Hashing and Bloom filters key lookup comparison

Josep Maria Olivé, Pol Monroig, Yaiza Cano

March 25, 2020

1 Introduction

This project is mainly aimed at learning different implementations of hashing based data structures and their effectiveness in applying a dictionary search problem.

A dictionary is used to maintain a set under insertion and deletion of elements while membership queries provide access to the data. The most efficient dictionaries, in theory, and practice, are based on hashing techniques.

To achieve different results and build a complete analysis, we have proposed searching with two types of dictionary data structures: hash tables and bloom filters.

The main performance parameters that we are going to study, analyze and contrast along this research work are: lookup time, build time, average probes and false positives. All the algorithms have been implemented in c++ and the same data has been applied to the execution of different versions of the same experiment.

2 Experiment pipeline and methodology

The experimentation process was a very tedious task since it required to take into account a lot of parameters. An experiment is defined by the series of parameters of section 4, thus they will no be explained here. To create the experiments we first defined parameters and we repeated the experiment for each type of dictionary, finally, we repeated the same experiment with different seeds and average the results to get a more precise result. When an experiment was done, we increased the load factor by 10% and started the experiment again. The project was implemented mainly in C++, with the exception of the plot creation that was made with Python. We used C++ because we wanted the best performance when performing each experiment, on the other hand, we used Python because thanks to its simplicity we could create the plots faster, and they are independent of the experimentation.

2.1 Open Addressing

Open addressing is one of the two main methods of collision resolution in hash tables. The four main techniques are:

- **Linear probing:** its principle is accomplished using two values: a start index and a stepping value. When a collision occurs, the table is searched sequentially for an empty slot adding repeatedly the second value to the first until either a free space is found or the entire table is traversed.
- **Quadratic probing:** its principle is accomplished using four values: a start index, a stepping value, and two constant values. When a collision occurs, an arbitrary quadratic polynomial value -made by the combination of all the values but the start index- is added to the first one; the finish criteria is the same as the previous technique. The idea is to skip regions in the table with possible clusters.
- **Double hashing:** its principle is accomplished using two values: a start index, a stepping value, and a third value. When a collision occurs, the combination of the stepping and the third value is added to the first one; the finish criteria is the same as the first technique. The idea is to tackle effectively clustering problems.
- **Cuckoo hashing:** its principle is accomplished using two values and either position is computed by the first or the second. When a collision occurs, the key in the position is replaced and the replaced key is assigned to the position given by the other value. If this new position is occupied, the procedure is repeated until the key is inserted or the iteration reaches an arbitrary value. The replacement behavior may result in an infinite loop.

Technique	Function
LP	h(k,i) = (h(k) + i) mod m
QP	h(k,i) = (h(k) + c_1i + $c_2 i^2$) mod m
DH	h(k,i) = (h_1(k) + i*h_2(k)) mod m
QH	h(k) = h_1(k) or h_2(k)

Parameters	Meaning
k	key, value to insert or to search
i	stepping value
c_1, c_2	constant values
h(),h_1(),h_2()	hash functions

Technique	Success theoretical behaviour	Fail theoretical behaviour
LP	$0.5 * (1 + \frac{1}{(1-\alpha)}))$	$0.5 * (1 + \frac{1.0}{(1-\alpha)} * (1 - \alpha))))$
DH	$\frac{1}{\alpha} * \frac{\frac{1.0}{(1-\alpha)}}{ln(\frac{1.0}{(1-\alpha)}}$	$\frac{1}{(1-\alpha)}$

2.2 Separate Chaining

Separate Chaining uses additional data structures inside the hash table, like linked lists or vectors, for collision resolution in each position. One of the main benefits of Separate Chaining is the permanent ability to insert values without increasing the table size. On the other side, using this technique can result in drastic performance penalties, when the hash table degenerates to its internal data structures because of all the collisions occurring in the same position. This also increases the cache miss rates notably.

- **Move to front:** this technique consists of the usage of linked lists in each position of the table to store all the keys with the same hash value that has been inserted. New inserted keys are appended at the end of the list, while whenever a key is accessed, it is moved to the beginning of the list. This way, recently used keys are always the ones looked up before, increasing performance. Most used keys tend to stay at the beginning of the list, while rarely searched ones will remain at the end.
- **Exact fit:** this technique uses vectors or arrays instead of linked lists, to exploit even further memory locality. This way, keys on the same position of the table are also stored in contiguous memory space, rather than dynamically allocated like before, resulting in fewer cache misses and reducing lookup time. On the other hand, using vectors and arrays introduces a previously nonexistent cost, the cost of increasing the size or moving the entire vector or array whenever a new value is inserted and there isn't enough space to store it.

Success theoretical behaviour	Fail theoretical behaviour
$1 + \frac{\alpha^2}{2}$	$1 + \frac{\alpha}{2}$

2.3 Bloom filters

Bloom filters [1] are a space-efficient probabilistic data structure that can represent a dictionary. These data structures are space-efficient because unlike hash tables, they don't store the key itself but a series of bits that determine if the value is in the dictionary or not. These bits are represented in an array or table. To insert a value into a bloom filter, it has to be hashed with k different hash functions, each hash returns a specific position of the bits table that will be marked with a 1. Thus, if a value is in the dictionary it means that if we hash the key to be queried k times every position will be marked with a 1 bit. Otherwise, if a bit is marked with a 0 it means that the value is not in the dictionary. This means that if considered k is a constant we can query and insert always in constant time. On the other hand, this causes some problems because it is possible that two keys k_1, k_2 have hash to the same k bits, and because we don't have a collision mechanism we would introduce a false positive rate when querying a specific value in the dictionary. This rate is directly dependent on the number of hash functions that we use and the load factor of the table. To minimize [2] this false positive rate different methods have been found although you can never make it disappear. Another problem that we encountered when developing bloom filters was that we needed a way to generate k different hash functions, although we found a very efficient and simple solution [4] based on creating a new hash function with two initial ones and an iterator.

$$g(Key)_i = h_1(Key) + i * h_2(Key) \tag{1}$$

$$Theoreticalvalue = (1 - \frac{1}{e^{k*\alpha}})^2 \tag{2}$$

3 Data Generation

The data used in each of the experiments consisted of a sequence of unsigned integers, the data was written in two different files. One file contained the keys to

insert into the dictionary and the other the text to find in the dictionary.

Linear Congruential Method

The randomness of the keys was based on the Linear congruential method (LCM) [5]. We needed to have experiments as reliable as possible thus we needed a way to control the data generation, that is why we used this method. The LCM is one of the oldest and best-known random number generators, it is also very easy to implement. This method generates a cyclic sequence of numbers based on an initial seed. Given a seed X_0 it is possible to generate the next number in the sequence

$$X_i = (X_{i-1} * a + c) \mod m \tag{3}$$

where a is called the multiplier, c the incrementer and m the modulus. The size of the sequence depends on the parameters used, independently on the initial seed. We used a combination of parameters that yield the maximum size sequence, that is parameters that yield m numbers. This ensured us that that there where never repeated numbers in either of the two files since the size of m was sufficiently big.

4 Experiment parameters

An experiment is purely defined by the subject, represented by the dictionary to test, and the parameters in each the dictionary were specified and tested.

- **Size of n:** The keys file contained n numbers and the text contained $2*n+p*n$ where p is the percentage of keys that are already inserted in the dictionary. We had a lot of discussions in which n used and we tried different combinations but they yielded very similar results so we finally choose a $n = 10e^7$. That happens to be 10 million keys that were inserted into the dictionary, although the table size was much bigger. We tried to use a bigger size of n but we encountered the size of the files we created where extraordinary big, we tried to divide the files but that wasn't of much use since we stilled needed to save all the numbers in the hash tables. When we tried to this the Linux oom-killer killed to process because of the out of memory that was caused. Nevertheless other researches [3] used similar sizes of n so this size wasn't really an issue.

- **Number of repetitions Q:** Each experiment was defined by a series of parameters to get the best results and avoid overfitting of the data to a single sample we repeated each experiment with $Q = 100$ that is we performed the same experiment 100 times and averaged the results.

- **Load factor α:** The load factor is one of the variables that help to compare each type of dictionary implementation, we used in total 9 different load factors ranging from 0.10 to 0.9 with an offset of 0.10. We could have subdivided the interval with a lower offset but after experimenting we found that there wasn't a substantial difference.

- **Key percentage p:** The key percentage represents the percentage of keys that are in the keys (inserted in the dictionary) that are in the search file (the second file where the lookups are done). This variable did not have a big

impact on the results since we average the lookup time of each key individually and even if we chose a really low key percentage we had a big enough Q that it didn't matter. Thus we chose $p = 0.5$;

- **Number of hash functions k:** This was a difficult parameter to choose since it had to be chosen optimally because it had a very big k then the number of false positives might diminish, but only if the size of the table was big enough, on the other hand, if we chose a small value of k the table would fill more slowly but it was more probable to find to keys with the same bits mask. To decide into what to choose we simply tested different values of k with different results.

- **Hash functions:** The hash functions we chose were not the purpose of this experiment although we tried to choose.

- **Random seed:** The seed number is a determining factor into how the dictionaries will perform since different seeds yield to different keys and consequently different hash outputs. That is why for each experiment we chose the same initial seed, although the next seed was chosen at random based on the initial seed, this enabled reproducibility. In total, a number of Q different seeds were selected, because of the results of the Q repetitions where average this enabled us to remove any outliers that could happen to choose a specific seed value.

5 Results

All the results that were obtained are specifically compared to the load factor of the table since it is a determining parameter in the efficiency of each dictionary.

Speed performance

To compare the behavior of each technique the main parameter computed is the run time.

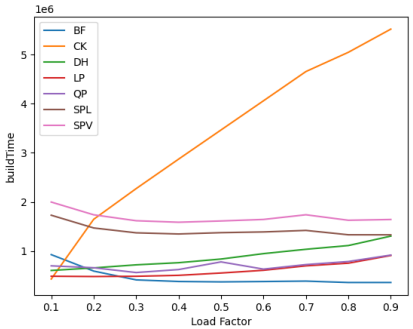

Figure 1: Build time

As we can see above, the *Cuckoo hashing* is, by far, the slowest algorithm due to its insertion method which can end in cycles; the higher the load factor, the longer the cycles get thus it takes longer to insert a key.

Both *Separate chaining* algorithms are the following ones which take longer because when there is an insertion, they may have issues with the memory. The one that performs with a vector may have to relocate all the elements in a given position. The other one just doesn't have the spatial locality principle.

On the other hand, the remaining *Open addressing* methods are faster but there is an increment of time as of the load factor.

Finally, *Bloom filters* show a little variation but stay almost constant given its implementation.

From the figures below, we can observe the behaviour performed in our experiments but we can't extract any significant conclusion regarding the algorithm's implementation. So, we speculate that they are due to the status of the computer.

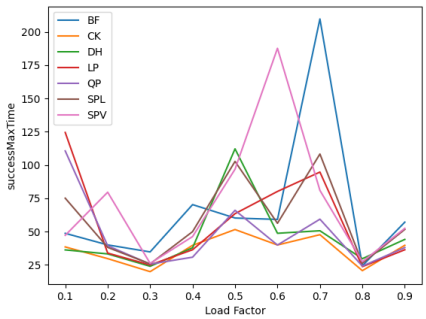

Figure 2: Successful max time

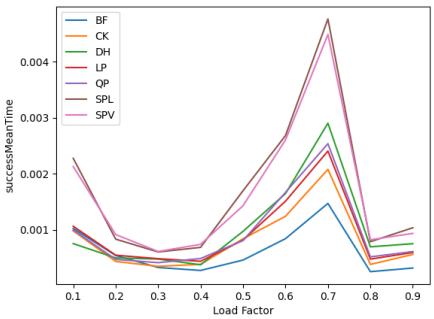

Figure 3: Successful mean time

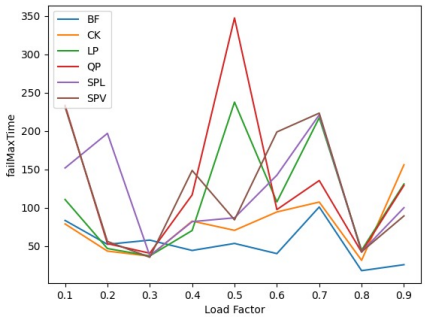

Figure 4: Fail max time

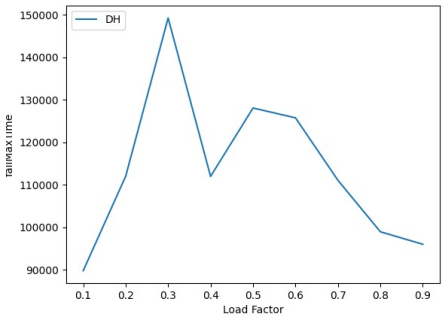

Figure 5: Fail max time

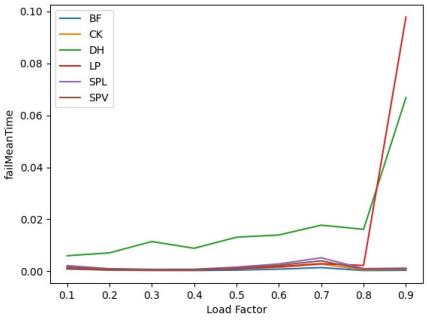

Figure 6: Fail mean time

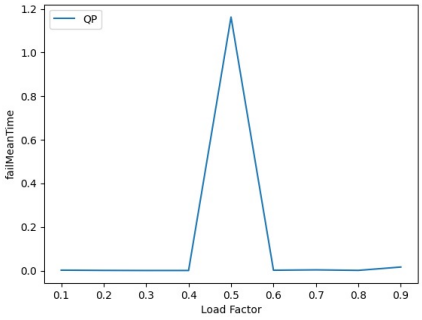

Figure 7: Fail mean time

Probes metric

In this section, we compare the average probes that were introduced in each search query regarding the load factor between the experimental values and the theoretical ones.

Open Addressing : Linear Probing

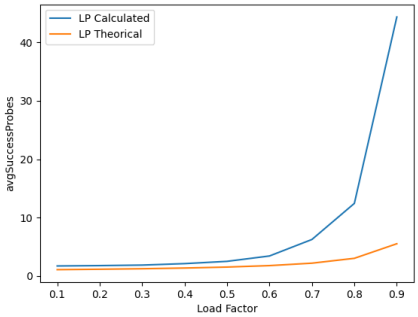

Figure 8: Successful probes

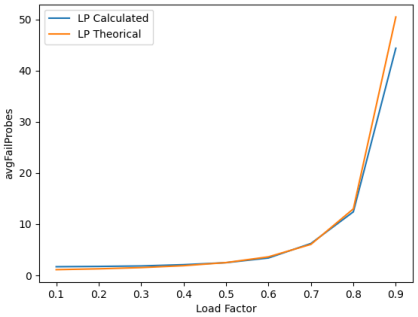

Figure 9: Fail probes

Linear Probing has similar behaviour in successful queries regarding the theory until $\alpha = 0.6$, then it starts losing the effectively until $\alpha = 0.8$ when the value soars even though we think 30 probes are not that much. Concerning the behavior in fail queries, it has almost the same performance that the theoretical one.

Open Addressing : Quadratic Probing

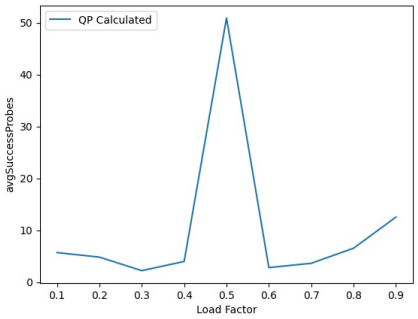

Figure 10: Successful probes

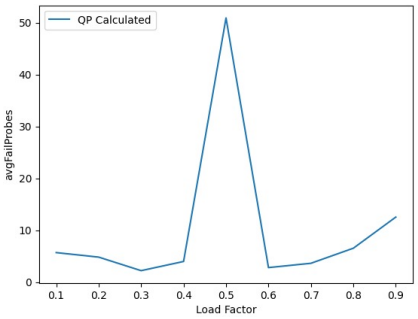

Figure 11: Fail probes

From Quadratic Probing we did not find the theoretical behaviour so we can just comment that the performance was done by both successful and fail queries is identical.

Open Addressing : Double Hashing

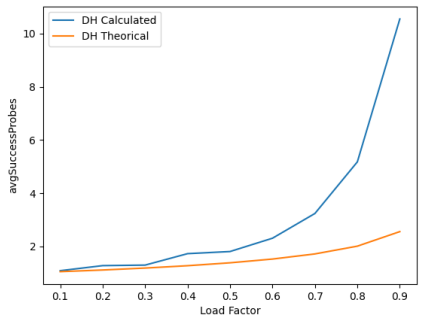

Figure 12: Successful probes

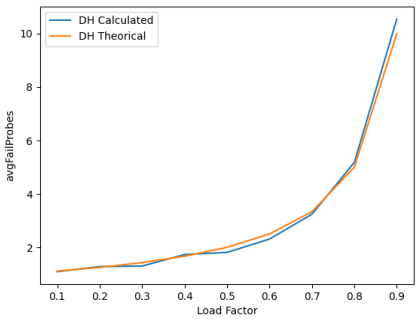

Figure 13: Fail probes

Double Hashing has similar behavior to that of linear probing but better because it needs just 10 probes maximum.

Open Addressing : Cuckoo Hashing

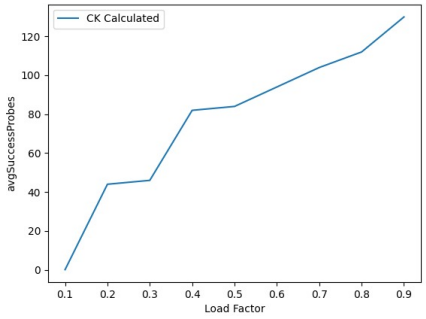

Figure 14: Successful probes

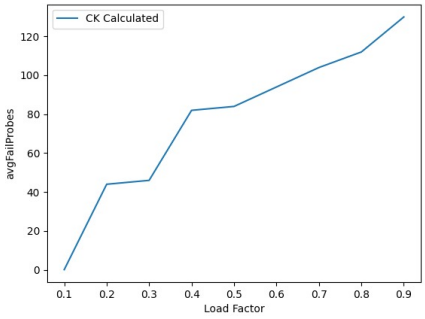

Figure 15: Fail probes

From Cuckoo Hashing, we did not find the theoretical behavior so we can just comment that the performance was done by both successful and fail queries is identical. The bad conduct is due to the inner cycles that have its implementation.

Separate Chaining

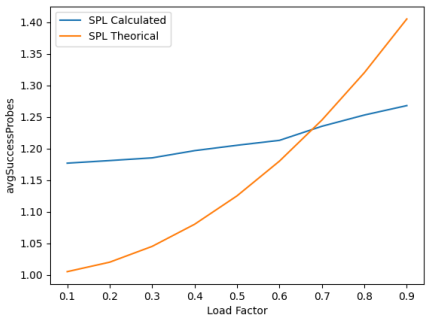

Figure 16: Successful probes

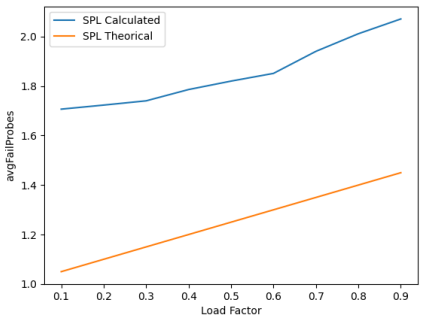

Figure 17: Fail probes

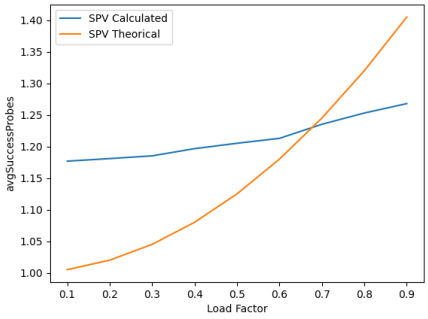

Figure 18: Successful probes

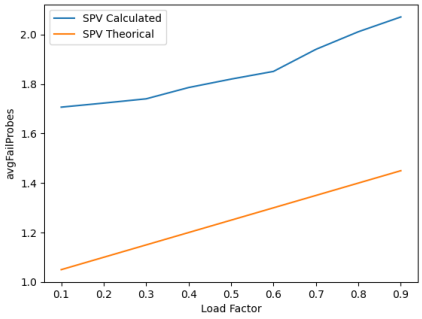

Figure 19: Fail probes

From all Separate Chaining's plots, we can observe that with both queries, with lower load factor we get fewer collisions and this decreases the number of accesses needed. As the load factor increases, the number of collisions also increases and as we have no repetitions, it becomes harder to find the value in the first search in the vector or list. Also, we can observe that all plots have a growing trend but with a vertical displacement that we attribute to the conditions of the experiment and the implementations of the algorithms.

False positives

This section is specifically centered on the false positive rate of the bloom filters since the other techniques do not have.

Both plots have a similar figure but in this case, the experimental results show a much better performance with a lower false-positive rate.

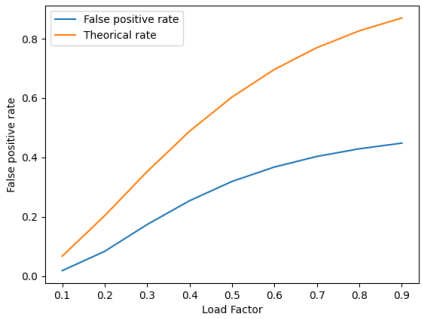

Figure 20: False positive rate comparison

6 Discussion

Along with this research work, we found out that no matter the implementation all techniques have a similar lookup time.
On the other hand, the implementation does affect the build time and, as we have seen with the experiment results, *bloom filters* is the one that performs faster; unfortunately, it has the issue of *false positives*.

Furthermore, in this project, we have learned how to design experiments, collect data, extract results and conclusions. We also have learned teamwork and the fair use of collaborative tools such as git and overleaf.

Finally, we wanted to add that we have struggled so many hours with LaTeX, the position of the plots and its compilation errors.

References

[1] Burton Howard Bloom. "Space/Time Tradeoffs in Hash Coding with Allowable Errors." In: *Communications of the ACM* (1970).

[2] Andrei Broder and Michael Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In: *Internet Mathematics* (2004).

[3] Bernhard Grill. "A Survey on Efficient Hashing Techniques in Software Configuration Management". In: (2014).

[4] Adam Kirsch and Michael Mitzenmacher. "Less Hashing, Same Performance: Building a Better Bloom Filter". In: *Wiley InterScience* (2007).

[5] Donald Knuth. *The art of computer programming, Volume 2*. Addison Wesley, 2011.

[6] Donald Knuth. *The art of computer programming, Volume 3*. Addison Wesley, 2011.

[7] Professor Clark F. Olson and Carol Zander. “Hashing and Hash Tables.” In: *University of Washington Bothell’s web site.* (2017).

[8] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo Hashing.” In: *Brics: Basic Research in Computer Science.* (2001).

[9] Abubakar Sulaiman Gezawa Abdurra’uf Garba Saifullahi Aminu Bello Ahmed Mukhtar Liman and Abubakar Ado. “Comparative Analysis of Linear Probing, Quadratic Probing and Double Hashing Techniques for Resolving Collusion in a Hash Table.” In: *International Journal of Scientific and Engineering Research.* (2014).