Wenlan Tian
COP 3530
Section 1087, MAEB 211
12/09/2014
Project 2V

ABSTRACT

The purpose of this project is to compare the performance of different hash functions, capacity effects, collision avoidance methods, open addressing vs randomized binary search tree. By comparing the mean system time and user time of four different hash functions, the hash_string_2 provided by the professor ran the fastest, and the hashed keys are relatively even distributed across all the slots. Thus, the hash_string_2 function was used in the other parts of the project. The second part of the project is to test the best capacity for the hash_string_2 function among all the datasets. However, it was incompleted, nor the remaining parts.

HONESTY PLEDGE

"On my honor, I have neither given nor received unauthorized aid in doing this assignment."

Wenlan Tian

LEARNING EXPERIENCE

The most difficult parts of the task to design the experiment because there are so many different factors involved.

The easiest part of the task is to implement it once the design has done, however, there are still many coding issues need to be solved.

The assignment's educational objective: test the effects of different factors for hash maps. Also be familiar with GNUPLOT.

How well I think I achieved them: I've tried my best, but I still can only finish the first part. I started the second part, but there is some bugs in the code so that I was unable to get the results. I did not even have time to start the other parts.

TABLE OF CONTENTS

INTRODUCTION

When dealing with programming, running time is an important factor we should think about. For a hash map, there are many factors could affect the speed of the program: hash function, capacity, methods for handling collisions, etc.

The purposes of this project are: 1) identify the best performing hash function for keys of type string; 2) identify the best performing heuristic for selecting a hash table capacity when using the best performing hash function for keys of type string; 3) identify the best performing collision avoidance strategy when using the best table capacity selection heuristic and the best performing hash function for keys of type string; 4) characterize the relationship between number of buckets and performance relative to the best open addressing scheme's performance; 5) compare the performance of your randomized binary search tree (BST) implementation to both the "best" performing open addressing hash map and the hash map with buckets.

The performance of hash map depends on a lot of factors, such as the hash function, capacity, collision handling methods and many others. The goal of this research will help to understand how these factors affect the performance, and thus we can choose the best one for real programming project.

OVERVIEW

The best hash function will be first determined, and the best hash table capacity will be selected based using the previous "best" hash function. After that, the best collision avoidance strategy will be determined. At last, the performance between randomized BST and open addressing map will be compared.

TEST PLATFORM AND CONDITON

The tests in this project were performed on OpenBSD 5.5, and figures were constructed using GNUPLOT which is already built in the OpenBSD 5.5.

1. HASH FUNCTION PERFORMANCE

Objective

The part is specifically to identify the best performing hash function for keys of type string.

Experimental protocol

In this part, four hash functions for type string were tested, including my own hash function in previous project and another three functions provided by the instructors.

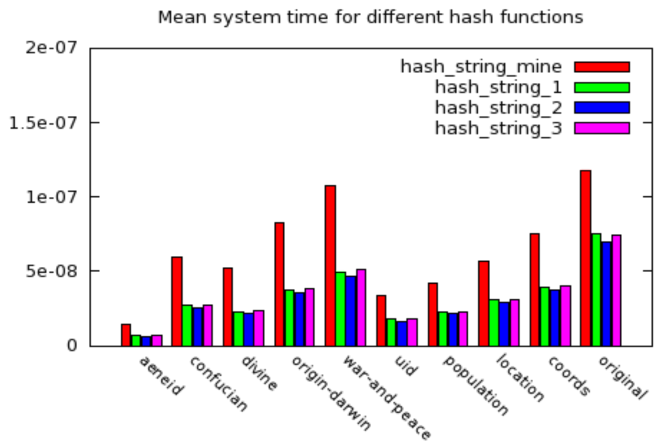
These three functions were tested individually using ten different data sets with the capacity the same as the total number of items in each data set respectively.

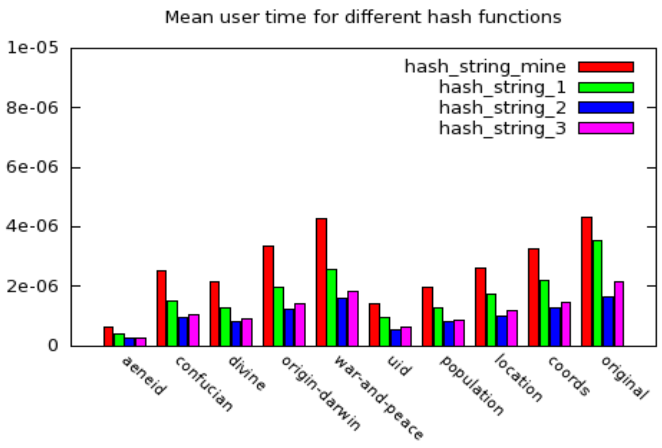
More detailed, the mean time to hash a key in dataset i using function for each {hash set + hash function} paring were tested. Thus the mean user time and system time were output from a total of 40 data set and hash function pairs. After that, the mean time to hash a key for each hash function across all the data sets were computed as well.

A total of ten data sets were used for this test, including aeneid_vergil-unique.txt, confucian-analects-unique.txt, divine-comedy_dante-unique.txt, origin-of-the-species_darwin-unique.txt, war-and-peace_tolstoi-unique.txt, localities-uid.txt, localities-population.txt, localities-location.txt, localities-coords.txt, and localities-original.txt.

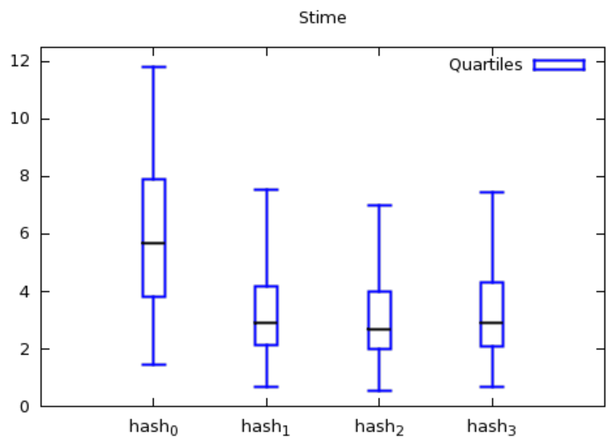
Results

For each dataset, both the mean system time and user time to hash a key in every dataset were tested when using four functions. The histogram below shows that the hash_string_2 function took the least time compared to other three functions in all the ten data sets for both user time and system time.





After that, the performance of each hash functions was tested across all the data sets. The shortest time, 1st quarter time, mean time, 3rd quarter time, and the longest time were used calculated, and a box-and-whisker wth median bar and whiskerbars graph were draw using the GNUPLO. From the graph, it also showed that hash_string_2 function has the least system time and user time overall. The deviations of the time for hash_string_2 are also the least compared to other three functions.



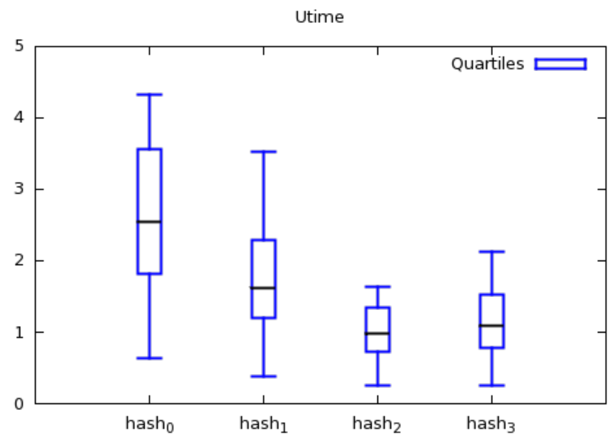
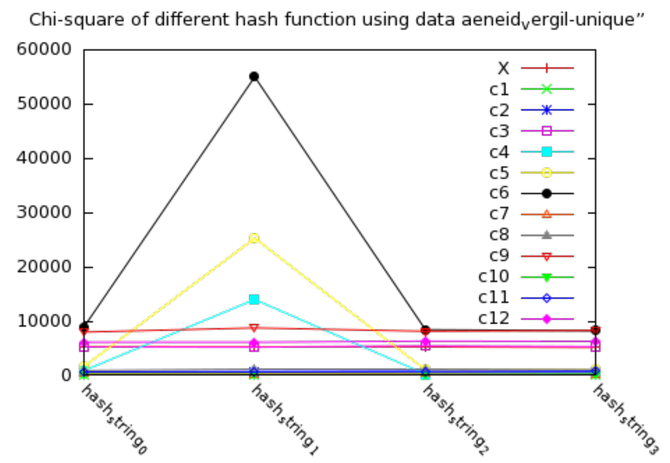
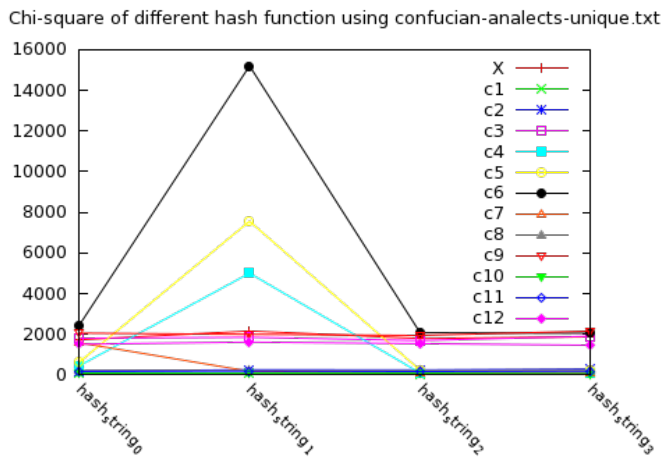


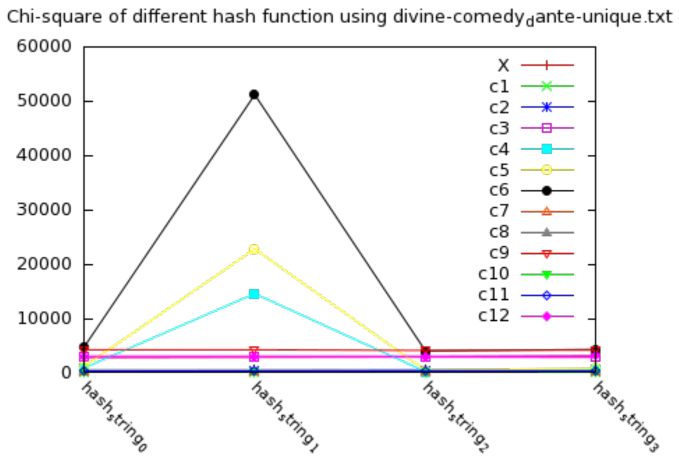
Table 1: Capacity table

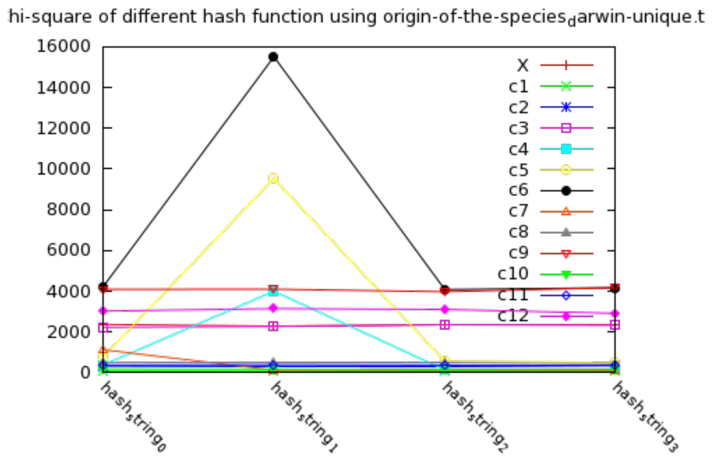
	Avu:5245			Cau:1788			Dcdu:2947			Odu:2334			Wtu:2135			Localities: 11111		
	X/23	X/7	X	X/23	X/7	X	X/23	X/7	X	X/23	X/7	X	X/23	X/7	X	X/23	X/7	X
	228.0	749.3	5245	77.7	255.4	1788	128.1	421	2947	101.5	333.4	2334	92.8	305	2135	483.1	1587.3	11111
nearest larger prime	229	751	5261	79	257	1789	131	431	2953	103	337	2339	97	307	2137	487	1597	11113
nearest larger power of two	256	1024	8192	128	256	2048	256	512	4096	128	512	4096	128	512	4096	512	2048	16384
prime nearest larger power of 2	251	1021	8191	127	251	2039	251	509	4093	127	509	4093	127	509	4093	509	2039	16381
prime in middle of nearest powers of two	193	769	6151	97	193	1543	193	389	3079	97	389	3079	97	389	3079	389	1543	12289

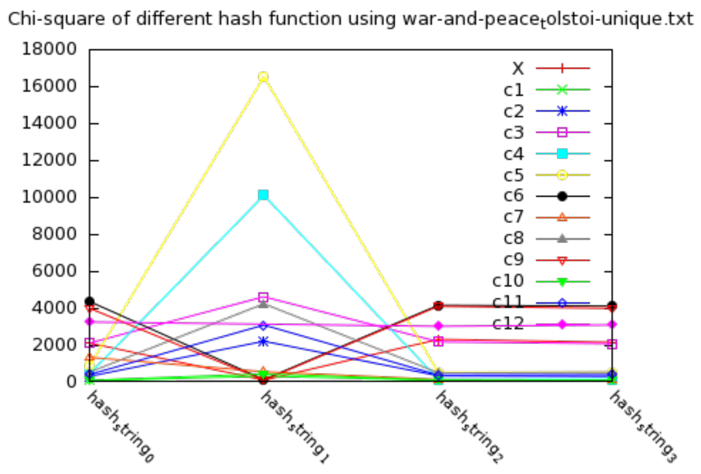
A good hash function should have the hashed keys evenly distributed across the map. Thus, for each dataset, Pearson's cumulative test statistic test was performed to evaluate the performance of the distribution of each function using 13 different capacities (Table 1). The figures below showed that, for 7 out of 10 data sets, hash_function_1 has the worst performance (highest X2) for almost all the capacities, while the other three functions have similar performance. For data set "war and peace", hash_function_1 showed different performance pattern for different capacity. For data set "localities-original", all the four functions have similar performance.

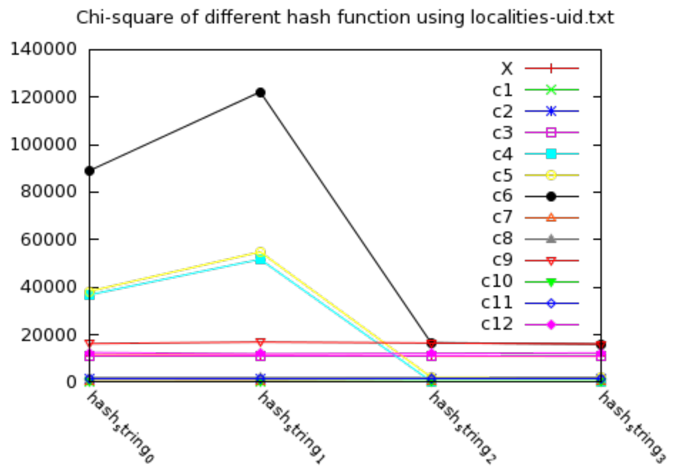


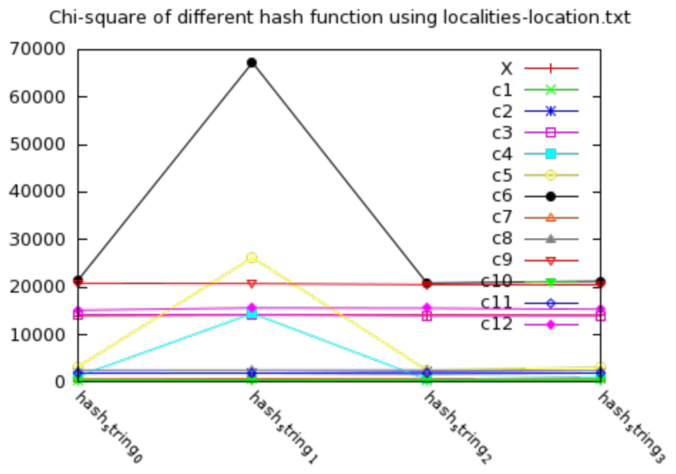


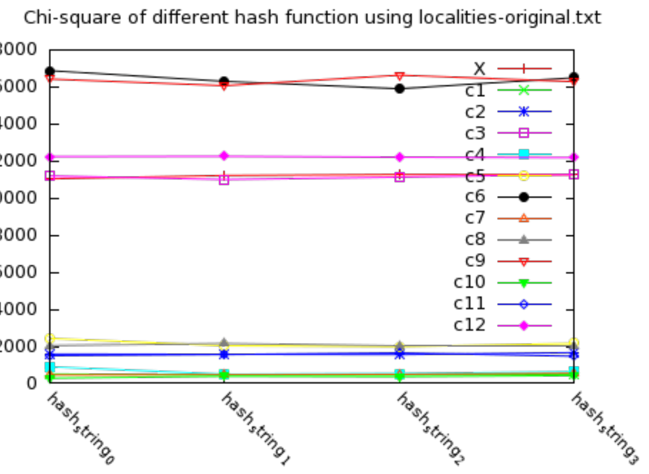


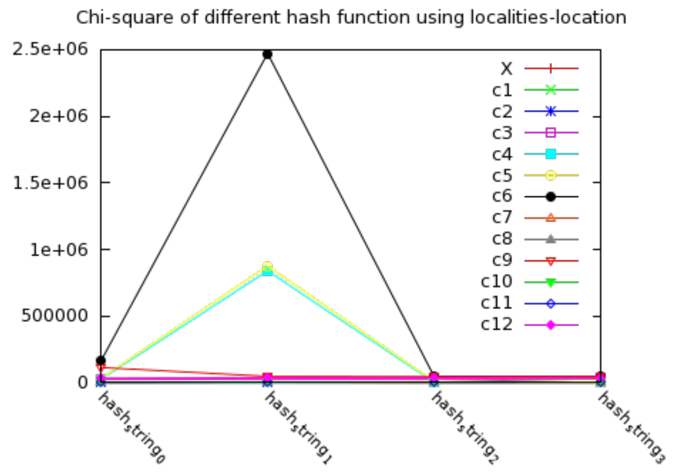


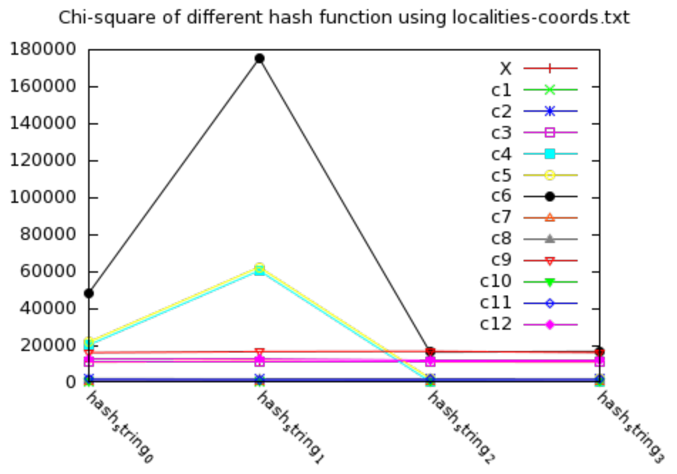

Chi-square of different hash function using war-and-peace$_t$olstoi-unique.txt
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
hash$_s$tring$_0$
hash$_s$tring$_1$
hash$_s$tring$_2$
hash$_s$tring$_3$
X
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12


Chi-square of different hash function using localities-uid.txt
140000
120000
100000
80000
60000
40000
20000
0
hash$_s$tring$_0$
hash$_s$tring$_1$
hash$_s$tring$_2$
hash$_s$tring$_3$
X
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12

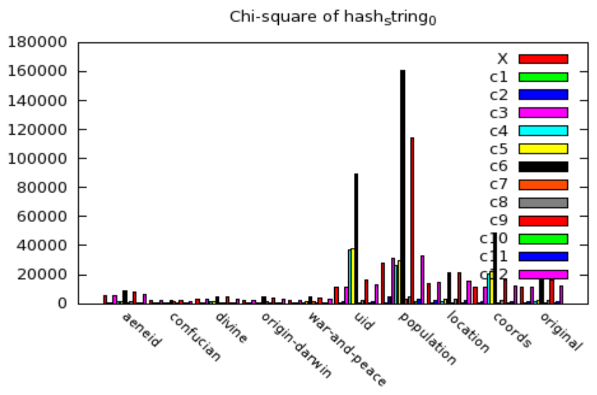

Chi-square of different hash function using localities-location.txt
70000
60000
50000
40000
30000
20000
10000
0
hash$_s$tring$_0$
hash$_s$tring$_1$
hash$_s$tring$_2$
hash$_s$tring$_3$
X
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12

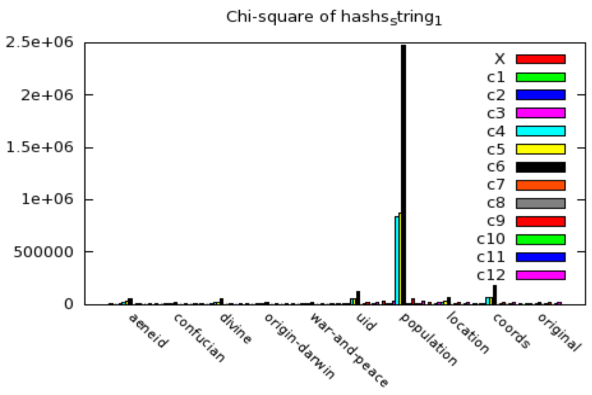

Chi-square of different hash function using localities-original.txt
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
hash$_s$tring$_0$
hash$_s$tring$_1$
hash$_s$tring$_2$
hash$_s$tring$_3$
X
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12

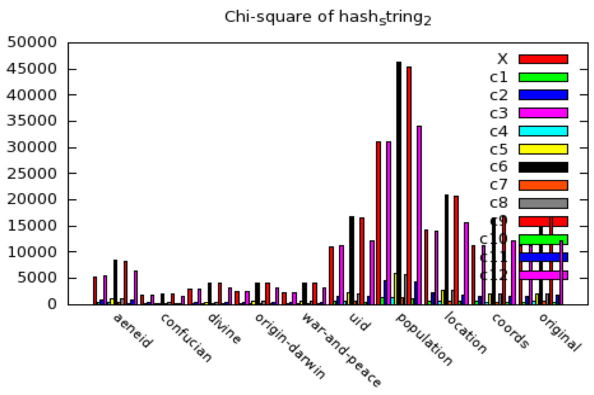

Chi-square of different hash function using localities-location
2.5e+06
2e+06
1.5e+06
1e+06
500000
0
hash$_s$tring$_0$
hash$_s$tring$_1$
hash$_s$tring$_2$
hash$_s$tring$_3$
X
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12

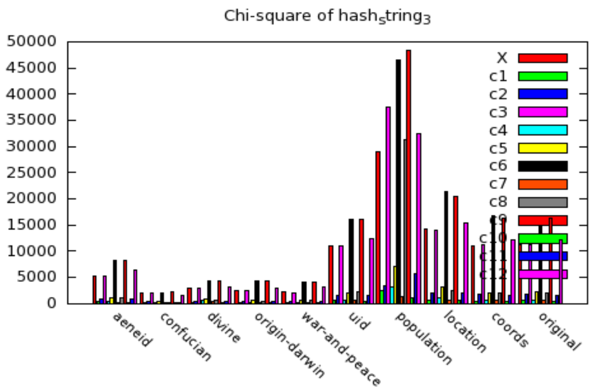

Chi-square of different hash function using localities-coords.txt
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
hash$_s$tring$_0$
hash$_s$tring$_1$
hash$_s$tring$_2$
hash$_s$tring$_3$
X
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12

For each hash function, the Chi-square test was also performed across all the ten data sets and 13 table capacities. From the four histograms shown below, the distribution of hash_string_2 and hash_string_3 are more even than hash_string_0 and hash_string_1. That means hash_string_2 and hash_string_3 will do good not depend on the data set and capacity very much, which is good for hashing.

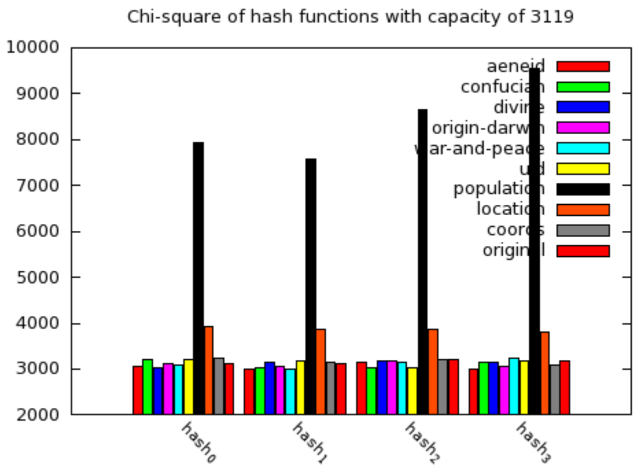








For a given capacity of 3119, the performance of all the four hash functions was tested across ten data sets. The histogram bellowed there no significant difference for the hash functions across most of the data sets, except the population data set.



Conclusions

How quickly does a particular function compute a hash for keys of type K?

As mentioned before, for each dataset, both the mean time to hash a key in every dataset were tested using four functions. The results showed that the hash_string_2 function took the least time compared to other three functions in all the ten data sets for both user time and system time. The performance of each hash functions was also tested across all the data sets, and the result showed that hash_string_2 function has the least mean time overall.

How well does a particular hash function for keys of type K, perform at randomly distributing those keys for a table capacity of n for each of the data sets?

For each dataset, Pearson's cumulative test statistic test was performed to evaluate the performance of the distribution of each function using 13 different capacities. The result showed hash_function_1 has the worst performance across the most of the data sets while the other three functions have the similar performance.

For data sets with keys of type K, which hash function does the best over-all job?

For each hash function, the Chi-square test was performed across all the ten data sets and 13 table capacities. The result showed the distribution of hash_string_2 and hash_string_3 are more even than hash_string_0 and hash_string_1. That means hash_string_2 and hash_string_3 will do good not depend on the data set and capacity very much, which is good for hashing.

For data sets with keys of type K, which hash function does the best over-all job for a table capacity of 3119?

With a given capacity of 3119, the performance of all the four hash functions was tested across ten data sets. The results showed there no significant difference for the hash functions across most of the data sets, except the population data set.

Based on the all above test, hash_string_2 has the most performance overall, which took least time for hashing, and showed even distribution for the hashed keys.

2. OPEN ADDRESSING HASH MAP PERFORMANCE: CHOOSING A MAXIMUM CAPACITY

Objective

This part is specifically to identify the best performing heuristic for selecting a hash table capacity when using the best performing hash function for keys of type string.

Experimental protocol

Five kinds of capacity will be tested based on the original unique number of items in the data set. These include: (M is 0.95 * Number of unique items)

1) M + aM, where $0 \leq a \leq 1.0$

2) N, where N, is the "closest" (≥) prime to (M + aM), where $0 \leq a \leq 1.0$

3) N, where N is the closest, power of two ≥ M
4) N, where N is the prime that is “closest” to the closest power of two ≥ than M
5) N, where *N* is a prime that is ≥ *M* and “closest” to being half-way between the nearest powers of two bounding *M* above and below.

Depending on different value of a, the first two type of capacity will vary, but the other three are the same (Table 2).

Table 2: capacity table

	No.unique lexeme	M = 0.95* No.unique lexeme	*M* + a *M*	Closest (≥) prime to (*M* + a*M*)	Closest power of two ≥ *M*	Prime “closest” to the closest power of two ≥ than *M*	Prime ≥ *M* and “closest” to being half-way between the nearest powers of two bounding *M* above and below
Avu	5245	4983			8192	8191	6151
War peace	2135	2028			2048	2039	1543
UID	11111	10555			16384	16381	12289
Coords	11105	10550			16384	16381	12289
Entries	11111	10555			16384	16381	12289

A total of five data sets were used for this part, which including the aeneid set of data sets, war and peasce set of data sets, geographical UID set of data sets, geographical cords set of data sets and geographical entries set of data sets (Table 3)

Table 3: input data for part 2

Aeneid	aeneid_vergil-unique.txt
	avu-1.txt
	avu-2.txt
	avu-3.txt
	avu-4.txt
	avu-5.txt
	aeneid_vergil.txt
War and Peace	war_and_peace-unique.txt
	wpu-1.txt
	wpu-2.txt
	wpu-3.txt
	wpu-4.txt
	wpu-5.txt

	war_and_peace.txt
Geographical UID	uid-unique.txt
	uid-1.txt
	uid-2.txt
	uid-3.txt
	uid-4.txt
	uid-5.txt
	localities-uid.txt
Geographical coords	coords-unique.txt
	coords-1.txt
	coords-2.txt
	coords-3.txt
	coords-4.txt
	coords-5.txt
	localities-original.txt
Geographical entries	localities-uid.txt
	localities-location.txt
	localities-country.txt
	localities-population.txt
	localities-coords.txt
	original-1.txt
	original-2.txt

Results

Conclusions

For a given load factor, how does each table size choice perform for each of the data sets?

For a given load factor, how does each table size choice perform overall for data sets using keys of type *K*?

For data sets with keys of type *K*, which table size choice has the best over all *balance between performance and memory required* (assuming the table's size will never exceed M)?

3. Open addressing hash map performance: collision handling strategies
The part is specifically to identify the best performing collision avoidance strategy when using the best table capacity selection heuristic and the best performing hash function for keys of type string.

4. Hash map performance: buckets vs open addressing

The part is specifically to characterize the relationship between number of buckets and performance relative to the best open addressing scheme's performance

5. Hash map vs randomized BSTs
The part is specifically to identify the "best" performing collision avoidance strategy when using the "best" table capacity selection heuristic and the "best" performing hash function for keys of type *K*.

3. OPEN ADDRESSING HASH MAP PERFORMANCE: COLLISION HANDLING STRATEGIES

Objective

This part is specifically to identify the best performing heuristic for selecting a hash table capacity when using the best performing hash function for keys of type string.

Experimental protocol

Results

Conclusions

4. HASH MAP PERFORMANCE: BUCKETS VS. OPEN ADDRESSING

Objective

This part is specifically to characterize the relationship between number of buckets and performance relative to the "best" open addressing scheme's performance.

Experimental protocol

Results

Conclusions

5. HASH MAP VS. RANDOMIZED BSTs

Objective

This part is specifically to compare the performance of your randomized binary search tree implementation to both the "best" performing open addressing hash map and the hash map with buckets.

Experimental protocol

Results

Conclusions

CONCLUSION

Only the first part of the project is completely done which shows the hash_string_2 function provided by the professor had the best overall performance among different data sets and capacity.

APPENDICES

A. examples source code for each trial type

```
////////////////code to calculate mean system time and user time
#include <sys/resource.h>
#include <iostream>
#include <fstream>
#include <math.h>

int capacity;

std::size_t hash(std::string key){
        int h = 0;
        int a = 127;
        for (std::size_t i =0; i < key.length(); i++){
                h = ( a * h + (int)key[i]);
        }
        return h;
}

void readfile(std::string fileName){
        std::fstream in(fileName.data());
        std::string item;
        std::ifstream inStream;
        std::string tempString;

        inStream.open(fileName.c_str(), std::ifstream::in);

        if(!inStream.is_open()) {
                std::cout << "File " << fileName << " cannot be opened!" << std::endl;
                return;
        }

        while (inStream >> tempString) {
                int index = hash(tempString) % capacity;
        }
        in.close();
}

int main(){

        int capacity_avu[13] = {5245, 229, 751, 5261, 256, 1024, 8192, 251, 1021, 8191,
193, 769, 6151};
        int capacity_cau[13] = {1788, 79, 257, 1789, 128, 256, 2048, 127, 251, 2039, 97,
193, 1543};
        int capacity_dcdu[13] = {2947, 131, 431, 2953, 256, 512, 4096, 251, 509, 4093,
193, 389, 3079};
        int capacity_odu[13] = {2334, 103, 337, 2339, 128, 512, 4096, 127, 509, 4093,
193, 389, 3079};
        int capacity_wtu[13] = {2135, 97, 307, 2137, 128, 512, 4096, 127, 509, 4093, 97,
389, 3079};
        int capacity_localities[13] = {11111, 487, 1597, 11113, 512, 2048, 16384, 509,
2039, 16381, 389, 1543, 12289};

        std::string fileName[10] = {"aeneid_vergil-unique.txt","confucian-analects-
unique.txt", "divine-comedy_dante-unique.txt", "origin-of-the-species_darwin-
unique.txt", "war-and-peace_tolstoi-unique.txt", "localities-uid.txt", "localities-
population.txt", "localities-location.txt","localities-coords.txt", "localities-original.txt"};

        int num_key[10] = {capacity_avu[0], capacity_cau[0], capacity_dcdu[0],
capacity_odu[0],capacity_wtu[0],capacity_localities[0],capacity_localities[0],capacity_lo
calities[0],capacity_localities[0],capacity_localities[0]};

        // print to file
        std::ofstream output;
        std::string outFileName = "hash_function.txt";

        output.open(outFileName.c_str());
        if (output.fail()){
                std::cerr << "File " << outFileName << " cannot be opened" << std::endl;
                return 1;
        }

        for (int i = 0; i < 10; i++){

                output << "====== Data: " << fileName[i] << " ======"<< std::endl;
```

```
int current_capacity[13];

if (fileName[i]=="aeneid_vergil-unique.txt"){
        for (int m = 0; m < 13; ++m){
                current_capacity[m] =  capacity_avu[m];
        }
}else if(fileName[i]=="confucian-analects-unique.txt"){
        for (int m = 0; m < 13; ++m){
                current_capacity[m] =  capacity_cau[m];
        }
}else if(fileName[i]=="divine-comedy_dante-unique.txt"){
        for (int m = 0; m < 13; ++m){
                current_capacity[m] =  capacity_dcdu[m];
        }
}else if(fileName[i]=="origin-of-the-species_darwin-unique.txt"){
        for (int m = 0; m < 13; ++m){
                current_capacity[m] =  capacity_odu[m];
        }
}else if(fileName[i]=="war-and-peace_tolstoi-unique.txt"){
        for (int m = 0; m < 13; ++m){
                current_capacity[m] =  capacity_wtu[m];
        }
}else {
        for (int m = 0; m < 13; ++m){
                current_capacity[m] =  capacity_localities[m];
        }
}

capacity = current_capacity[0];

for (int k =0; k< 100000; ++k){readfile(fileName[i]);}

struct rusage r;
struct timeval utime, stime;
float mean_utime, mean_stime, total_utime, total_stime;
getrusage(0, &r);
utime = r.ru_utime;
stime = r.ru_stime;

total_utime = utime.tv_sec + 1.0*utime.tv_usec/1000000;
total_stime = stime.tv_sec + 1.0*stime.tv_usec/1000000;

mean_utime = (utime.tv_sec+
1.0*utime.tv_usec/1000000)/num_key[i]/100000;
mean_stime = (stime.tv_sec +
1.0*stime.tv_usec/1000000)/num_key[i]/100000;
```

```
                output << "User time: " << total_utime << " seconds" << std::endl;
                output << "System time: " << total_stime << " seconds" << std::endl;
                output  << "Mean user time: " << mean_utime << " seconds" << std::endl;
                output  << "Mean system time: " << mean_stime << " seconds" <<
std::endl;

        }
        return 0;
}

/////code part2- testing only
int closePrime(int n){
        std::string fileName = "prime.txt";
        std::fstream in(fileName.data());
        std::ifstream inStream;
        int temp;

        inStream.open(fileName.c_str(), std::ifstream::in);

        if(!inStream.is_open()) {
                std::cout << "File " << fileName << " cannot be opened!" << std::endl;
           return 0;
        }

        int i = 0;

        while(inStream >> temp){
                if (temp > n){return temp;}
        }

        in.close();
        return 0;
}

int main(){
        HMOA hmoa;

        std::string data_sets[5]={"aeneid","war_peace","UID","coords","entries"};
        std::string data_aeneid[7] = {"aeneid_vergil-unique.txt","avu-1.txt","avu-
2.txt","avu-3.txt","avu-4.txt","avu-5.txt","aeneid_vergil.txt"};
        std::string data_war_peace[7] = {"war-and-peace_tolstoi-unique.txt","wtu-
1.txt","wtu-2.txt","wtu-3.txt","wtu-4.txt","wtu-5.txt","war-and-peace_tolstoi.txt"};
        std::string data_UID[7] = {"localities-uid-unique.txt","uid-1.txt","uid-2.txt","uid-
3.txt","uid-4.txt","uid-5.txt","localities-uid.txt"};
```

```
	std::string data_coords[7] = {"localities-coords-unique.txt","coords-
1.txt","coords-2.txt","coords-3.txt","coords-4.txt","coords-5.txt","localities-coords.txt"};
	std::string data_entries[8] = {"localities-coords.txt","localities-
country.txt","localities-location.txt","localities-original.txt","localities-
population.txt","localities-uid.txt","original-1.txt","original-2.txt"};

	int capacity_avu[5] = {4983, 0, 8192, 8191, 6151};
	int capacity_wp[5] = {2028, 0, 2048, 2039, 1543};
	int capacity_uid[5] = {10555, 0, 16384, 16381, 12289};
	int capacity_coords[5] = {10550, 0, 16384, 16381, 12289};
	int capacity_entries[5] = {10555, 0, 16384, 16381, 12289};

	std::ofstream output;
	std::string outFileName = "linear.txt";

	output.open(outFileName.c_str());
	if (output.fail()){
		std::cerr << "Output file cannot be opened" << std::endl;
		return 1;
	}

	//for different a
	for (double i = 0; i <1.05; i+=0.05){
		capacity_avu[0]= 4983+4983*i;
		capacity_avu[1]=closePrime(capacity_avu[0]);

		capacity_wp[0]= 4983+4983*i;
		capacity_wp[1]=closePrime(capacity_wp[0]);

		capacity_uid[0]= 4983+4983*i;
		capacity_uid[1]=closePrime(capacity_uid[0]);

		capacity_coords[0]= 4983+4983*i;
		capacity_coords[1]=closePrime(capacity_coords[0]);

		capacity_entries[0]= 4983+4983*i;
		capacity_entries[1]=closePrime(capacity_entries[0]);

		double load_factor = 0;

		//for each load factor from 0.03 to 0.99 by 0.03
		for (double j = 0.03; j < 0.09; j+=0.03){
			load_factor = j;

			//for each set of data sets  [e.g., The Aeneid set of data sets]
			for (int k = 0; k < 5;++k){
```

```
int mySize = 0;

if (k==4){mySize = 8;}else{mySize = 7;}

std::string current_data[8];
int current_capacity[5];

if (data_sets[k]=="aeneid"){
        for(int m =0; m < 7; ++m){
                current_data[m]=data_aeneid[m];
        }
        for(int m =0; m < 5; ++m){
                current_capacity[m]=capacity_avu[m];
        }
}else if (data_sets[k]=="war_peace"){
        for(int m =0; m < 7; ++m){
                current_data[m]=data_war_peace[m];
        }
        for(int m =0; m < 5; ++m){
                current_capacity[m]=capacity_wp[m];
        }

}else if (data_sets[k]=="UID"){
        for(int m =0; m < 7; ++m){
                current_data[m]=data_UID[m];
        }
        for(int m =0; m < 5; ++m){
                current_capacity[m]=capacity_uid[m];
        }

}else if (data_sets[k]=="coords"){
        for(int m =0; m < 7; ++m){
                current_data[m]=data_coords[m];
        }

        for(int m =0; m < 5; ++m){
                current_capacity[m]=capacity_coords[m];
        }

}else if (data_sets[k]=="entries"){
        for(int m =0; m < 8; ++m){
                current_data[m]=data_entries[m];
        }

        for(int m =0; m < 5; ++m){
                current_capacity[m]=capacity_entries[m];
```

```
                    }
                }

                //for each data set in that set of data sets
                for (int l =0; l<mySize; ++l){
                    std::string fileName = current_data[l];

                    std::fstream in(fileName.data());
                    std::ifstream inStream;
                    std::string tempString;

                    inStream.open(fileName.c_str(), std::ifstream::in);

                    if(!inStream.is_open()) {
                        std::cout << "File " << fileName << " cannot
be opened!" << std::endl;
                        return 1;
                     }

                    int o = 0;

                    while(inStream >> tempString){
                        ++o;
                        while (hmoa.load()  < load_factor){
                            hmoa.insert(tempString, i);
                        }

                        hmoa.remove_random();
                        hmoa.insert(tempString, o);
                    }

                    in.close();

                    for (int n = 0; n < mySize; ++n){
                        if(n == l or n == mySize-1) continue;

                        std::string fileName1 = current_data[n];

                        std::fstream in(fileName1.data());
                        std::ifstream inStream1;
                        std::string tempString1;

                        int p = 0;
                        int success = 0;
                        int failure = 0;
                        int search_result = 0;
```

```
                                                        while(inStream1 >> tempString1){
                                                                ++p;
                                                                search_result =
hmoa.search(tempString1, p);
                                                                if (search_result > 0){++success;}
                                                                else {++failure;}
                                                        }
                                                        in.close();

                                                        output << "capacity" << std::endl;
                                                        output << "table size: " << std::endl;
                                                        output<< "number_trial: " << o << std::endl;
                                                        output<<"number_unique:"<<p <<std::endl;
                                                        output << " probe count: " << std::endl;
                                                        output<< "success:" << success <<std::endl;
                                                        output<< "failure: " << failure  << std::endl;
                                                }
                                        }
                                }
                        }
                }
                return 0;
}
```

B. gnuplot scripts (or commands) used for each graph style

```
#histogram
set terminal pngcairo  transparent enhanced font "arial,10" fontscale 1.0 size 500, 350
set output 'histogram_stime.png'
set boxwidth 0.9 absolute
set style fill   solid 1.00 border lt -1
set key inside right top vertical Right noreverse noenhanced autotitles nobox
set style histogram clustered gap 1 title  offset character 0, 0, 0
set datafile missing '-'
set style data histograms
set xtics border in scale 0,0 nomirror rotate by -45  offset character 0, 0, 0 autojustify
set xtics  norangelimit font ",8"
set xtics   ()
set title "Mean system time for different hash functions"
set yrange [ 0.00000 : 0.0000002 ] noreverse nowriteback
i = 22
plot 'hash_function_stime.dat' using 2:xtic(1) ti col, '' u 3 ti col, '' u 4 ti col, '' u 5 ti col


#box-and-whisker
set terminal pngcairo  transparent enhanced font "arial,10" fontscale 1.0 size 500, 350
set output 'box_stime_test.png'
set boxwidth 0.2 absolute
set title "box-and-whisker with median bar and whiskerbars"
set xrange [ 0 : 5]
set yrange [ 0 : 12.5]
plot 'box_stime_test2.dat' using 1:3:2:6:5:xticlabels(7) with candlesticks lt 3 lw 2 title
'Quartiles' whiskerbars,      ''                using 1:4:4:4:4 with candlesticks lt -1 lw 2 notitle


#Chi-square histogram
set terminal pngcairo  transparent enhanced font "arial,10" fontscale 1.0 size 500, 350
set output 'histogram_Chi_avu.png'
set key inside right top vertical Right noreverse noenhanced autotitles nobox
set datafile missing '-'
set style data linespoints
set xtics border in scale 1,0.5 nomirror rotate by -45  offset character 0, 0, 0 autojustify
set xtics  norangelimit font ",8"
set xtics   ()
set title "Chi-square of different hash function using data aeneid_vergil-unique”
i = 22
plot 'hash_Chi_avu.dat' using 2:xtic(1) title columnheader(2), for [i=3:14] '' using i title
columnheader(i)
```