Advanced Algorithms Course. Lecture Notes. Part 12

Hashing

This part may be skipped if you know hashing already very well from Data Structure courses. But make sure that you also understand the probability theory behind it.

Let U be a universe (a huge set) of elements. A dictionary is a data structure that keeps track of a set $S \subset U$ and supports the following operations: insert, delete, lookup. That is, a dictionary enables us to quickly insert, delete, and retrieve elements of a set.

Hash tables are among the most important implementations of dictionaries. In the following, n is always some fixed size bound being much smaller than $|U|$. A hash table H is an array of size n, with indices $0, \ldots, n-1$, where $n \geq |S|$. That is, H allocates enough space for storing sets S of at most n elements. However, several elements may be stored in the same entry of H, for example as a list. Then we speak of collisions.

A hash function h maps U onto this index set. In order to execute any of the dictionary operations for an element, we compute the index of that element and access the corresponding entry of H. Of course, h must be easily computable, and it is essential that our hash function keeps collisions to a minimum: If many elements are stored in the same entry, we still have to search for the desired element there, and this would slow down the dictionary operation.

Since U is much larger than n, collisions cannot be avoided, but with a good randomized approach we can keep their expected number small. (Compare the situation to load balancing.) In the following, note again that randomness is only in the algorithm (here: in the design of our hash function h), but we do not make any probabilistic assumptions on the set S we want to store.

Here is a classical simple hashing scheme, along with a rigorous analysis of its performance. We will choose h at random from a certain class of easily computable functions. We call a function class "universal" if for any pair $u, v \in U$ the probability of $h(u) = h(v)$ is at most $1/n$.

Being universal is a good property for hashing, because, if we pick a random h from a universal class, then, for any fixed element u, the expected number of other elements $s \in S$ with $h(s) = h(u)$ is at most 1, and we barely get large bags of elements in the same entry of H. In particular, our dictionary will be able to do any operation in $O(1)$ expected time.

But do such universal classes of functions exist? Trivially, the class of *all* functions from U into the index set has this property. But what would it mean to choose a random h from the class of all functions? Since the values of such h are random and independent, h has "no structure", and we can "compute" the values of h for given elements only by looking them up, in a table of size $|U|$, which is against the very idea of hashing. We need a restricted class of functions which are easily computable but still "shake well" the elements of any subset with at most n elements. One possible construction uses facts from elementary number theory.

We choose a prime number p slightly larger than our n. (Prime numbers are "dense enough" in the set of integers, we will always find such p. We do not go into details of this preprocessing step.) We represent the elements of U as vectors $x = (x_1, \ldots, x_r)$ with $0 \leq x_i < p$ for all i. The dimensionality we need is clearly $r \approx \log |U| / \log p$. (This may look complicated, but note that these vectors can be seen as arbitrary "names" of the elements.) For every $a = (a_1, \ldots, a_r)$ we define a hash function $h_a(x) = (\sum_{i=1}^{r} a_i x_i) \bmod p$. For any given $x \in U$ these values are really easy to compute. It remains to analyze the collisions. We will see that the class of all functions h_a is universal.

Very little help from number theory is needed: If p is a prime number and $z \neq 0 \bmod p$, then $az = bz \bmod p$ implies $a = b \bmod p$ for any two numbers a, b. (The proof is straightforward.)

Using this fact we will show, for any two $x, y \in U$, that $h_a(x) = h_a(y)$ happens with probability at most $1/p$. Recall where this probability comes from: We took some random a.

Here is the proof. Since $x \neq y$, their vectors must differ somewhere. Let j be some position where $x_j \neq y_j$. A nice trick makes the probability calculation extremely simple: Instead of considering a random a, we fix all a_i, $i \neq j$, and choose only a_j randomly, where $0 \leq a_j < p$. Then the

probability result applies also to the random vector a. (Why?)

By the construction of h_a, a collision $h_a(x) = h_a(y)$ appears if and only if $a_j(y_j - x_j) = \sum_{i \neq j} a_i(x_i - y_i) \bmod p$. Since we have fixed the right-hand side, we can treat it as a constant, say m. Now define $z := y_j - x_j$. Due to the above number-theoretic fact, there exists exactly one a_j with $a_j z = m \bmod p$. Hence the probability of collision is $1/p \leq 1/n$, and our hash table can execute dictionary operations in $O(1)$ expected time.

A final remark: There is often confusion about the time complexity of hash table operations: $O(1)$ is the expected number of arithmetic operations. But the bit complexity is not constant. It grows logarithmically in the size of the sets we want to deal with. Thus, hash tables are asymptotically not faster than other dictionary implementations such as balanced search trees. The real advantage of hash tables is that they are easy to implement (just evaluation of some simple functions) and use only arithmetic operations, which are physically faster than manipulations with pointers, etc., that would be needed to implement trees.

Closest Points

For the problem of finding a closest pair of n points in the plane there exists a divide-and-conquer algorithm running in $O(n \log n)$ time. It follows a simple idea but is a bit complicated when it comes to the implementation details. Here we show a Las Vegas algorithm that is not only simpler but also solves the problem already in $O(n)$ expected time plus $O(n)$ dictionary operations.

We can always assume that our n points are in a unit square. In our algorithm we maintain a real number d which is the smallest distance between two points known so far.

We consider the n points in random order. For every new point p we test whether p has distance smaller than d to some earlier point, and in this case we update d.

For an efficient test we have to avoid computing the distances to *all* earlier points. Therefore we divide the unit square into squares of side length $d/2$. Since d is the smallest distance, at most one earlier point can be located in each square. Moreover, those points which might have a distance smaller than d to p are in squares close to the square containing p. More precisely, they are in a 5×5 grid of squares. Thus we have to test at most 25 candidates in every step. Hence $O(n)$ computations are enough, for all n

points. So far we have not even used the fact that points are processed in random order.

However, some complications begin here: We need to know which points are in the candidate squares! For this purpose we may use a hash table, with an entry for every point. But whenever d is diminshed, our partitioning into squares of side length $d/2$ changes totally, and we have to create a new hash table from scratch. How often do we have to insert our points into the various hash tables? Only here the randomized order of points becomes relevant.

Let X be a random variable for the total number of insertions. Let X_i be another random variable, with $X_i = 1$ if the ith point causes an update, and $X_i = 0$ else. Clearly, $X = n + \sum_i iX_i$.

The key fact is that we have $X_i = 1$ with probability at most $2/i$: For each number i, the first i points are randomly ordered as well. Hence, the event that some of the two points in a closest pair is exactly the ith point has probability $2/i$. Linearity of expectation gives $E[X] = n + \sum_i iE[X_i] \leq 3n$. Thus, the expected number of dictionary operations is $O(n)$, and each of them needs $O(1)$ expected time.

From these two facts it follows that the total expected time is $O(n)$. Stop! The latter conclusion seems obvious at first glance. But referring to linearity of expectation is not enough here, since the number of random variables to be added is a random variable itself. A real proof needs a careful analysis of conditional expectations, since we combine here two different sources of randomness. But here we skip these technicalities.