Lecture 12 - Randomization in algorithms

Hashing

Hashing essentially performs a mapping from a set of keys to indexs/hash values (usually within a fixed range) using a function called the hash function.

Data structures like hashtables, hashmaps are built using hash functions to support operations like insertion, lookup and deletion of keys.

Universal hash functions

A family of hash functions is universal if, for any two items in the universe, the probability of collision is as follows,

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

Note that the probability of a collision using a random hash function is a property of the hash family, which means we maintain the performance of the hashing independent of the input by randomly choosing a hash function from the universal hash family. Universal hash functions are generally used in the case of unknown data distribution. They are an important class of hash functions forming the backbone of widely used data structures like hash-tables, hash-maps etc.

Perfect hashing

Perfect hashing is an ideal scenario where we want to have a hash function that has no collisions. We usually obtain this in two steps. The initial step corresponds to mapping each key to one of the slots in the hash table which is followed by a secondary hash function that accounts for the keys mapped to the same slot.

Family of Perfect Hashing: There exists a hash function from the family of hash functions such that its possible to have no collisions among the given keys. Mostly useful when the set of keys are fixed and known in advance.

Static Perfect Hashing: Static perfect hashing refers to the case when the keys are static and known ahead of time. Then it might be possible to construct a hash function such that each keys maps to a unique slot ensuring no collisions. This approach is highly efficient for lookup operations, as it guarantees constant time complexity, $O(1)$, for searching a key.

Dynamic Perfect Hashing: This a version of the perfect hashing case where the hashing method allows for a dynamic set of keys , supporting insertion and deletions. At a high level this shares the intuition of dynamic tables, i,e, the hash table can grow or shrink with respect to the number of active keys.

Collision analysis

The expected number of collisions is one of the key indicators for the performance of the hashing algorithm.

Given a hash function, a collision is happens when 2 distinct keys (k_i and k_j) hash to the same value/ index.

Consider a set of distinct keys $k_1, k_2, \ldots, k_n$, h as the hash function and m as the size of the hash value range. Formally the pair (i, j) is said to be in collision when $i \neq j$ and $h(k_i) = h(k_j)$.

To analyze this we shall define an indicator random variable X_{ij}. Note, that an indicator random variable is 1 when the event is true zero otherwise. Here, $X_{ij} = 1[h(k_i) = h(k_j)]$, which is 1 if $h(k_i) = h(k_j)$, otherwise 0. Hence, the total no of collisions = $\sum_{ij} X_{ij}.$, which covers all the pairs ij.

We want to analyse the the expected number of collisions, $E[\#\text{ collisions}]$, where:

$$\mathbb{E}[\#\text{ collisions}] = \sum_{ij} \mathbb{E}[X_{ij}] = \sum_{ij} \Pr[h(k_i) = h(k_j)] = \sum_{ij} \frac{1}{m} = \binom{n}{2} \frac{1}{m}$$

(expectation of an indicator random variable is the probability of it being 1.)
This shows that the expected number of collisions is dependent on the number of keys, the size of the hash table, and the uniformity of the hash function.

Static Perfect Hashing using Quadratic Table Size

In the case of static perfect hashing, the size of the hash table is an important factor tp compute the expected no of collisions. Specifically, if its set to be n^2, the expected no of collision pairs becomes 0.5.

$$E[\#collisions] = \binom{n}{2} / n^2 < \frac{1}{2}$$

Using Markov's inequality, we get:

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

Since the number of pairwise collisions is be an integer, the above inequality essentially computes the probability that the number of collisions is 0. Therefore if we try hashing only once, there is around 50% chance of no collisions. Given that the probability of success in each trial is $p = \frac{1}{2}$, the expected number of trials until we have the required hash function is $\frac{1}{p} = 2$. In other words, the expected time to construct a collision-free static perfect hashing table is twice the time taken for a single trial.

Two-Level Hashing

Note that while the above design achieves the desired quality, the space complexity of the method is not efficient. It turns out that we can achieve the same using linear space complexity without sacrificing the quality of the hashing. In order to do this, we employ a two-level hash table.

The top level comprises of a primary hash function h which corresponds to a main table T of size n. Each entry in T is associated with a secondary hash function $h_i : U \to \{0, \ldots, n_i^2\}$.

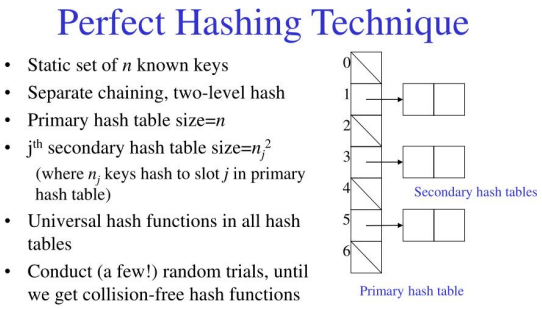

Figure 1: Two-Level Hashing.

Thus, the lookup happens in two levels: firstly, finding the slot in the main table T_i where $i = h(k)$, and then searching within that slot in the inner level, $T_i(h_i(k))$.

Putting $m = n$, the expected number of collisions becomes:

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

Using the following identity for any integer x:

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

the space complexity 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, which is equal to $\frac{n-1}{2}$. Therefore, the expected total size of the entire structure is $2\frac{n-1}{2} + n = 2n - 1$.

Thus, the space complexity is reduced from quadratic to linear in expectation while allowing search operations in $\Theta(1)$.

Construction of Two-Level Hashing Scheme

The top-level hash function h is reconstructed as long as the number of collisions exceeds twice the expected value, which is $2 \times \frac{n-1}{2}$. The probability of retrying is $\frac{1}{2}$, which implies we need 2 tries in expectation to succeed.

Once a suitable top-level hash function is achieved with small no of collisions, we proceed with the second level of the structure. Overall, the space complexity is. $O(n)$, search is $O(1)$, and the construction time is $O(n)$ in expectation, taking into account the rebuilding of the top-level hashing.

Bloom Filters

Bloom filters are a class of data structures particularly useful for answering membership queries in a stream of data. The 2 main operations supported are insertion and find. The find operation is probabilistic; it reports the non-existence accurately however the existence can be a false positive with some small probability. In addition to having good theoretical guarantees, they are often employed in real life large scale databases, 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.

Let n as the number of items , m be the table size, and k be the number of hash functions, which is usually a small number. Consider a table T with m bits and k hash functions, $h_i : U \to \{0, 1, \ldots, m-1\}$ for $0 < i < k$. Assume the hash functions are independent, meaning there is no correlation between the outcomes of different hash functions.

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 "does not exist"
    return "exist"
```

For the $Find(x)$ operation, we check all the bits in positions $h_1(x), \ldots, h_k(x)$. If any bit is 0, then x does not exist, otherwise x might be in the set.

Analysis: Assume that each of the hash functions h_i are independent and uniform. Therefore the probability that a bit is not set to 1 by any of the k hash functions is $(1-1/m)^k$. Using the fact, $\lim_{m\to\infty}(1-1/m)^m = 1/e$, we have

$$(1-1/m)^k = ((1-1/m)^m)^{k/m} \approx e^{-k/m}$$

.

The probability that a bit remains unset after n insertions is $(e^{-k\cdot n/m})^n$. Consequently, the probability that a bit is set or the occurrence of the false positive probability, ϵ, is:

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

For minimizing ϵ, we need $k = m/n \cdot \ln 2$:

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

This shows that Bloom filters can be optimized for very small false positives, making them a useful tool in large-scale data settings.

Randomized min-cut problem

Consider an undirected graph $G(V, E)$. A cut (A, B) is defined such that $A \cup B = V$, $A \cap B = \varnothing$, and both $A, B \neq \phi$. The goal is to find partitions A, B that minimize the size of the edge set $E(A, B)$, defined as $\{(u, v) \mid u \in A, v \in B\}$.

This is not a NP-Hard problem.

Similarity to network flow:
The min-cut problem is quite similar to the network flow problem except with 2 differences; the edges are un-directed, and there is no specified source or sink which has to be in different sets..

Definitions:

- Multigraph: The graph may have multiple edges between a pair of vertices.
- Edge Contraction: A merge operation of two nodes (u, v) of an edge of the graph into a single node while preserving all the connections of node u, v . Edge contractions remove self-loops but can create multiple edges between nodes.

Karger's algorithm: The algorithm is straightforward and simple.

```
Min-cut(G(V,E)):
    while |V|>2:
            Pick a random edge and contract the corresponding vertices.
            Replace G with G/{e}, where e is the contracted edge.
    return
```

In this resultant graph with 2 vertices, there is only one cut, and the number of edges crossing this cut correlates with the mincut size.

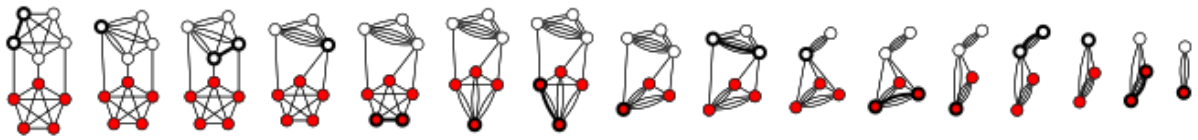

Figure 2: Sequence of edge contractions.

Let OPT be the size of the actual minimum cut in the original graph. Obviously, the cut obtained by Karger's algorithm can never be smaller than OPT. However, we can hope to obtain the min-cut with high probability. The exact probability analysis of this algorithm for finding the mincut depends on various factors, including the randomness associated with the edge contraction.