Programming Project 2 - Fall 2022
EN.605.620.81.FA22 Algorithms for Bioinformatics

Variations on Hashing

Gerald McCollam

Professors
Eleanor Chlan & Sidney Rubey

Contents

1 Introduction

The term ‘hash‘ typically refers to mixed chopped meat. As a verb, it means to "chop and mix thoroughly". This is exactly how the technical term should be read. A hash function chops and dices bits of data — typically of arbitrary type and size — into other types and sizes. In this lab, we consider different flavors of hashing in terms of their use as functions and in hash tables. We want to understand the ramifications of using one or other hashing technique in terms of how it manages a challenge facing all hashing methods: how to avoid conflict, or *collisions*. We'll explore the idea of collision in hashing and evaluate different collision resolution techniques. These approaches fall into two general categories:

- **Closed Hashing** (Open addressing)
  - Linear Probing
  - Quadratic Probing
- **Open Hashing**
  - Separate Chaining

In closed hashing, a hash table has a static, predetermined size. If a collision occurs during insertion, an alternative is tried until an empty bucket is found. In open hashing, collisions are resolved by the use of an element array that stores objects having the same key. We'll explore each of these in greater depth.

2 What Are Hash Tables?

To understand hashing, first consider the fundamental data structure on which it depends, the hash table, or hash map. Hashing is a way of sending data through a function so that it returns a simpler reference that's smaller and more easily managed. For example, one can hash a complex object and reduce it to a single integer representation to be used as an index into an array of buckets. The term ‘bucket‘ simply refers to a storage location in the array. In technical terms, a hash table is an abstract data type that maps keys to values. When inserting, the hash function computes an index into a table of ‘buckets‘. When retrieving, the key is again ‘hashed‘ and this yields the corresponding value from the table. [1]

2.1 Hash Functions

For our purposes, a hash function takes a key and produces an integer which is then used as an index into an array. There are many possible hash functions and each can be suited to a particular data type and application. Common to all is the need to address certain trade-offs, among them being space and time limits. If we had infinite space then our hash function could be extremely simple, we could use the input value itself as a key in such cases. Given infinite time, we might hash every key to the same integer and perform a sequential search.

2.2 Collisions

Unfortunately, there's no free lunch. Eventually we must acknowledge that no hashing function is perfect. By the "pigeonhole principle" if n items are put into m containers, with $n > m$, then one container will hold more than a single item. We can create algorithms that resist collision, but eventually any algorithm under the right circumstances will map different data to the same hash. A more realistic approach is to avoid the collisions that inevitably arise by managing them.

2.3 Open Addressing

Another way of understanding why hash collisions are inevitable is the birthday paradox. Consider the probability of a set of two randomly chosen people having the same birthday out of n people. It's surprising to learn that this probability is greater than 50% in a group of only 23 people. In fact, a well-known application of the birthday paradox is an attack that reduces the complexity of finding a collision in a hash function. [2] Since hash functions map keys of variable-length to fixed-length indices (i.e. they map an infinite set to a finite one) collisions will eventually occur. An intuitive way of resolving such collisions is through **open addressing**, where we catch and address the collision immediately by searching for an alternative location in the array. There are many possible ways to probe for an open slot; here we discuss two, **linear** and **quadratic** probing.

2.4 Separate Chaining

For open addressing methods such as linear probing (where the interval between probes is fixed) or quadratic probing (where the interval increases quadratically), we assume a fixed container size that cannot be increased (unless we resize it). This leads some to refer to open addressing as, confusingly, closed hashing. In **separate chaining** we address the limits of slot size by turning

each slot into a sub-array; thus, the parent array becomes an array of arrays. Using this method, all keys that map to the same hash value are now kept in a list. If we incur a collision, each slot of the array now contains its own array for any key-value pair having the same hash. New key-value pairs are added to the end of the list so that it just keeps growing.

3 Solution

3.1 Problem Statement

In this lab, we are asked to implement and test a number of these variations on hashing. Our assigned "schemes" are by division modulo 120, by 113, and by 41. We choose our own hashing scheme for each handling method and I've chosen multiplicative. The assigned collision schemes are linear, quadratic, and chaining. Our bucket size varies for open addressing: being of size 1 or size 3. For chaining, bucket size is always 1.

3.2 Implementation

My approach to each collision handling scheme is to contain all data structures and operations into separate classes. My code implements a single class for each technique: *LinearProbing*, *QuadraticProbing*, and *SeparateChaining*. Operations upon each include insertion, deletion, search, and an internal hash function. An initialization method allows for setting properties, such as the type of hashing method, modulus, slot size, and slot depth. Here, slot depth refers to the 2nd dimension in a two-dimensional array. This is used only when bucket size is equal to 3, though my code allows for any arbitrary 2d size.

3.3 Testing Methodology

To test a hashing method, any number of approaches might be used. The key is to gauge how a hashing scheme manages collisions where each method has its own benefits and drawbacks. By analyzing differences in collision rate versus load factor, slot size and table size, along with execution time, the optimal solution can be identified for a particular task. The load factor is the number of elements in a hash table divided by the number of available slots; it is signified as α. In general, the higher the load factor, the slower the operations of insertion and retrieval will be.

3.4 Assumptions & Analysis

Our assigned data consists of six sets of ten integers, where table size = 120. The highest expected load factor when hashing 60 elements (n) over 120 slots (m) is $n/m = \alpha = 0.5$. If we assume an initial constant cost of hashing, then the cost of our hash table operations (assuming we have a reasonable hash function) will be, on average, $O(1+\alpha)$. If the load factor α does not exceed some fixed value of α, then any operation on the hash table is $O(1 + max(\alpha)) = O(1)$. However, this is a theoretical goal, in practice we'll get best performance when α is within range of about $1/2$ to 1. In sparse, smaller arrays performance is likely to be enhanced when $\alpha = 0.5$. [3]. For this reason, I have created a slightly larger data set and have based my analysis on it. My approach assumes that a more interesting analysis can be achieved if we raise the max expected load factor. For example, it is known that clustering becomes an issue for linear probing, where performance degrades with $\alpha > 0.5$. [3] Thus, I have based this analysis on a created dataset containing ten by ten integers where $max(\alpha) = 0.833$ (given a table size of 120 slots).

3.5 Resizing

Each of the three classes I have written for this assignment has the ability to increase its capacity if/when the load factor goes beyond a certain threshold. In normal use - when we can't predict how large an array should be or how many slots it should hold - a dynamically-resizable array is used. When adding a new element causes α to exceed some threshold, the hash table generates a larger array whose size is a multiple of its original size. This capability is not used for the current exercise.

3.6 Results

A summary is presented in **Table 1**. This table provides a compilation of all hashing test combinations and their accompanying results. Over the eleven tests assigned, these fall into four groups, those that maintain a one-dimensional array, with modulus of 120 or 113 (Groups 1, 2. and 4) and those with modulus of 41 and bucket size of 3 (Group 3). Group 4 uses a multiplicative hashing scheme. A set of 100 integers were used for each run as input data; the table size was 120 slots regardless of bucket size; execution time reveals that the separate chain/multiplicative scheme (11-4) was fastest; it also shows relatively few collisions.

#/Grp	Scheme	Collisions	Load	Filled	Empty	Time (ms)
1-1	**Linear Probe Div Mod 120 Bucket Size 1**	171	0.833/1	100	20	2.210
2-1	**Quadratic Div Mod 120 Bucket Size 1**	51	"""	"""	"""	1.217
3-1	**Separate Chain Div Mod 120 Bucket Size 1**	61	"""	"""	"""	1.198
4-2	**Linear Probe Div Mod 113 Bucket Size 1**	192	"""	"""	"""	1.202
5-2	**Quadratic Div Mod 113 Bucket Size 1**	46	"""	"""	"""	1.193
6-2	**Separate Chain Div Mod 113 Bucket Size 1**	64	"""	"""	"""	1.173
7-3	**Linear Probe Div Mod 41 Bucket Size 3**	3098	"""	"""	"""	3.041
8-3	**Quadratic Div Mod 41 Bucket Size 3**	73	"""	"""	"""	2.646
9-4	**Linear Probe Multiplicative Bucket Size 1**	245	"""	"""	"""	1.166
10-4	**Quadratic Multiplicative Bucket Size 1**	48	"""	"""	"""	1.191
11-4	**Separate Chain Multiplicative Bucket Size 1**	62	"""	"""	"""	1.152

Table 1: Compilation of hashing test combinations and accompanying results.

Regarding schemes, linear probing reports the highest number of collisions, regardless of modulus or bucket size, whereas quadratic probing shows the fewest. Looking at execution times, linear probing is slowest (note that the result for 7-3 is suspect, in red) for division mod 41 and bucket size 3; separate chaining and quadratic are generally fastest with the exception of 8-3, quadratic using division mod 41 and bucket size 3. It is notable that the default table size, 120, is not a prime number.

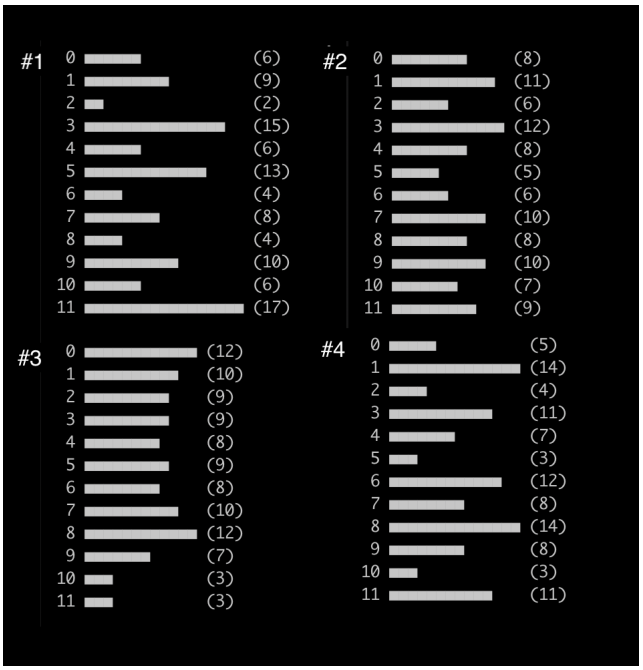

Figure 1: Hash key distributions for the four Groups, bin size=12.

Figure 2: Hash key distributions, bin size=2

Figures 1,2 presents a different perspective and may serve as an adjunct to the previous table. It is expected that a hash function has some overlap in its output, where two keys produce the same hash and result in a collision. In addition, how generated keys are distributed across a hash table also impacts the end result. Clustering occurs when there is less-than-optimal distribution of hash codes that don't take advantage of all available slots. What if the distribution of hash values is uneven?

3.7 Hypothetical Versus Actual

Figure 2 shows the distribution of hash keys over two bins. Ideally, one hopes for an even distribution - two horizontal lines that line up - whereas, for the four groups shown, the closest

we have that match is in Group 3 (division mod 41, bucket size 3). If we increase the bin size to 12 (as in **Figure 1**) we get another perspective. Group 3 seems to have the least random distribution of all groups. Groups 1 and 4 seem equally random in relative terms in **Figure 1**, though Group 4 seems less so in **Figure 2**.

The hypothetical notion of a hash function is as a random Oracle.[1] For any given input, the Oracle should return uniformly random output. This is an ideal of how hashing ought to work and not being uniform certainly weakens the strength of any hash function. But is randomness a necessary condition for good performance? If the hash distribution is 'normal' then certain outputs should be more common than others and we should expect more collisions. In effect, this may be what we see in the distribution for Group 3. It seems well distributed, but Group 3's performance is not the best.

3.8 Plots

In **Figure 3** (following page) a series of plots pair total collision number (on the y-axis) against load factor (on the x-axis, 0.0-0.8) for all test combinations. The four rows coincide with the four different groups: purple plots are those generated by linear probing; green is quadratic, and yellow is separate chaining.

On first impression these plots look fairly similar, which is not what one expects, with the exception of the yellow for separate chaining. On closer inspection several trends are evident. For example, all bear out the fact that in smaller arrays (such as these) performance is likely to be enhanced when $\alpha \leq 0.5$. [3]. Indeed, this seems to be the case for all.

Another point is that while all plots look similar they're not the same: the y-axis represents the total number of collisions and that total varies widely among them. The collisions span a range (discounting 7-3) from 0-50 to 0-250. What is of interest is that within that range, all behave in a similar way. Once $\alpha \geq 0.5$ then collisions grow at a rate that's more than linear. As the load factor increases, the performance degrades because of the time required to resolve collisions.

We expect $O(1)$ performance from a hash table for insertion, deletion and search. In the worst case we should expect no less that $O(n)$ for each operation. Often, the worst case is the result of the underlying array not being large enough. Another cause is through the effects of clustering.

[1] An Oracle conceived as a theoretical black box that responds to every query with a truly random response.

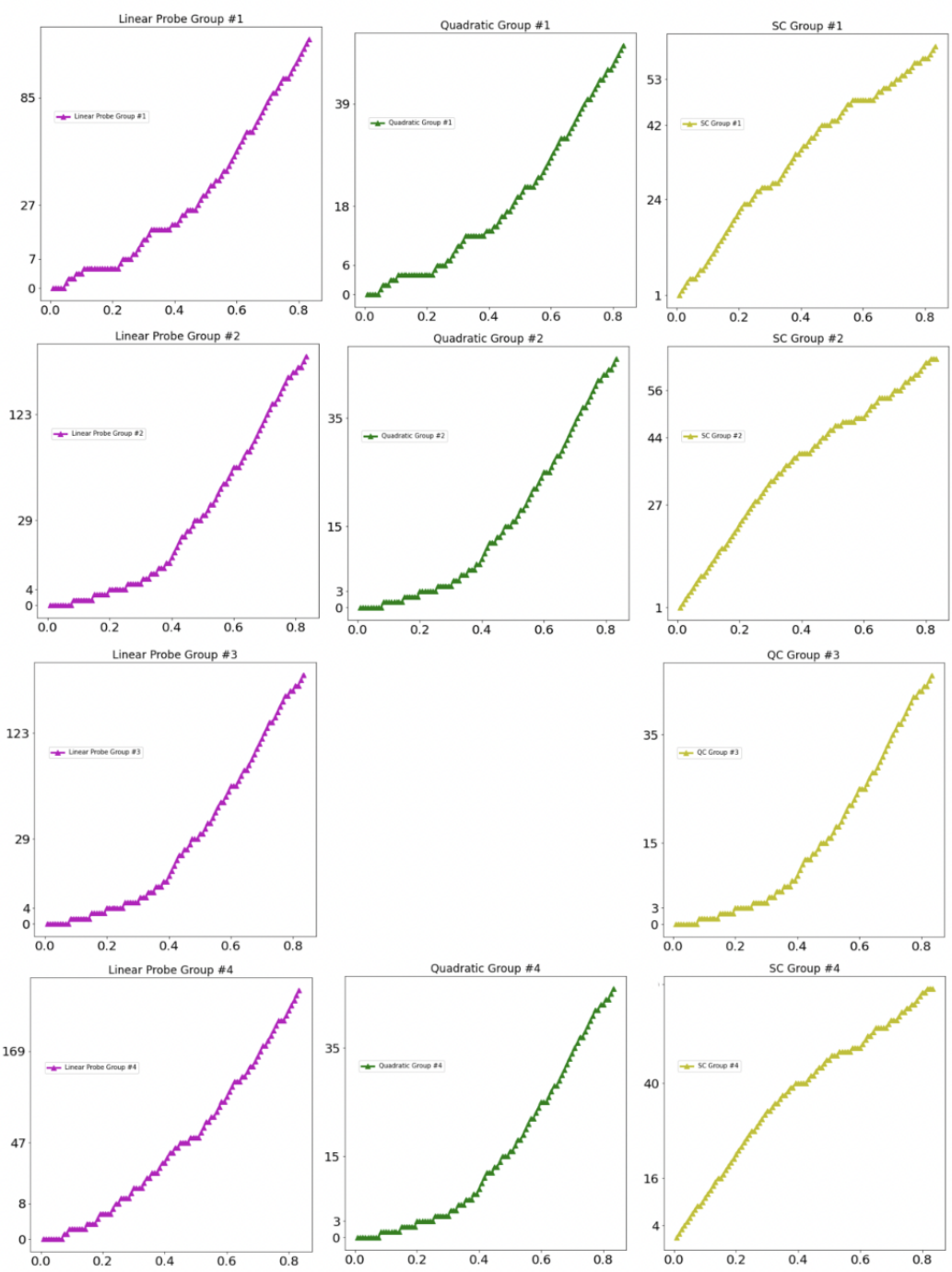

Figure 3: Plots pairing collision number (y-axis) against load factor (x-axis) for all test combinations. Purple plots are linear probing; green plots are quadratic probing, yellow is separate chaining.

What these graphs suggest is that in all cases regardless of scheme the expected constant running time can be compromised by a weak hash function and/or by the effects of clustering. If clustering is occurring, some buckets will have more elements than they should, and some will have fewer. Ideally, if the number of buckets is proportional to the number of elements contained

in the table, then we'll have $n = O(m)$ and, by extension, $\alpha = n/m = O(m)/m = O(1)$. This seems never to be the case for any of the plots in **Figure 3**. At best, they are linear or perhaps approaching some function that is linear times a constant value.

4 Discussion

Completing this lab was one of the more challenging assignments this semester. While the concept of hashing and hash tables are straightforward, I struggled to unpack and understand the relationships that determine hashing performance, e.g. those that obtain between the type of function, the size of the array, and the scheme for handling collisions. In this exercise, I spent time implementing each scheme successfully; I did not have sufficient time to exercise them over a larger dataset. One lesson learned was the benefit of multiplicative versus division for hashing. Multiplicative hashing is cheaper than modular hashing because multiplication is usually faster than division.

Compare modular hashing $h(K) = k \ mod \ M$, where k is the key value, and M is the size of the hash table, with multiplicative hashing, $h(K) = floor(M \ (kA \ mod \ 1))$ where M is the size of the hash table, k is the key value, and A is a constant value. The benefits of modular hashing are that it can be fast and it's good for any value of M. On the other hand, multiplicative hashing is much faster on the condition that table size is a power of two. In this lab, the table size was not a power of 2 so it's unclear whether or not we gained any of that advantage.

With regard to results obtained, I am confident that each scheme implemented performs as it should, with the possible exception of Group 3 (linear probing, div mod 41, bucket size 3). One change that I would make in revising my program would be to implement a linked list approach for the array expansion, where bucket size is greater than 1. I did so for the separate chaining part; in other cases, I relied on tricks to flatten and unflatten an n-dimensional Python array. These techniques are effective, but likely require extra work that impacts performance.

5 Applicability to Bioinformatics

It's hard to overstate the importance of an efficient hashing algorithm to any number of diverse applications. In bioinformatics, we have see how counting k-mers (substrings of length k) in DNA/RNA sequencing reads is an important preliminary step to many bioinformatics applications. [4] In the article cited with the assignment, an approach is presented that is capable of

counting all 27-mers of the human genome dataset with no more than 4.0 GB of memory and moderate disk space (160 GB). This remarkable achievement is entirely dependent on efficient hashing.

Another example that depends ultimately on efficient hashing is found in the work of Jianan Wang, et al. [5]. In their paper, the authors propose an efficient k-mer counting algorithm based on a lock-free chaining hash table. Known as CHTKC, their work proves that hash-table-based methods can solve the k-mer counting problem in an effective manner.

6 Conclusion

This has been a rewarding exercise. While my results are accurate, I feel they stop short of revealing what they potentially might show. For example, if my algorithm implementations were run on larger data, they would likely reveal more about the relative performance differences that these several hashing configurations possess.

References

[1] W. contributors., *Wikipedia*, 2022, December 14. [Online]. Available: https://en.wikipedia.org/wiki/Hash_table.

[2] M. Soltanian, *Theoretical and Experimental Methods for Defending Against DDoS Attacks.* OCLC, 2015.

[3] D. Liu, *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, p. 477–484, 2014. [Online]. Available: https://doi.org/10.1109/ICIS.2014.6912180

[4] D. L. Guillaume Rizk and R. Chikhi, "Dsk: K-mer counting with very low memory usage." *Bioinformatics*, vol. 29, no. 5, p. 652–53, 2013. [Online]. Available: https://doi.org/10.1093/bioinformatics/btt020

[5] J. Wang, "Chtkc: A robust and efficient k-mer counting algorithm based on a lock-free chaining hash table," *Briefings in Bioinformatics*, vol. 22, no. 3, p. 652–53, 2021. [Online]. Available: https://doi.org/10.1093/bib/bbaa063