New York University Tandon School of Engineering

Department of Computer Science and Engineering

Algorithm II: Lecture Note 12 - Randomization in algorithm design (Fall 2023)

Authored by Majid Daliri

Contents

Universal Sets of Hash Functions

Definition: A family H of hash functions mapping a universe U to a set of hash codes $\{0, 1, \ldots, m-1\}$ is called *universal* if, for every two distinct elements $u, v \in U$, the number of hash functions $h \in H$ for which $h(u) = h(v)$ is $|H|/m$. Formally, H is universal if:

$$\forall u, v \in U, u \neq v : \quad Pr_{h \in H}[h(u) = h(v)] = \frac{1}{m} \tag{1}$$

Universal hashing mitigates the worst-case scenarios that can occur in hash tables, especially under adversarial conditions where an opponent might deliberately choose keys that cause collisions. By using a hash function chosen randomly from a universal family, we can ensure that the performance of the hash table remains efficient on average, regardless of the input.

Universal hash functions are widely used in scenarios where the distribution of input data cannot be predicted. They are crucial in creating hash-based data structures like hash tables, hash maps, and in cryptographic functions where predictable collisions can lead to security vulnerabilities.

Perfect Hashing and Static Perfect Hashing Functions

Perfect hashing is a technique used in computer science to create a hash function that has no collisions. This is typically achieved in two levels. The first level maps each key to a slot in a hash table, and the second level is a secondary hash function that handles the keys mapped to the same slot without collisions.

Family of Perfect Hashing:

- A family of perfect hashing functions ensures that for any given set of keys, there exists at least one hash function that creates a perfect hash table with no collisions.
- This is particularly useful in scenarios where the set of keys is static and known ahead of time.

Static Perfect Hashing:

- Static perfect hashing refers to a scenario where the set of keys is fixed and known beforehand.
- In this case, a perfect hash function can be created that maps each key to a unique slot in the hash table without any collisions.
- This approach is highly efficient for lookup operations, as it guarantees constant time complexity, $O(1)$, for searching a key.

Expected Size of Collision

In this section, we focus on calculating the expected number of collisions in a hash table, which is key to understanding hash table efficiency in data structures and algorithms.

Hash tables store data in an associative manner, using a hash function $h(x)$ to convert an input ('key') into a set of numbers for indexing. A collision occurs when two distinct keys (k_i and k_j) hash to the same value.

Consider $k_1, k_2, \dots, k_n$ as distinct keys and m as the size of the hash table. The collision pairs are pairs (i, j) where $i \neq j$ and $h(k_i) = h(k_j)$.

Define an Indicator Variable $X_{i,j}$ as $X_{i,j} = 1[h(k_i) = h(k_j)]$, which is 1 if $h(k_i) = h(k_j)$, else 0. The total collision count is collision $= \sum_{i,j<n} X_{i,j}$.

The expected number of collisions, $E[\text{collision}]$, is calculated as follows:

$$\begin{aligned}
\mathbb{E}[\text{collision}] &= \sum_{i,j<n} \mathbb{E}[X_{i,j}] \\
&= \sum_{i,j<n} \Pr[h(k_i) = h(k_j)] \\
&= \sum_{i,j<n} \frac{1}{m} \\
&= \binom{n}{2} \times \frac{1}{m}
\end{aligned}$$

This shows that the expected number of collisions in a hash table depends on the number of keys, the size of the hash table, and the uniformity of the hash function. This understanding is fundamental for designing efficient hash functions and managing collision handling in hash tables.

Analysis of Static Perfect Hashing with Quadratic Table Size

In the domain of static perfect hashing, setting the size of the hash table, m, to the square of the number of keys, n (specifically, $m = n^2$), significantly affects the expected number of collisions. Under this configuration, the expected size of collision pairs becomes $\frac{1}{2}$. Utilizing Markov's inequality, we understand that since the number of collisions is an integer value, the probability that the number of collisions is less than 1 is greater than $\frac{1}{2}$:

$$\Pr[\text{Collision} < 1] > \frac{1}{2} \tag{2}$$

By trying only once, there is a 50 percent chance of no collision occurring in the hash table, leading to an efficient lookup time of $\Theta(1)$. However, an iterative trial approach may be employed, where the construction process is repeated until a hash table of size n^2 without any collision is achieved. This iterative approach ensures that, eventually, a perfect hashing table is constructed. The search operation in this hash table will have a worst-case time complexity of $\Theta(1)$, but it requires $\Theta(n^2)$ space. Given that the probability of success in each trial is $p = \frac{1}{2}$, similar to a fair coin toss, the expected number of trials until a successful construction is $\frac{1}{p} = 2$. Therefore, the expected time to construct a collision-free static perfect hashing table is twice the time taken for a single trial. This approach to static perfect hashing, characterized by quadratic table size and iterative trials, balances efficient lookup times with the expectation of achieving a collision-free table in a reasonable number of attempts, albeit at the cost of increased space complexity.

Two-Level Hashing with Quadratic Subtable Size

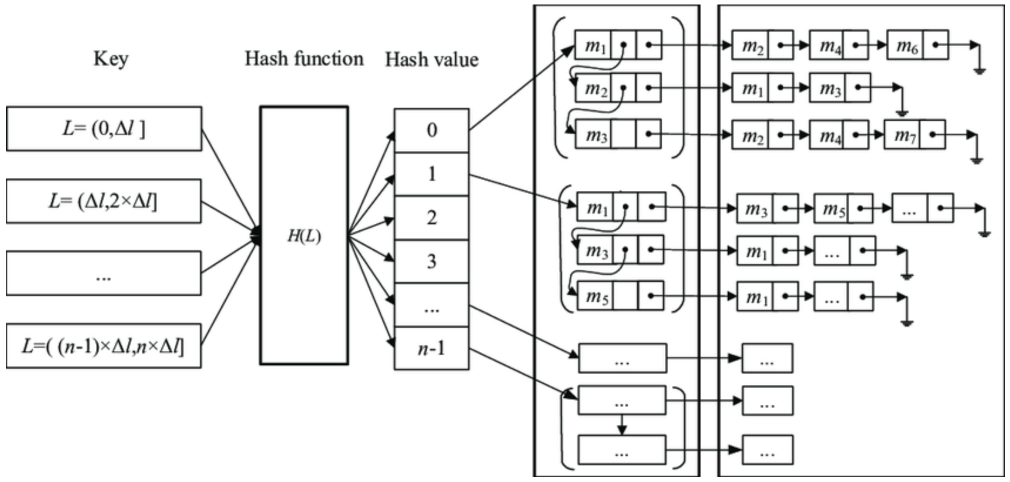

Figure 1: Two-Level Hashing.

In our pursuit to optimize hashing, we employ a two-layer hashing structure. This involves constructing a data structure where each key k_j maps to an index i, and n_i denotes the number of keys that map to i. We then build a subtable of size n_i^2 for each index i, applying the previous method of collision expectation management.

The structure comprises a top-level hash function h which builds a main table T of size n. Each entry in T is associated with a second-level hash function $h_i : U \to \{0, \ldots, n_i^2\}$. The lookup process involves two steps: first, finding the appropriate subtable T_i where $i = h(k)$, and then searching within $T_i(h_i(k))$.

When considering the construction time and space requirements, it is crucial to recall the expected number of collisions. When $m = n$, the expected number of collisions is calculated as:

$$\frac{\binom{n}{2}}{m} = \frac{\frac{n*(n-1)}{2}}{n} = \frac{n-1}{2}. \tag{3}$$

An important identity for an integer x is:

$$2\binom{x}{2} + x = x^2. \tag{4}$$

Applying this, the main table T is of size n, and for each entry, we have a subtable of size n_i^2. Thus, the total size can be expressed as:

$$\sum_{i=0}^{n} n_i^2 = 2\sum_{i=0}^{n}\binom{n_i}{2} + \sum_{i=0}^{n} n_i, \tag{5}$$

where $\sum_{i=0}^{n} n_i = n$.

Focusing on $2\sum_{i=0}^{n}\binom{n_i}{2}$, the expected size of $\sum_{i=0}^{n}\binom{n_i}{2}$ is the number of collisions of the top-level function, computed as $n-1$. Therefore, the expected total size of the entire structure is $n-1+n=2n-1$.

This ingenious approach allows us to achieve search operations in $\Theta(1)$ and store $\Theta(n)$ items in expectation, striking a remarkable balance between efficiency and space.

Probabilistic Construction of Two-Level Hashing Structure

In the previous sections, we focused on the expected behavior of our two-level hashing structure without delving into the probabilities and risks associated with building these data structures. We now turn our attention to a method that ensures the effectiveness of this approach.

The primary strategy involves retrying the top-level hash function h if the number of collisions exceeds twice the expected value, which is $2 \times \frac{n-1}{2}$. The probability of having to retry is $\frac{1}{2}$, indicating that, on average, we may need to attempt this process twice (since $1/(1/2) = 2$) to achieve a desirable number of collisions. This retry mechanism ensures that we can successfully construct the hash structure with a reasonable expectation of success.

Once a suitable top-level hash function is established with fewer collisions, we proceed to build the two-level structure. The space required for this structure is $\Theta(n)$, and the search operation within this framework operates in $\Theta(1)$ time. However, due to the probabilistic nature of the construction process, the expected time to build this two-level structure is approximately $\Theta(n)$, factoring in the likelihood of needing to retry the top-level hashing.

This approach not only guarantees a highly efficient lookup time but also ensures that the storage space is not solely based on expectations. By incorporating this retry mechanism, we significantly mitigate the risk of an unexpectedly large number of collisions, thereby making the structure more reliable and robust in practical applications.

Bloom Filters: A Data Structure for Approximate Membership Queries

Bloom filters are a practical data structure particularly useful when we do not have all the keys at once and they are arriving in a stream. This structure supports two operations: insertion and find. The find operation is probabilistic; a 'no' response is definitive, while a 'yes' response is likely but not certain.

Consider a large database with a Bloom filter at its front. When searching for an item, if the Bloom filter returns 'no', the item is definitely not in the database. If it returns 'yes', the item might be in the database, and a further lookup is necessary. This structure is not purely theoretical and is used in large-scale databases like those at Google.

We define three variables: n as the number of items to store, m as the table size, and k as the number of hash functions, which should be a small integer.

Consider a table T with m bits. The hash functions are $h_i : U \to \{0, 1, \ldots, m-1\}$ for $0 < i < k$.

Insertion Operation:

```
Insert(x):
    for i = 1 to k do
        T[h_i(x)] = 1
```

For each insertion x, we set the bits at positions $h_1(x), \ldots, h_k(x)$ in T to 1.

Find Operation:

```
Find(x):
    for i = 1 to k do
        if T[h_i(x)] == 0 then
            return "No"
    return "Yes"
```

When executing $Find(x)$, we check all the bits at positions $h_1(x), \ldots, h_k(x)$. If all are set to 1, then x might be in the set; otherwise, we are sure x is not in our set.

We assume that h_i functions are independent, meaning there is no correlation between the outcomes of different hash functions.

Benefits of Using Bloom Filters: Bloom filters are efficient in space and time, especially suitable for large data sets where exact membership information is not critical. They provide quick lookups and insertions, making them ideal for applications like cache filtering or network data processing.

False Positive Probability Analysis: The probability that a bit is not set by any of the k functions is $(1 - 1/m)^k$. Knowing that $\lim_{m\to\infty}(1 - 1/m)^m = 1/e$, we get $(1 - 1/m)^k = ((1 - 1/m)^m)^{k/m} = e^{-k/m}$.

After inserting n items, the probability that a bit remains unset is $(e^{-k \cdot n/m})^n$. Consequently, the probability that a bit is set (and hence the false positive probability, ϵ) is given by:

$$\epsilon = (1 - (1 - 1/m)^{kn})^k \approx (1 - e^{-kn/m})^k \tag{6}$$

To minimize ϵ, setting $k = m/n \cdot \ln 2$ gives:

$$\epsilon = (1/2)^k \Rightarrow k = -\frac{\ln \epsilon}{\ln 2} = -\log_2(\epsilon) = \log_2\left(\frac{1}{\epsilon}\right) \tag{7}$$

This analysis highlights how Bloom filters can be optimized for minimal false positives, making them a valuable tool for large-scale data management where exact precision is not paramount.

The Minimum Cut Problem in Undirected Graphs

The problem of finding a minimum cut, often referred to as the mincut problem, involves an undirected graph $G(V, E)$. A cut (A, B) is defined such that $A \cup B = V$, $A \cap B = \varnothing$, and both A and B are non-empty. The goal is to find partitions A, B that minimize the edge set $E(A, B)$, defined as $\{(u, v) \mid u \in A, v \in B\}$.

A lower bound for this problem is the minimum degree of the graph, as removing all neighbors of a vertex can disconnect the graph.

Related to Flow Problem: The mincut problem is similar to the flow problem with two main differences: the lack of edge direction in the mincut problem, and the constraint that the source and sink in the flow problem must be in different sets.

Definitions:

- *Multigraph*: A graph that may have several edges between a pair of vertices.
- *Edge Contraction*: Involves picking an edge between vertices u, v, merging them into a single node UV, and connecting all vertices linked to u or v to UV. In a multigraph, this can lead to multiple edges between nodes.

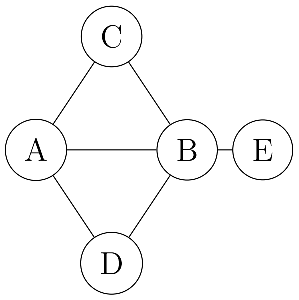

Figure 2: Original Graph

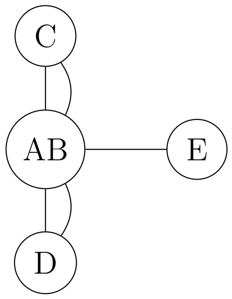

Figure 3: Graph after Contracting A and B

Algorithm for Mincut: The algorithm is straightforward. While the number of vertices in G is greater than 2, repeatedly apply the following steps:

1. Pick a random edge and contract the corresponding vertices.
2. Replace G with $G/\{e\}$, where e is the contracted edge.

Continue this process until only two vertices remain. In this resultant graph, there is only one cut, and the number of edges crossing this cut correlates with the mincut size.

Probability Analysis: Let OPT denote the size of the minimum cut in the original graph. The cut obtained by the algorithm will never be smaller than OPT, but there is a high probability of obtaining the mincut. The exact probability of this algorithm successfully finding the mincut depends on various factors, including the structure of the graph and the randomness of the edge selection.