CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 10: Hash Functions and Graphs

Harvard SEAS - Fall 2021 *Oct. 5, 2021*

1 Announcements

Recommended Reading:

- Hash Tables Wrap-Up: CLRS 11.3, Roughgarden II 12.3.6–12.4
- Graph Search: Roughgarden II Sec 7.1–7.2, CLRS Appendix B.4
- PS4 due tomorrow!
- Participation Portfolio 1 due Sunday.
- Midterm practice problems to be released tomorrow
- COVID protocol: No eating/drinking in class
- Staff appreciation!

2 Hash Tables Wrap-Up

Last time we saw how we can use a random hash function $h : [U] \to [m]$ to obtain Las Vegas data structures for the dynamic dictionary problem that can do insertion in time $O(1)$ and search and deletion in *expected* time $O(1 + \alpha)$, where $\alpha = n/m$ is the *load* of the table and n is the number of item-key pairs being stored. (Here the expectation is taken over the choice of the random hash function h.)

Q: What is unsatisfactory about this solution?

1. To maintain both time and space efficiency, we need to tailor the size m of the table to the size n of the dataset, which we may not know advance. Solution: dynamically resize the dataset as the hash table gets too full. For example, when we reach load $\alpha = 2/3$, we can double the table size to bring us back to $\alpha = 1/3$.
2. We require a finite universe $[U]$ for our keys. Solution: see cryptographic hash functions below.
3. When we "choose a random function from $[U]$ to $[m]$" (i.e. set $h(K) = \texttt{random}(m)$, independently for every $K \in [U]$), merely *writing down* such a function takes time and space U, which is usually infeasibly large. Indeed, the reason we introduced hashing at all is that we didn't want to keep an array of size U.

Solution 1: we use a smaller family $\mathcal{H}$ of hash functions h that allows us to (a) store h compactly, (b) evaluate h efficiently, and (c) still prove that the worst-case expected time for operations on the hash table is $O(1+\alpha)$. An example: pick a prime number $p > U$. Then for $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ we define the hash function

$$h_{a,b}(K) = ((aK + b) \mod p) \mod m.$$

This takes 2 words to store, can be evaluated in $O(1)$ time, and maintains the same pairwise collision property: for every $K \neq K' \in [U]$, we have

$$\Pr_{a,b}[h_{a,b}(K) = h_{a,b}(K')] \leq \frac{1}{m} \tag{1}$$

(For a proof, see CLRS. This requires a little bit of number theory and is beyond the scope of this course.) A hash family satisfying (1) is known as an *universal hash family*, and this property suffices to prove our expected runtime bounds of $O(1+\alpha)$.

Solution 2: We could also use a "cryptographic" hash function like SHA-3, which involves no randomness but it conjectured to be "hard to distinguish" from a truly random function. (Formalizing this conjecture is covered in CS 127.) This has the advantages that the hash function is deterministic and that we do not need to fix a universe size U. On the other hand, the expected runtime bound is then based on an unproven conjecture about the hash function, and also these hash functions, while quite fast, are not quite computable in $O(1)$ time. By combining them with a little bit of randomization, they can also be made somewhat resilient against adversarial data, where an adversary tries to learn something about the hash function by interacting with the data structure and uses that knowledge to construct data that makes the data structure slow.

Q: Why do we choose a random function at all — why not choose some fixed function h?

For every fixed $h : [U] \to [m]$ where $U \gg m$, there will be a huge number of distinct keys that collide via pigeonhole. Since we are required to succeed over all possible inputs, we randomly pick h to make sure no fixed input always lies in one of these bad regions.

3 Storing and Search Synthesis

We have seen several approaches to storing and searching in large datasets (of item-key pairs):

1. Sort the dataset and store the sorted array
2. Store in a binary search tree (balanced and appropriately augmented)
3. Store in a hash table
4. Run Randomized QuickSelect

For each of these approaches, describe a feature or combination of features it has that none of the other approaches provide.

4 Graph Search

Motivating Problem: Google Maps. Given a road network, a starting point, and a destination, how can/should I travel to get from the starting point to the destination?

Q: How to model a road network?

A: graphs!

Definition 4.1. A *directed graph* $G = (V, E)$ consists of a finite set of *vertices* V (sometime called *nodes*), and a set E of ordered pairs (u, v) where $u, v \in V$ and $u \neq v$.

Example:

- Sometimes we allow *multigraphs*, where there can be more than one edge from u to v, and possibly also self-loops. Our definition as above is that of a *simple* graph.
- We have defined an *unweighted* graph, but we may also want to assign weights/costs/lengths to each edge (e.g. modelling travel time on a road).
- An *undirected* graph has unordered edges $\{u, v\}$. Equivalently, we can think of this as a directed graph where if $(u, v) \in E$, we also have $(v, u) \in E$. (We could think of this as a road network with no one-way roads.)
- A graph is *planar* if it can be drawn in a 2D plane without edge crossings. Road networks are mostly but not entirely planar (e.g. consider overpasses).
- Some real-world graphs have some additional (e.g. hierarchical) structure that might be useful to exploit in algorithms (e.g. we may know that usually the best way to drive from one city to another is to use local roads to get to/from a highway).

Unless we state otherwise, assume *graph* means a **simple, unweighted, undirected** graph, and a *digraph* means a **simple, unweighted, directed** graph.

Remark: graphs are useful for modelling a vast range of different kinds of relationships, e.g. social networks, the world wide web, kidney donor compatibilities, scheduling conflicts, etc.

Next time we will see an efficient algorithm for the following problem (which we will define more precisely):

Input	**:** A directed graph $G = (V, E)$ and two vertices $s, t \in V$
Output	**:** A *shortest path* from s to t in G, or $\perp$ if no path from s to t exist

Computational Problem ShortestPaths

Remark: here and in other computational problems we are using the failure symbol $\perp$ differently than we did with the RAM model. In computational problems, we mean that the algorithm should produce an output indicating that there is no solution. In the RAM model, we used it to mean that a program does not halt (i.e. runs forever). We should have used different notations for these two meanings! (Thanks to Albert for pointing this out.)