1 Static dictionary problem

In this lecture we will study the static dictionary problem and describe two-level perfect hashing. Sometimes it is also called “FKS perfect hashing” after its inventors: Fredman, Komlós, and Szemerédi.

Recall in the static dictionary problem you are given \( n \) (key, value) pairs up front and want to create a data structure supporting query (but not insert and delete). We will show how to do this in worst case query time \( O(1) \) (as opposed to the expected \( O(1) \) time for hashing with chaining) in \( O(n) \) space, with \( O(n) \) expected pre-processing time to create the data structure given the input database.

1.1 Quadratic space

To begin, recall the birthday paradox where, assuming random birthdays, you shouldn’t be surprised that two people have the same birthday when you have \( \sqrt{365} \) people in one room. Another way of viewing this is: if you have \( n \) people in a room with random birthdays but the calendar has much fewer than \( n^2 \) days, then you don’t expect any two people to share a birthday.

Let’s bring this back to hashing. If we were willing to make a table whose size is quadratic in size \( n \), then we will see that we can easily construct a perfect hashing of the input keys (where a hash function \( h \) is “perfect” for a set of keys \( S \) if \( h \) has no collisions amongst any keys in \( S \)). Let \( \mathcal{H} \) be a universal hash family and \( m = n^2 \).

Claim: If \( \mathcal{H} \) is universal and \( m = n^2 \), then \( P(h(\text{no collisions}) \geq 1/2 \) when using hash function \( h \in \mathcal{H} \).

Proof: In order for a collision to occur, some pair of keys \( x \neq y \) must equal each other amongst the keys in \( S \). Let \( Z_{x,y} \) be an indicator random variable for the event \( h(x) = h(y) \), so that the number of collisions is \( Z = \sum_{x \neq y \in S} Z_{x,y} \). Then using linearity of expectation,

\[
\mathbb{E}Z = \mathbb{E} \sum_{x \neq y \in S} Z_{x,y} = \sum_{x \neq y \in S} \mathbb{E}Z_{x,y} \leq \frac{n}{2}/m < 1/2.
\]

Note that a collision occurs iff \( Z \geq 1 \). Thus by Markov’s inequality, the probability a collision occurs is \( P(h(\text{collision occurs}) = \mathbb{P}(Z \geq 1) \leq \mathbb{E}Z < 1/2 \).

This is the opposite of the birthday paradox since we are looking for the probability that no pair of people has the same birthday. The complement then shows that the probability of no collisions must \( \geq 1/2 \). Our method then involves trying a random \( h \) from \( \mathcal{H} \). If we have any collisions, we pick another \( h \). On average, we would only need to do this twice.

1.2 Linear space

Let’s say we want to get a better space complexity. The general idea is that we are going to perform a 2-level hashing scheme: first we pick a random function \( h_1 \in \mathcal{H} \) from a universal hash family mapping \([U]\) into \([n]\) and hash
all elements, where our keys are all from the universe \( |U| := \{1, \ldots, U\} \). Let \( B_i \) represent the number of items that hash to bucket \( i \). We then wish to keep picking random \( h_i \in \mathcal{H} \) until we find \( h_i \) such that

\[
\sum_{i=1}^{m} B_i^2 \leq 4n
\]

Note that we do not know how long it will take to find \( h_i \) that fulfills the above condition. Once we do find our satisfactory \( h_i \), we can use the birthday paradox, which states that if we have \( n \) possible days in the year, once we have \( \sqrt{n} \), we can expect to find two people with the same birthday. Likewise, if we have significantly fewer than \( T = \sqrt{n} \), if we taking values between 1 and \( T^2 \) and we have much less than \( T \) people, we can be pretty sure that there isn’t a collision.

For bucket \( i \), we can then hash all \( B_i \) of the values in it to \( 1, \ldots, B_i^2 \) using \( h_i : |U| \rightarrow [10B_i^2] \), then by the birthday paradox, there are no collisions. To summarize, the entire method is then

1) First we take our elements that take on values \( v \in [n] \) and hash them using first-level hash function \( h : |U| \rightarrow |m| \) such that

\[
\sum_{i=1}^{m} B_i^2 \leq 4n
\]

where \( B_i \) is the number of items in the bucket \( i \). This creates first-level table \( A \).

2) Next, we use \( m \) second-level hash functions \( h_1, \ldots, h_m \) and \( m \) second-level tables \( A_1, \ldots, A_m \). Note that we pick \( h_i \) such that there are no collisions in \( A_i \).

Thus, we can ensure no collisions in the 2-level hash table structure. Our space complexity is \( O(m + \sum_{i=1}^{m} 10B_i^2) \) since we need \( m \) first-level buckets and \( \sum_{i=1}^{m} 10B_i^2 \) second-level buckets. Note that since we chose \( h \) such that the sum of \( \sum_{i=1}^{m} B_i^2 \leq 4m \), our space complexity is \( \Theta(n) \).

Let’s back up though. We initially said to keep picking \( h \) such that \( \sum_{i=1}^{m} B_i^2 \leq 4n \). How long will that take?

**Claim:** \( P_h(\sum_{i=1}^{m} B_i^2 > 4n) \leq 1/2 \)
**Proof:** Let $Q_{ji} = 1$ if item $j$ hashes to $i$ and 0 otherwise. We can rewrite $B_i$ as

$$B_i = \sum_{j=1}^{n} Q_{ji}$$

$$B_i^2 = \left( \sum_{j=1}^{n} Q_{ji} \right)^2 = \sum_{j=1}^{n} Q_{ji}^2 + \sum_{j \neq k} Q_{ji}Q_{jk}$$

$$= \sum_{j} Q_{ji} + \sum_{j \neq k} Q_{ji}Q_{ki} \quad (1)$$

$$\sum_{i=1}^{m} B_i^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} Q_{ji} + \sum_{i=1}^{m} \sum_{j \neq k} Q_{ji}Q_{ki}$$

$$= n + \sum_{i=1}^{m} \sum_{j \neq k} Q_{ji}Q_{ki} \quad (2)$$

$$\mathbb{E} \left( \sum_{i=1}^{m} B_i^2 \right) = n + \mathbb{E} \left( \sum_{i=1}^{m} \sum_{j \neq k} Q_{ji}Q_{ki} \right)$$

$$= n + \mathbb{E} \left( \sum_{i=1}^{m} \sum_{j \neq k} Q_{ji}Q_{ki} \right)$$

$$= n + n - 1 = 2n - 1 \quad (3)$$

Above, we show a few significant and potentially non-intuitive steps. In equation 1, we can drop the square of $\sum_{j=1}^{n} Q_{ji}$ since $Q_{ji}$ is either 0 or 1. In equation 2, we can reorder the summations: we know that $\sum_{i=1}^{m} Q_{ji} = 1$ since item $j$ will hash to exactly one of the $m$ values. We then find $\sum_{j=1}^{n} 1 = n$.

For equation 3, we know that $Q_{ji}Q_{ki}$ will have a product of 1 if and only if $h(j) = h(k)$, meaning $j$ and $k$ hash to the same bucket. Our resulting term is then

$$\mathbb{E} \sum_{j \neq k} 1 \text{ if } h(j) = h(k)$$

We can then linearize the expectation inside the summation, yielding the probability that the two item $j, k$ collide, which is exactly universal hashing. The probability of collision $\leq 1/m$, and there are $n(n-1)$ possibilities since there are $n$ choices for $j$ and $n-1$ choices for $k$, giving us

$$\sum_{j \neq k} P(j, k \text{ collide}) = \frac{n(n-1)}{m}$$

Since $m = n$, we get that $\sum_{j \neq k} P(j, k \text{ collide}) = n - 1$.

From Markov’s inequality, we get

$$P(X > \lambda \mathbb{E}X) < \frac{1}{\lambda}$$

$$P \left( \sum_{i=1}^{m} B_i^2 > 4n \right) < \frac{1}{2}$$

This clears up the mystery of why we chose 4 as a coefficient in $\sum_{i=1}^{m} B_i^2 \leq 4n$: because it’s twice the expectation.
Now we know when picking \( h \) randomly from \( H \), we have to pick an expected two times to fulfill our condition. Picking \( h_i \) is a similar story.

**Claim:** When picking \( h_i : [U] \to [10B_i^2] \), \( P(\exists \text{ collision}) < 1/2 \)

**Proof:** We define again \( X_{jk} = 1 \) if \( j, k \) collide under \( h_i \) and 0 otherwise. The expected number of collisions is then

\[
\mathbb{E}(\text{\# of collisions}) = \mathbb{E}\left( \sum_{j=1}^{B_i} \sum_{k=1}^{j-1} X_{jk} \right)
\]

\[
= \sum_{j=1}^{B_i} \sum_{k=1}^{j-1} \mathbb{E}X_{jk}
\]

\[
= \sum_{j=1}^{B_i} \sum_{k=1}^{j-1} P(\text{\# collisions})
\]

\[
\leq \sum_{k<j} \frac{1}{10B_i^2}
\]

\[
< \frac{B_i^2}{2} \cdot \frac{1}{10B_i^2} = \frac{1}{20}
\]

By Markov’s inequality again, \( P(\text{\# of collisions} \geq 1) < 1/20 \)

We have now shown that we can construct a 2-level hash table using hash functions \( h \) and \( h_1, \ldots, h_m \) to make a static dictionary since \( h \) and \( h_1, \ldots, h_m \) can be chosen in constant time each and linear time overall.

### 2 Karger’s algorithm

We are given an undirected, unweighted graph \( G = (V, E) \) with \( |V| = n, |E| = m \). A cut of \( G \) is a partition of the vertices into two non-empty sets. The weight \( w(C) \) of a cut \( C = (S, T) \) is the number of edges crossing the cut (i.e. \( w(C) = |\{ e = (u, v) \in E : u \in S, v \in T \}| \)). A minimum cut of \( G \) is a cut achieving the minimum weight over all cuts of \( G \). The algorithmic problem we consider is thus as follows: given \( G \), compute a minimum cut of \( G \).

One motivation for studying minimum cut is that it is some measure of the reliability of a network. Imagine that each vertex represents a network switch, and edges correspond direct links between switches. Let us assume our graph is connected (which hopefully our communication network is!). Then the weight of a minimum cut answers the following question: what is the minimum number of links that need to go down in our network to make our graph disconnected, i.e. so that not every switch can still talk to every other switch? The minimum cut doesn’t deal with switches themselves failing (i.e. vertex failures), but that’s a problem we won’t consider today.

There is an obvious brute-force algorithm for the minimum cut problem: try all cuts then take the one with the minimum weight. There are \( 2^{n-1} - 1 \) cuts to try. To see this, write \( V = \{1, \ldots, n\} \) and put cuts in correspondence with length-\((n-1)\) binary strings. The \( i \)th bit (1-indexed) is 1 if vertex \( i + 1 \) is on the same side of the cut as vertex 1. There are \( 2^{n-1} \) such strings, but the all-1s string is not allowed since that would mean one side of the cut would be empty.

A more efficient Monte Carlo algorithm, due to David Karger (a Harvard alum ’89!), is the following. It is known as Karger’s contraction algorithm. In what follows, \( G \) may be a multigraph (i.e. there may be multiple parallel edges between the same two vertices; even if \( G \) doesn’t start this way, it may become this way in later recursive calls).
Figure 1: Contracting one of the \((u,v)\) edges highlighted in red to merge \(u,v\) into \(w\).

procedure CONTRACT\((G = (V, E))\)

if \(|V| = 2\) then output the only cut
else
    pick a random edge \(e\)
    contract the edge \(e\) to form \(G' = (V', E')\)
return CONTRACT\((G')\)
end

What does it mean to *contract* an edge \(e = (u,v)\)? It means we remove the vertices \((u,v)\) from the graph and insert a new vertex \(w\). Any edge in \(G\) of the form \((u,x)\) for \(x \neq v\) is replaced by the edge \((w,x)\). Similarly any edge of the form \((v,x)\) for \(x \neq u\) is replaced by \((w,x)\). All edges of the form \((u,v)\) are removed. Figure 1 gives an example of obtaining some \(G'\) by contracting an edge in \(G\). Essentially when we contract \((u,v)\), we are saying that we are promising to place \(u,v\) on the same side of the final cut we output. Thus when \(|V| = 2\) in the last level of recursion, the two vertices actually represent the vertices placed into the two sides of the cut.

**Lemma 1** Let \(C\) be some minimum cut of \(G\) with \(w(C) = k\). Then the probability that Karger’s contraction algorithm outputs \(C\) is at least \(1/\binom{n}{2}\).

**Proof:** We say \(C\) survives the \(i\)th level of recursion if no edge in \(C\) is contracted in recursive level \(i\). Then \(C\) is output as long as \(C\) survives every level. Thus

\[
\mathbb{P}(C \text{ is output}) = \mathbb{P}(C \text{ survives every recursive level}) = \prod_{i=0}^{V-2} \mathbb{P}(C \text{ survives level } i | C \text{ survived levels } 0, 1, \ldots, i - 1)
\]

where \(\mathbb{P}(A|B)\) is the conditional probability of event \(A\), given that event \(B\) occurred. Let \(m_i\) be the number of edges in our current graph \(G_i\) in recursive level \(i\) (note this graph is a multigraph: there can be parallel edges between vertices). Then the probability \(C\) survives recursive level \(i\) conditioned on it having already survived up
until this point is 1 − |C|/m_i = 1 − k/m_i. Since C has survived up until this point, the minimum cut of G_i is also still k. Thus the minimum degree of any vertex in G_i is at least k, since otherwise the cut separating a low-degree vertex from everyone else would have smaller cut value. Since m_i is just half the sum of all degrees, we have m_i ≥ n_k/2 = (n − i)/k. 2. Thus 1 − k/m_i ≥ 1 − 2/(n − i). Thus

$$\Pr(C \text{ is output}) \geq \left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \cdots \cdot \frac{1}{3} = \frac{n - 2}{n} \cdot \frac{n - 3}{n - 1} \cdot \frac{n - 4}{n - 2} \cdot \frac{n - 5}{n - 3} \cdot \cdots \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n - 1)}. \quad (4)$$

due to cancellation.

You might ask: can we improve the $1/(\binom{n}{2})$ success probability? The answer is no as the lemma is stated, since if there are t minimum cuts then one of them will be output with probability at most $1/t$. We can obtain $t = \binom{n}{2}$ by letting $G$ be the cycle on $n$ vertices (it has $\binom{n}{2}$ minimum cuts: namely choose any 2 of its $n$ edges to cut).

Now, $1/(\binom{n}{2})$ success probability seems low, but we can alleviate this by running the algorithm $\binom{n}{2}$ times and outputting the smallest weight cut we ever find. Then the probability that we never see a particular minimum cut is at most $(1 - 1/(\binom{n}{2}))^{\binom{n}{2}} \leq 1/e$, using our favorite approximation $1 + x \leq e^x$ for all real $x$. Thus if we repeat this $\lceil \ln(1/P) \rceil$ times for some $0 < P < 1/2$, the probability of never seeing a minimum cut is at most $e^{-\ln(1/P)} = P$.

We also leave it as an exercise to show that the contraction algorithm above can be implemented to run in time $O(n^2)$.

## 3 2SAT

Previously, we used the implication graph and strongly connected components to determine whether there exists a solution to 2SAT. Here, we will show another possible way to solve the 2SAT problem.

Recall that the input to 2SAT is a logical expression that is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \lor \overline{x_2}) \land (\overline{x_1} \lor x_3) \land (x_1 \lor x_2) \land (x_4 \lor \overline{x_3}) \land (x_4 \lor \overline{x_1}).$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied— that is, there is at least one true literal in each clause. For example, the assignment $x_1 = T, x_2 = F, x_3 = F, x_4 = T$ satisfies the 2SAT formula above.

Here is a simple randomized solution to the 2SAT problem. Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied or you get tired of flipping coins.

In the example above, when we begin with all variables set to F, the clause $(x_1 \lor x_2)$ is not satisfied. So we might randomly choose to set $x_1$ to be T. In this case this would leave the clause $(x_4 \lor \overline{x_1})$ unsatisfied, so we would have to flip a variable in the clause, and so on.

Why would this algorithm tend to lead to a solution? Let us suppose that there is a solution, call it $S$. Suppose we keep track of the number of variables in our current assignment $A$ that match $S$. Call this number $k$. We would like to get to the point where $k = n$, the number of variables in the formula, for then $A$ would match the solution $S$. How does $k$ evolve over time?
At each step, we choose a clause that is unsatisfied. Hence we know that $A$ and $S$ disagree on the value of at least one of the variables in this clause— if they agreed, the clause would have to be satisfied! If they disagree on both, then clearly changing either one of the values will increase $k$. If they disagree on the value one of the two variables, then with probability 1/2 we choose that variable and make increase $k$ by 1; with probability 1/2 we choose the other variable and decrease $k$ by 1.

Hence, intuitively, in the “worst case”, $k$ behaves like a random walk— it either goes up or down by 1, randomly. This leaves us with the following question: if we start $k$ at 0, how many steps does it take (on average, or with high probability) for $k$ to stumble all the way up to $n$, the number of variables?

We can formalize this idea of “worst case” as follows: let $y$ be a random walk on $[0, n]$, and suppose that $k$ and $y$ are changing in parallel and sharing the same random coins. In particular, $y$ goes left or right with probability 1/2, whereas $k$ either does the same as $y$ (i.e. the clause chosen disagrees on one variable) or goes right (i.e. the clause chosen disagrees on both variables). The only exception is if either variable is at 0, in which case we immediately move it to 1 before simulating the next step.

Then, $X$ and $Y$ be random variables. $X$ will be the number of steps in our simulation before we find a satisfying assignment (assuming one exists), plus the number of times $k$ is moved from 0 to 1. $Y$ will be the number of steps for $y$ to reach $n$, plus the number of times that $y$ is moved from 0 to 1.

We see that $X$ simulates the number of rounds for our algorithm to find a satisfying assignment. Then, we want to say that $E X \leq E Y$ so that we can use $E Y$ to get an upper bound on $E X$.

Why is this true? In particular, we can check by induction on the number of steps 0 that $k \geq y$. The base case is simply our starting scenario, where $k = y = 1$ (after the move). For the inductive step, assume that $k \geq y$ at the end of the previous step. Then, if $k$ moves left, so must $y$.

• If neither is moved right from 0 immediately after the next step, then we immediately obtain that $k \geq y$ at the end of that step.

• Otherwise, for a move right from 0 at the end of the next step to be possible, at least one of $k$ and $y$ must have been equal to 1 in the previous step.

  – If $k = 1$, then $y = 1$ as well because of our inductive hypothesis. If $k$ moves left, so must $y$, and they both end up at 1 again after having been moved away from 0. If $k$ moves right, then $k \geq y$ at the end of the next step regardless of whether $y$ moves right or left.

  – If $y = 1$ but $k \neq 1$, then if both move right it still holds that $k \geq y$. Otherwise, if $k$ moves left, then $k$ will end up back at 1 (after having been moved there from 0). Meanwhile, because $k$ was at least 2, it must be at least 1 after a move.

In all cases, we see that $k \geq y$ still after another step.

Hence, because $k \geq y$ after any number of steps, then $k$ must reach $n$ before $y$ or at the same time. As a result, for any fixed set of coin flips, $X \leq Y$, so when we take the expectation over all coin flips, it must therefore be true that $E X \leq E Y$.

Now, note that $Y$ is just equal to the number of steps for a random walk to get from 0 to $n$, where we remove the automatic move from 0 to 1 (but if $y = 0$, it is still only allowed to move right).

We can now check that $E Y = n^2$. In fact, we claim that the average amount of time to walk from $i$ to $n$ is $n^2 - i^2$.\n
Note that the time average time $T(i)$ to walk from $i$ to $n$ is given by:

\[
\begin{align*}
T(n) &= 0 \\
T(i) &= \frac{T(i-1)}{2} + \frac{T(i+1)}{2} + 1, \quad i \geq 1 \\
T(0) &= T(1) + 1.
\end{align*}
\]

These equations completely determine $T(i)$, and our solution satisfies these equations!

Hence, on average, we will find a solution in at most $n^2$ steps. (We might do better – we might not start with all of our variables wrong, or we might have some moves where we must improve the number of matches!)

We can run our algorithm for say $100n^2$ steps, and report that no solution was found if none was found. This algorithm might return the wrong answer – there may be a truth assignment, and we have just been unlucky. But most of the time it will be right: Markov’s inequality gives that if we run the algorithm for $cn^2$ steps for $c \geq 1$, then it will be wrong at most $1/c$ of the time.