)
----------------------------
```

< Fig 60 > v8-3x3 실행 시간

위의 결과들을 확인해 봤을 때, 큰 성능의 변화는 관찰되지 않았다.

Part 3. Conclusion

Part 2에서 수행한 다양한 최적화 단계를 표로 한 번에 정리하였다.

• 최적화 단계별 특징 / 특이 사항

version	최적화 방식	실행 시간	전 과정 대비 개선 정도	특이 사항
v1	초기 버전	-	-	• 소설 전문 입력 시 프로그램 종료 x • 1,200줄 텍스트 입력 시 173.1s 소요
v2	정렬 방식 개선 Insertion -> Quick	11.48s	69,856x	• 1,200줄 텍스트에 대해서 2.48ms 소요 • 1,200줄 텍스트 기준 69,856배 성능 상승 • 소설 전문에 대한 테스트 가능 => 11.48s 소요
v3	버킷 개수 증가	≈ 6s	1.91x	• 불균일한 hashing으로 인해 성능 상승에 제한 • Bucket 개수를 늘려감에 따라 점근적으로 6초 정도의 실행시간 측정
v4	Hash 함수 개선	590.89ms	10.15x	• 균일한 hash 분포를 갖게 되면 bucket과 성능은 비례 관계 • 메모리 사용 대비 성능 상승을 고려하여 bucket의 수를 100,003으로 지정
v5	Data 의존성 줄이기	472.74ms	1.25x	• hash 함수의 data flow diagram상의 critical path를 수정하는 방식을 사용 • 실질적 최적화의 한계점으로 고려됨
v6	함수 호출 줄이기	487.74ms	0.97x	• 실행마다 측정되는 시간이 진동하는 형태를 보임 • 전체 프로그램 대비 차지하는 시간이 적어, v6~v8의 최적화가 실질적으로 유의미한지 판단하기는 어려움 -> CPE 수준의 분석이 필요 -> 개선이 되었더라도 실제 성능 개선을 체감하기엔 어려움
v7	Code motion	478.33ms	1.02x	
v8	Loop unrolling 2x2	494.21ms	0.98x	
	Loop unrolling 3x3	484.98ms	1.02x	

• 마무리

▶ 최적화를 어렵게 하는 요소

기본적으로 bigram_analyzer에서는 문자열 비교와 리스트의 정렬이 가장 핵심이고, 시간을 많이 차지하는 부분이다. 교재에서 프로그램의 초기 조건으로 가변 길이의 문자열을 저장하고, 또 검색하는 데 있어서 사용할 자료구조로 linked list를 제시하였기 때문에, liked list가 갖는 구조적 한계가 최적화를 방해하는 요소로 작용할 수밖에 없었다.

Numeric data 기반의 프로그램은 특정 레지스터에 정보를 저장하고 해당 정보 기반의 사칙연산을 수행하는 방식으로 메모리 접근을 최소화할 수 있다. 그러므로 최대한 메모리 접근을 하지 않는 형태로 최적화하기에 용이하다. 하지만, 문자열을 다루는 프로그램의 특성상 byte 단위로 메모리에 저장된 값에 접근하여 순차적으로 정보를 이용해야 한다.

search_bigram() 함수와 같이 문자열 비교를 하는 경우를 고려해보자. 두 문자열이 같다는 것을 확인하기 위해서는 첫 문자부터 끝 문자까지 모든 문자에 대해 접근해서 비교 연산을 수행해야 한다. 이 문자열 비교를 가장 최적화하는 방법은 서로 다른 문자임을 확인한 순간 비교를 멈추는 것이 최선일 것이다.

```
/* Compare S1 and S2, returning less than, equal to or
   greater than zero if S1 is lexicographically less than,
   equal to or greater than S2.  */
int
strcmp (const char *p1, const char *p2)
{
  /* Handle the unaligned bytes of p1 first.  */
  uintptr_t n = -(uintptr_t)p1 % sizeof(op_t);
  for (int i = 0; i < n; ++i)
    {
      unsigned char c1 = *p1++;
      unsigned char c2 = *p2++;
      int diff = c1 - c2;
      if (c1 == '\0' || diff != 0)
        return diff;
    }

  /* P1 is now aligned to op_t.  P2 may or may not be.  */
  const op_t *x1 = (const op_t *) p1;
  op_t w1 = *x1++;
  uintptr_t ofs = (uintptr_t) p2 % sizeof(op_t);
  return ofs == 0
    ? strcmp_aligned_loop (x1, (const op_t *)p2, w1)
    : strcmp_unaligned_loop (x1, (const op_t *)(p2 - ofs), w1, ofs);
}
```

< Fig 61 > glibc에 정의된 strcmp()

하지만, 이러한 로직은 glibc를 확인해 보면 이미 서로 다른 문자를 발견한 순간 함수를 반환하도록 구현이 되어있었다.[2] 이런 구조상에서 문자열 비교 함수 자체로는 더 이상 최적화할 여지가 없었다.

---

[2] The GNU C Library, https://sourceware.org/git/?p=glibc.git;a=blob;f=string/strcmp.c;h=11ec8bac816b630417ccbfeba70f9eab6ec37874;hb=HEAD

이제 linked list와 연계되어 각 노드에 비교할 문자열들이 존재하고 있다면, 순차적인 검색을 수행하면 최악의 경우엔 전체 노드 수만큼의 비교 횟수가 필요하다. 그리고 비교 횟수만큼 다음 노드로 접근하기 위한 메모리 access가 필요하다. 문자열의 비교와 linked list 자료구조의 특성상 최소로 필요한 메모리 접근 및 비교 연산의 횟수 자체는 최적화가 불가능하다.

그렇기에 연결 리스트의 순차적인 메모리 접근으로 인한 시간 지연을 줄이려면 비교 대상 자체를 줄여야만 했다. 문자열의 1차 분류를 위한 방식으로 hash table을 사용한 것이 가장 최적화로 적절했다고 판단된다.

하지만 hash table의 규모를 설정하는 것은 데이터의 절대적인 개수에 의존한다. 셰익스피어의 소설 하나만을 대상으로 실행되는 본 bigram analyzer의 경우 추출되는 bigram 수를 미리 파악한 뒤 그것에 최적화시킬 수 있었지만, 이것은 해당 텍스트 데이터에 과적합(Over-fitting) 된 최적화로 판단할 수 있다.

좀 더 범용적인 텍스트에 대해 효율적인 시·공간 복잡도를 구현하기 위해서는 자료의 메타 데이터로부터 적절한 hash table의 규모를 계산하는 로직이 필요할 것으로 판단하였다.

본 프로젝트의 bigram analyzer 프로그램처럼 메모리 접근이 필연적으로 많이 요구되는 환경에서는 최적화가 까다롭다는 것을 실감했고, 이런 상황에서는 어떤 방식으로 최적화를 수행할 수 있을지 더 분석할 필요성을 느꼈다.

▶ 성능 개선의 체감도

최적화 과정 중 v1 ~ v4까지는 한 요소를 개선하여 상당히 큰 성능의 변화를 확인할 수 있었다. 이들의 공통점을 확인해 보면, call graph를 분석했을 때 전체 실행 시간에서 90% 이상의 비중을 갖던 함수들을 최적화한 것이다.

한편, v5 ~ v8까지의 최적화는 call graph 상에서 5 ~ 10%를 차지하던 함수들에 대해 교과서가 제시한 전략을 적용한 것이다. amdahl의 법칙에 의하면, 5 ~ 10% 비중을 차지하는 함수(코드)의 실행 시간을 0에 가깝게 개선시켜도 10% 정도의 성능 상승만을 기대할 수 있다. 그렇기에 v5 ~ v8의 최적화는 실질적으로 개선이 되었다고 하더라도, 실제 프로그램의 실행 시간에 크게 반영되지 않았을 것으로 판단할 수 있다.

프로그램의 전체적 퍼포먼스를 상승시키려면, 상당히 많은 부분을 개선시켜야 한다는 amdahl's law의 관점을 직접 경험해볼 수 있었다.

▶ 마무리하며

지금까지 직접 만든 프로그램을 대상으로 gprof 툴을 이용하여 프로파일링 기법과 최적화에 대한 실습을 수행해 보았다. 이론으로만 학습하던 교재 5장의 내용을 실제 코드에 어떻게 적용하면 좋을지 고민하는 과정이 어렵게 다가왔지만, 정말 이롭게 느껴졌던 프로젝트였다.

이것으로 시스템 프로그램 Assignment-03 [Program Profiling & Optimization] 프로젝트 보고서를 마칩니다.