COMPSCI 330: Design and Analysis of Algorithms 10/21/2014 and 10/23/2014

Lecture #15 and #16

Lecturer: Debmalya Panigrahi *Scribe: Nat Kell and Ang Li*

1 Introduction

In this lecture, we will cover *amortized analysis*, which is a method where we consider the average work done over an entire sequence of operations instead of just considering the worst-case cost/time for a single operation; by doing such an analysis, we can obtain tighter bounds in scenarios when expensive operations occur rarely enough that we can charge their cost to more frequent inexpensive operations. We cover such charging arguments as well as the *potential method*, which is another powerful tool for doing amortized analysis. We will apply both these techniques to *binary counting*, the *union-find* data structure (which is needed to implement Kruskal's algorithm efficiently), and *dynamic arrays*.

2 Binary Counting

2.1 Amortized Analysis

Suppose we wanted to increment a variable x starting at 0 until we reach some value n, where we represent x as a binary number with $\lfloor \log n \rfloor + 1$ bits (we'll denote this binary representation as $\text{BIN}(x)$). Using the basic carry addition algorithm (that we all know so well from primary school), we implement each x++ as follows. Starting at the least significant bit (LSB), if we observe a 1 we flip it to 0 and then move to the next bit to the left. Otherwise if we observe a 0, we flip it to a 1 and stop (for instance, if we add 1 to 10100110*1111111*, we obtain 10100110*1000000*, where the digits in italics are the bits that get flipped).

Over all n increments, we can ask the question: *how many bit flips do we do in total?* Clearly, the number of flips for a single increment is $O(\log n)$ since there are only $\lfloor \log n \rfloor + 1$ bits, and since we do n increments in total, a rough upper bound on the total flips is $O(n \log n)$. However, it seems like we should be able to tighten this aggregate bound since on many increments we are only flipping a few bits; ideally, we would like to show a bound of $O(n)$, which would establish that, even if the worst-case bound of a single increment is $\Theta(\log n)$, on average we are only doing a constant amount of work for each operation.

We can indeed obtain an $O(n)$ bound with the following "direct analysis." Observe that the kth LSB (i.e., 0th least significant bit is the right most bit) can be flipped from 1 to 0 and back to 1 at most $n/2^k$ times. When this event happen, the counter has increased by 2^k, and since our final total is n, it follows that these two flips can occur at most $n/2^k$ times. Summing over all bit positions k and using the closed form for an infinity geometric series we obtain:

$$\text{Total bit flips} = \sum_{k=0}^{\lfloor \log n \rfloor + 1} \frac{n}{2^k} < n \cdot \sum_{k=0}^{\infty} \frac{1}{2^k} = 2n = O(n).$$

2.2 Charging Arguments

This analysis is similar to arguments we have seen previously this semester: For each possible index for a bit flip, we argued an upper bound on the number of times this type of flip/event can happen and summed

these upper bounds over all possible positions to obtain the desired upper bound for the total number of flips. Now, we are going to perturb this analysis somewhat and instead frame it as a *charging argument*. The idea is that instead of analyzing each operation directly, we distribute or *charge* the work done on expensive operations onto inexpensive operation and then sum over all the work done over all operations with this new work distribution.

To make this more concrete, consider the following charging scheme for binary counting. First observe that:

1. On a given increment, there is exactly one 0-1 flip (the last flip we perform); therefore, there are exactly n 0-1 flips in total.

2. Each 1-0 is proceeded by a unique 0-1 flip.

Therefore, we *charge every 1-0 flip the preceding 0-1 flip*. By Observation 2, a given 0-1 flip is charged exactly once; thus now, we have distributed the work so that 1-0 flips cost nothing and 0-1 flips cost 2—one unit of work for doing the actual flip and then another unit of work from the charge the proceeding 1-0 flip is now placing on this operation. Even though we have made 0-1 flips more expensive, we have only done so by a constant amount. By Observation 1, there are only n 0-1 flips in total, and therefore using this new work distribution we conclude that the number of flips is at most $2n = O(n)$.

2.3 Potential Functions

Another method for doing amortized analysis is using what are called *potential functions*. Broadly defined, a potential function $\Phi(i)$ maps the state of an algorithm or data structure after the ith operation to a carefully defined non-negative value. The naming convention of calling $\Phi(i)$ a "potential" comes from the fact that the change in $\Phi(i)$ is somewhat analogous to the state system accumulating potential energy and then releasing it later as kinetic energy. On an inexpensive operation i, $\Delta\Phi = \Phi(i) - \Phi(i-1)$ will likely be positive and thus we build potential energy in the system. Then on expensive operations, we release potential energy as kinetic energy by making $\Delta\Phi$ negative, which will counterbalance the higher cost. Another analogy that is often used is that the potential function is a bank account where we save money on inexpensive operations and spend the money we have saved on expensive operations.

More specifically, let c_i be the cost of operation i (so in the case of the binary counter, this is the number of bits flipped on the ith increment). Given a potential function Φ, we define the *amortized cost* $\hat{c}_i$ of operation i to be $c_i + \Phi(i) - \Phi(i-1)$. Observe that we have the following bound if we perform n operations:

$$\text{Total amortized cost} = \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(i) - \Phi(i-1)) = \sum_{i=1}^{n} (c_i) + \Phi(n) - \Phi(0). \tag{1}$$

The last equality follows since all the $\Phi(i)$ terms in the summation *telescope* except for $\Phi(0)$ and $\Phi(n)$ (i.e., for each $j = 1, \ldots, n-1$, the positive term $\Phi(i)$ when $i = j$ will cancel with the negative term $-\Phi(i-1)$ when $i = j+1$). Therefore, as long as $\Phi(0)$ and $\Phi(n)$ are non-negative, then amortized cost is an upper bound on actual cost (since actual cost is just $\sum_{i=1}^{n} c_i$).

As we mentioned for our binary counter example, c_i is just the number of bit flips on the ith increment. To define our amortized cost, we will use the following potential function:

$$\Phi(i) = \text{The number of 1s in } \textsc{bin}(x) \text{ after } i \text{ increments.} \tag{2}$$

To do our analysis, consider the difference between the number of 1s in x before and after the ith increment; namely, we will define $\Phi(i)$ in terms of $\Phi(i-1)$. Observe that based on the algorithm, there

are $c_i - 1$ 1s that become 0s on the ith increment: the first $c_i - 1$ are 1s being flipped to 0, and then the last unit of cost is added from flipping the last 0 to a 1. Since this last flip also adds an extra 1 to x, the total number of 1s after the ith increment is $\Phi(i) = \Phi(i-1) - (c_i - 1) + 1$. Therefore, the total amortized cost is as follows:

$$\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(i) - \Phi(i-1)) \\
&\leq \sum_{i=1}^{n} (c_i + (\Phi(i-1) - c_i - 1) + 1 - \Phi(i-1)) \qquad \text{(since } \Phi(i) = \Phi(i-1) - (c_i - 1) + 1\text{)} \\
&= \sum_{i=1}^{n} 2 = 2n.
\end{aligned}$$

Clearly, both $\Phi(0)$ and $\Phi(n)$ are non-negative (we cannot have a negative number 1s in $\text{BIN}(x)$); therefore by (1), $2n = O(n)$ is an upper bound on the total actual cost, as desired.

To get some intuition as to how the potential function is working, observe that whenever we only flip the first bit (i.e., we are incrementing an even number to an odd number), the amortized cost is 2 instead of just 1—these are the operations where we save potential. For any increment where we flip three or more bits, there is a drop in potential where we spend the money we saved from each of the odd increments. Note that such operations can charge the potential much more than just a cost of 1 (it can be as high as $\log(n)$); however, since we are saving potential on half of the increments, we always have enough money in the potential's bank account to lower the cost of every operation down to just 2.

3 Revisiting Kruskal's Algorithm: Union-Find Data Structure

3.1 Storing Components for Kruskal's Algorithm

For a weighted graph $G = (V, E)$ where w_e denotes the weight of edge $e \in E$, recall Kruskal's algorithm for computing a minimum spanning tree (MST) of G (if you are having trouble remembering the MST problem or Kruskal's algorithm, you should go back and review the notes for Lecture 8). At a high level, we begin Kruskal's algorithm by initializing each vertex to be in its own component. Then in order of increasing edge weight, we repeatedly add edges to the tree if they merge two of the current components together (i.e., if $e = (u, v)$ is the edge we are considering, we add edge e to our tree if u is currently in a different component than v). Once all vertices lie in the same component, we argued that the resulting structure does in fact give us a MST.

However, when we previously outlined the pseudo-code for Kruskal's, we glossed over how to represent these collections of vertex sets in memory. On the iteration where we consider adding edge $e = (u, v)$, we need to quickly find out if u and v belong to the same component, and if they do not, we need to merge these components together.

To perform such queries and operations, we will implement a *union-find* data structure. A union-find data structure D is defined over a set of n elements $U = \{x_1, \ldots x_n\}$ and maintains a collection of disjoint subsets $S_1, \ldots S_h$ to which these elements belong, where $1 \leq h \leq n$. As in our scenario above, every element is in its own subset when D is initialized. D then supports the following two operations:

- $\text{FIND}(x)$: return S_i such that $x \in S_i$ (in an actual implementation, we would likely just return the representative element for set S_i).

- UNION(S_i, S_j): Replace S_i and S_j with $S_i \cup S_j$ in the set system.

So for Kruskal's algorithm, we initialize a union-find data structure over the vertices. For each edge $e = (u, v)$, if FIND$(u) \neq$FIND(v), then we call UNION(FIND(u), FIND(v)) to merge u and v's components (otherwise, we move on to the next edge). In total we will issue at most $2m$ FIND queries and always perform n UNION operations.

3.2 Implementing Union-Find

We now turn to the details of implementing FIND(x) and UNION(S_i, S_j) efficiently. Our first implementation decision is to how to represent the "labels" for each set. Here, we will use elements as representatives: At any given time, there will be a unique $x \in S_i$ which we will return as the label of S_i whenever we call FIND(y) for any $y \in S_i$ (in the following implementations, we will make it clear how each representative is determined/maintained).

3.2.1 Union-Find with Linked Lists

The most obvious way to represent the set system is to just use a collection of linked lists. For each set S_i, we have a corresponding linked list L_i which contains the elements in S_i. The representative of L_i will just be element at the head of the list, which is then preceded by the the rest elements in S_i through a sequence of pointers. To execute FIND(x), we start at x and follow the path of pointers leading to the head and then return it as the label of the set. Note that since there can be $\Omega(n)$ elements in a set, we might have to traverse $\Omega(n)$ links to reach a set's label; therefore when using linked list, FIND(x) runs in $\Theta(n)$ time in the worst case. UNION operations, however, are quite simple. To implement UNION(S_i, S_j), we just make the head of L_i point to the tail of L_j (or vice versa). Since we can store head/tail metadata along with the head of a list, UNION is an $O(1)$ time operation.

As noted above, a given run of Kruskal's may do $2m$ FIND(e) queries, which gives us a $\Theta(mn)$ time algorithm in the worst case. When we first presented Kruskal's algorithm, we claimed a running time of $O(m \log n)$; therefore, using this linked list implementation will not suffice.

3.2.2 Union-Find with Trees

If we want to maintain the property that UNION operations still take $O(1)$ time, a natural improvement to this linked list scheme is to instead maintain a set of trees. Now, a set S_i corresponds to a tree T_i, where the representative of the set is at the root. To implement UNION(S_i, S_j), we make T_i a subtree of T_j by making the root of T_j the parent of the root of T_i. Note that this implies that each tree is *not* necessarily binary since a fixed root r can participate in several UNION operations (it is possible that each UNION results in another subtree rooted at r).

FIND(x) still works in the exact same way—we simply start at x and follow a path up the corresponding tree via parent pointers until we reach the root. Our hope is that if each tree structure remains balanced, then we can bound the longest path from node to root when doing a FIND query. However, our current

specifications do not ensure balance. For example, consider the sequence of n unions

$$\{x_1\} \cup \{x_2\}$$
$$\{x_3\} \cup \{x_1, x_2\}$$
$$\{x_4\} \cup \{x_1, x_2, x_3\}$$
$$\vdots$$
$$\{x_n\} \cup \{x_1, \ldots, x_{n-1}\}.$$

Informally, we grow one particular set in the set system, and then with each UNION we add one of the remaining singleton sets to this growing set. When we perform $\text{UNION}(S_i, S_j)$ in this scheme, note that we are arbitrarily picking which root (the root of T_i or the root T_j) becomes the new root when we combine T_i and T_j. Thus in the above example, it is possible that when we merge $S = \{x_i\}$ with $S' = \{x_1, \ldots, x_{i-1}\}$, we use x_i as the new root each time. If we are unfortunate enough to have this sequence of events happen for each union, then the resulting tree structure will just be an n element linked list (and therefore it is still possible for $\text{FIND}(x)$ to take $\Omega(n)$ time).

A straight forward way to fix this pitfall is to do what is called *union-by-depth*. For each tree T_i, we keep track of its depth d_i, or the longest path from the root to any node in the tree. Now when we perform a UNION, we check to see which tree has the larger depth and then use the root of this tree as the new root. Note that this extra information can be easily stored and updated with the root of each tree: If we call $\text{UNION}(S_i, S_j)$ and $d_i \leq d_j$, then the root of T_j becomes to root of $T_i \cup T_j$, and we update the depth of $T_i \cup T_j$ to be $\max(d_j, d_i + 1)$ (note this max is only necessary in the case where $d_i = d_j$—otherwise, the depth of the combined tree is no larger than depth of T_j).

What does "union-by-depth" buy us? The following theorem establishes that this feature does indeed balance the trees in the set system.

Theorem 1. *For a tree implementation of the union-find data structure that uses union-by-depth, any tree T (representing set S_i in the set system) with depth d contains at least 2^d elements.*

Proof. We do a proof by induction on the tree depth d. Since a tree T with depth 0 has has $2^0 = 1$ elements, the base case is trivial. For the inductive step, assume that the hypothesis holds for all trees with depth $k-1$, i.e., any tree with depth $k-1$ contains at least 2^{k-1} nodes. Observe that in order to build a tree T with depth k, we must merge together two trees T_i and T_j that both have depth $k-1$; otherwise, we would either have:

1. Both T_i and T_j have depth strictly less than $k-1$. Since the depth of $T_i \cup T_j$ can be no more $\max(d_j, d_i) + 1$, the combined tree $T_i \cup T_j$ can have depth at most $k-1$ (note this is true regardless of whether we use union-by-depth).

2. Exactly one tree has depth $k-1$; without loss of generality, suppose $d_j = k-1$ and $d_i < k-1$. Since we are using union-by-depth, we will make the root of $T_i \cup T_j$ the root of T_j. Since $d_i < k-1$, the length of any path from this new root of to any node in T_i can be at most $k-1$. Since T_j has depth $k-1$ and no node in within this subtree changes depth in $T_i \cup T_j$, the depth of the combined tree is exactly $k-1$.

Therefore, assume $d_i = d_j = k-1$; we can then apply our inductive hypothesis to both T_i and T_j to obtain:

$$|T| = |T_i \cup T_j| = |T_i| + |T_j| \geq 2^{k-1} + 2^{k-1} = 2^k,$$

as desired.

□

Theorem 1 implies that any tree with n elements can have depth at most $\log n$ (the theorem implies $n \geq 2^d$ where d is the depth of the largest tree/subset, implying $\log n \geq d$). Therefore, FIND(x) runs in $O(\log n)$ when using union-by-depth. From Kruskal's perspective, this gives us the desired running time. The initial sort we do on the edge weights takes $O(m \log m) = O(m \log n^2) = O(m \log n)$ time. We then do n UNIONs that each take $O(1)$ time and $2m$ FINDs that each take $O(\log n)$ time. Therefore, the overall running time of Kruskal's using this implementation is $O(m \log n) + O(n) + O(m \log n) = O(m \log n)$.

3.2.3 Union-Find with Stars

Although doing a tree implementation that uses union-by-depth gave us the desired asymptotic running time of $O(m \log n)$, it is a bit unsettling that UNIONs take constant time and FINDs could take $\Omega(\log n)$ time. Since $n = O(m)$ for any graph where we want to find a spanning tree, it seems a bit wasteful that our implementation gives us a faster running time for the function we call fewer times (recall we perform n UNIONs and at most $2m$ FINDs). Therefore in this section, we will look at an implementation where we force each FIND to take $O(1)$ time, but as a result make UNIONs operations more expensive (but hopefully by not by too much).

The most naive way to achieve $O(1)$-time FINDs is to represent sets as *star graphs*. A star graph is simply a tree with a designated a center node such that every other node in the graph is a leaf that is only adjacent (or points) to this center node. Thus, we will maintain that each tree T_i that represents a set S_i is a star graph, where the center node of T_i is the representative of S_i. Clearly with this scheme, when we call FIND(x) we must only traverse at most 1 link to reach the representative node, and therefore the running time of FIND(x) is $O(1)$.

However to maintain this star graph structure, we will need to take more time when we make a UNION call. If we have two star graphs T_i and T_j that we want to merge, we first need to pick which representative element we will use for $T_i \cup T_j$ (just like for our previous implementation with balanced trees). If we pick T_i's center c_i to be the new center, we then need to iterate through every element $x \in T_j$ and make x point to c_i. Since T_j could have $\Omega(n)$ elements, this operation could take $\Omega(n)$ time. Therefore if we do n UNION operations, our running time for Kruskal's is now $\Theta(n^2)$ (which could be worse than $O(m \log n)$).

To avoid this problem, we will use a rule that is similar to union-by-depth. Namely, we will use *union-by-size*. Namely, if we are given two star graphs T_j and T_i, we will dissemble the smaller of the two sets and make these elements point to the center of the larger set (and leave the star graph in the larger graph untouched).

To analyze the speedup obtained from doing union-by-size, we use a charging argument to do an amortized analysis over the n UNIONs performed by Kruskal's. We use the folioing charging scheme: Any time we merge two trees T_s and T_ℓ such that $|T_s| \leq |T_\ell|$, we will simply put a unit of charge on each element in T_s (remember that we are taking the elements of T_s and changing their pointers to the center of T_ℓ). Note that for all $x \in T_s$, x now belongs to a set that is twice as large. We also know that for $x \in U$, the set to which x belongs can double at most $\log n$ times (the size of final merged set is n); therefore, the charge on a given element x can be at most $\log n$ after n unions. Since the total time needed over all n unions is equal to the total charge distributed over the elements, the time it takes to make n UNION calls is $O(n \log n)$. Note that even though Kruskal's algorithm still runs in $O(m \log n)$ time since we must initially sort the edges, we have reduced the time it takes to execute Kruskal's merging procedure to $O(n \log n + m)$.

3.2.4 Optimal Union-Finds: Path Compression and Union-by-Rank

We will now outline the best scheme for implementing a union-find data structure. This implementation will be more akin to what we saw in Section 3.2.2 when we used balanced trees to represent our set system. The main feature we will add to this implementation is what is known as *path compression*, which will attempt to make our trees more "star-graph-like" whenever we make a FIND call (so in some sense, we are combining the strategies in sections 3.2.2 and 3.2.3).

More specifically, whenever we call FIND(x) where $x \in S_i$, we will follow some path P from x to the root r_i of S_i. For each element $y \in P$, we now know that y belongs to set S_i; therefore at this point, it makes sense to make each of these elements point directly to r_i. FIND(x) with path compression does exactly this modification, and therefore after the procedure completes, r_i and all the elements along P now form a star graph in T_i. Note that it is not too hard to implement FIND(x) such that it returns r_i, makes every element in P point directly to r_i, and runs in $O(|P|)$ time.

To implement UNION, we essentially still use union-by-depth. We still merge components using the same rule (we make the tree with the smaller depth a subtree of the tree with the larger depth). Note, however, that because we might have path compressing FIND calls in-between UNION calls, we might compress a path that defined the depth a given tree T_i. In such a case, d_i no longer accurately stores the depth of d_i.

How does one fix this issue? The answer is that we do not. Instead, we just call this d_i the *rank* of T_i and use it in the same way we would in our union-by-depth scheme. It turns out that using these two features in combination gives us an extremely good bound over n UNIONs and $2m$ FINDs. Clearly, the n UNION operations still take $O(n)$ time since each UNION call takes $O(1)$ time. Using an advanced amortized analysis, one can show that the $2m$ FIND(x) queries take $O(m \cdot \alpha(n))$ time, where $\alpha(\cdot)$ is the inverse of the *Ackermann's function*[1]. Although $\alpha(n)$ is unbounded as n tends to infinity, it grows incredibly slowly. For example, for all $n \leq 10^{80}$ (which is more than the number of atoms in the observable universe), $\alpha(n) \leq 4$. Therefore, for all practical purposes we can treat $\alpha(n)$ as a constant (however we still express the running time as $O(2m \cdot \alpha(n))$ to be mathematically correct).

Dynamic Table is a data structure that resizes itself according to the number of elements present. As a result, it supports arbitrarily large number of elements.

4 Dynamic Table

4.1 An Insert Operation on a Dynamic Table

The following pseudo-code describes the operation to insert the i-th element into the Dynamic Table.

Insert(i):
    if there's space, insert.
    if space=$i-1$, $(i = 2^k + 1)$
        allocate new space $2(i-1)$
        copy current table over

Total time:

[1] In lecture, we spent some time defining Ackermann's function and outlined some intuition for why it grows so quickly. You will not be responsible for knowing Ackermann's function offhand, so we will omit this discussion from these lecture note; however, we encourage the reader to visit the Wikipedia page on the subject: `http://en.wikipedia.org/wiki/Ackermann_function`

$O(1)$ if there is space
$O(i)$ if new space allocation is required
Hence, total time to insert n elements $= 1+2+4+...+n = O(n^2)$

4.2 Charging Scheme Analysis

1. If each element charged 1 when it is copied,
worst-case charge on an element = $O(\log n)$.
Total charge = $O(n \log n)$

2. If each element in the second half of the current table is charged 2,
each element is charged at most once.
Total charge $\leq 2n = O(n)$

4.3 Potential Function Analysis

Potential function to use: $\phi = 2 * (\#\ of\ elements) - size\ of\ table$
1. $\phi_0 = 0$
2. $\phi_i \geq 0$
$A_i = c_i + \phi_i - \phi_{i-1}$
When there's no copying: $A_i = 1+2 = 3$
When there's copying: $A_i = 1+i+2-(i-1) = 4$
Hence, $4n \geq \sum_i A_i = \sum_i c_i + \phi_n\ (at\ least\ 0) - \phi_0\ (equals\ 0)\ \geq \sum_i^n c_i$

4.4 If deletion is allowed

We can double the size of the array when the array is full, and half the size of the array when the array is less than one quarter full. An amortized $O(1)$ time for insert or delete operation is achieved this way.