Lecture 1, Sep 11, 2023

Runtime Complexity Analysis (Review)

Definition

Let A be an algorithm and $t(x)$ be the number of steps taken by A on input x; then the *worst-case time complexity* is

$$T(n) = \max_{\text{all input } x \text{ of size } n} t(x) = \max\{\, t(x) \mid x \text{ is of size } n \,\}$$

- "Size" is typically defined as the number of bits used to represent the input; e.g. number of elements in an array, number of nodes/edges in a graph, number of bits of an integer
- To define an upper bound for $T(n)$, we have to prove that for *every* input of size n, A takes *at most* some number of steps
- To define a lower bound for $T(n)$, we only have to prove that for *some* input of n, A takes *at least* some number of steps

Definition

$T(n)$ is $O(g(n))$ iff

$$\exists c > 0, \exists n_0 > 0, \text{ s.t. } \forall n \geq n_0, T(n) \leq c \cdot g(n)$$

In other words, for sufficiently large input and within a constant factor: for *every* input of size n, A takes *at most* $c \cdot g(n)$ steps.

Definition

$T(n)$ is $\Omega(g(n))$ iff

$$\exists c > 0, \exists n_0 > 0, \text{ s.t. } \forall n \geq n_0, T(n) \geq c \cdot g(n)$$

In other words, for sufficiently large input and within a constant factor: for *some* input of size n, A takes *at least* $c \cdot g(n)$ steps.

Definition

$T(n)$ is $\Theta(g(n))$ iff it is both $O(g(n))$ and $\Omega(g(n))$.

- The notions of O, Ω, Θ allow us to ignore constant factors and restrict the analysis to sufficiently large input sizes

Lecture 2, Sep 13, 2023

Definition

An *abstract data type* (ADT) describes an object and which operations you can apply to it.

A *data structure* is a particular implementation of an ADT.

Max-Heaps

Definition

Given a set of S elements with keys (i.e. priority) that can be compared, a *priority queue* has the operations:

- INSERT(S, x): insert element x in S
- MAX(S): returns an element of highest priority in S (note there may be multiple elements with the same priority)
- EXTRACTMAX(S): returns the max element and removes it from S

- Priority queues can be implemented in a variety of ways, e.g. with linked lists or sorted arrays, but max-heaps are a particularly efficient implementation that offers $\Theta(\log n)$ operations

Definition

In a (binary) *max-heap* of n elements, the elements are stored in a complete binary tree such that the max-heap property holds, i.e. the priority of each element is greater than or equal to all its children.

- Note: in a complete binary tree, all levels are filled except the last one, and the last level is filled starting from the left
  - The height of a complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$
- For a given sequence of elements, there is no unique max-heap configuration
- Max-heaps (or in general any complete binary tree) can be stored very efficiently in a simple array by laying out its elements from top to bottom and left to right
  - If an element is at index i, then its children will be at indices $2i$ and $2i+1$ respectively (note we're using 1-based indexing; with zero-based indexing, the children will be at $2i+1$ and $2i+2$)
    - To go back to the parent, we take $\left\lfloor \frac{i}{2} \right\rfloor$
  - The size of the heap is tracked so we know where the array ends
- To insert an element, we add the element at the end, increase the heap size, and restore the heap property
  - To restore the heap property, compare the current element with its parent, and if it has higher priority, then swap the two elements; repeat all the way until the element no longer has greater priority than its parent, or it has reached the root
  - Since we have to do this for at most all levels of the tree, this operation is $\Theta(\log n)$ (since the tree's height is $\lfloor \log n \rfloor$)
- To get the max element, we simply return the root of the tree, which is a constant time operation
- To extract the max element, we extract the root of the tree and replace the now empty root with the last element, and restore the heap property
  - To restore the heap property, compare the current element with its children, and if it is smaller than any of its children, swap it with the bigger child; repeat until the element is bigger than its children, or becomes a leaf
  - This again goes through at most all levels of the tree, making it $\Theta(\log n)$ complexity
- Some example applications:
  - HeapSort: take an array, make it a heap, and extract the max n times until the array is empty; this makes for a simple $\Theta(n \log n)$ sorting algorithm

Lecture 3, Sep 18, 2023

Binomial Heaps

- What if we had 2 priority queues, and we wanted to merge them in an efficient way?
  - Using a binary heap, this would take $n \log n$ time; can we find a way to do this in $\log n$ time?

- We will present a new data structure, a binomial heap, which can do all operations in $O(\log n)$ time (at the cost of having $O(\log n)$ time to view the max element, as opposed to constant time)

Definition

A *binomial tree* is recursively defined as follows:

1. A binomial tree of height 0, B_0 consists of a single node
2. A binomial tree of height k, B_k consists of two binomial trees B_{k-1}, where one becomes the parent of the other

Alternatively, the root B_k has k children, with the first child being a binomial tree B_0, the second being B_1, etc. until the k-th child B_{k-1}.

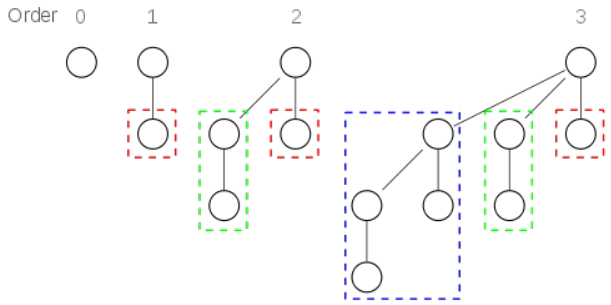

Figure 1: Example binomial trees.

- A binomial tree B_k has:
  - Height k
  - 2^k nodes
  - $\binom{k}{d}$ nodes at depth d (hence why it's called a binomial tree)

Definition

A *binomial forest* of size n is denoted F_n, which contains a sequence of B_k trees with strictly decreasing ks and n nodes in total.

- Example: if we want a forest F_7, we can decompose it into 3 binomial trees with 4, 2, and 1 nodes respectively, so the forest contains B_2, B_1, B_0
  - This is equivalent to expressing n in binary
  - e.g. $9 = 1001_2 = 2^3 + 2^0$ so F_9 contains B_3, B_0
  - Let $\alpha(n)$ be the number of 1s in the binary representation of n, then F_n has $\alpha(n)$ trees
- F_n has $n - \alpha(n)$ edges (since each node comes with 1 edge, except for roots, and there are $\alpha(n)$ trees)

Definition

A *min-binomial heap* of n elements is a binomial forest F_n such that each binomial tree B_k within the forest satisfies the min-heap property, i.e. a parent is smaller than both its children.

- A binomial heap of N elements can be built in $O(n)$ operations from scratch
  - One key comparison is needed per binomial heap edge
- But how do we actually put this in memory?
  - Different nodes have different number of children, so using pointers directly will be inefficient

  - For every node, we will have 3 pointers: one to its parent, one to its leftmost child, and one to its next sibling
    * To access the children of a node, we access the leftmost child first, and then go through the chain of sibling pointers
    * The sibling pointers of root nodes connect between different trees, pointing to the next bigger B_k tree
  - Finally, we need to keep track of the head of the data structure, which is a pointer to the smallest B_k tree
  - All the pointers make the binomial heap much less efficient in memory than binary heaps
- Two important properties to note:
  - We can merge two min-heap ordered B_k trees into a single min-heap ordered B_{k+1} tree with just a single key comparison, by making the tree with the smaller root the parent tree
  - Deleting the root of a min-heap ordered B_k tree gives a min binomial heap, since the children are binomial trees $B_0, B_1, \ldots, B_{k-1}$
- We can now implement the $\textsc{Union}(T, Q)$ operation, which merges two heaps
  - The operation works like binary addition:
    * Start with the B_0 trees in both heaps
    * If there is a B_k in one heap but not the other heap, simply add this tree to the result
    * If there are B_k trees in both heaps, merge them into a B_{k+1} tree and make it a "carry"; merge this carry with the existing B_{k+1} trees of the two heaps
  - Since each B_k tree merge takes constant time and both input heaps have at most $\log n$ trees, the complexity is $O(\log n)$
- An insert operation is equivalent to merging with a heap of size 1
- For $\textsc{Min}(T)$, we have to scan the root of every B_k tree; this gives us $O(\log n)$ complexity
- For $\textsc{ExtractMin}(T)$, find the B_k root that contains the minimum and remove it; this gives us another binomial heap made of its children, so we can simply merge it into the original heap

Lecture 4, Sep 20, 2023

Binomial Heaps Continued

- Another operation is $\textsc{DecreaseKey}(T, x, k)$ which decreases the priority of x to k
  - This can be implemented simply in $O(\log n)$ time in both binomial and binary heaps by moving the element up
- $\textsc{Remove}(T, x)$ removes the element x; this can also be implemented in $O(\log n)$ time for binomial heaps
- Even though a single Insert operation takes $O(\log n)$, inserting k keys in sequence has a cost of at most $2k$ key-comparisons as long as $k > \log n$

Dictionaries

Definition

Given a set S of elements with keys, a *dictionary* provides the operations:
- $\textsc{Search}(S, x)$: returns element with key x if it is in S, else "not found"
- $\textsc{Insert}(S, x)$: inserts x in S
- $\textsc{Delete}(S, x)$: deletes x from S

- If we naively implement this with a linked list, we get $\Theta(1)$ insert, but $\Theta(n)$ search and delete; can we find a structure that does each operation in $\Theta(\log n)$ time?
  - We can try a binary search tree

Definition

A *binary search tree* is a binary tree in which for each node, all the keys in its left subtree are smaller or equal to itself, and all the keys in its right subtree are bigger or equal to itself.

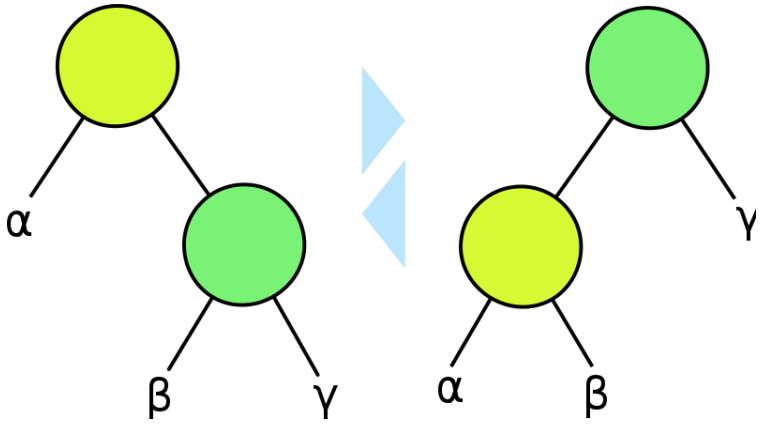

Figure 2: Visualization of binary tree rotations. From the left diagram to the right diagram is a *left* rotation; from right to left is a *right* rotation.

- Given a node in the BST, we can *rotate* it:
  - Left rotation: the node goes "down" and the right child is "lifted up"
    - The right child becomes the new parent; the original node becomes its left child
    - The original left subtree of the right child now becomes the right child of the original parent
  - Right rotation: the node goes "down" and the left child is "lifted up"
    - The left child becomes the new parent; the original node becomes its right child
    - The original right subtree of the left child now becomes the left child of the original parent
- BSTs can be searched recursively: given a key to search for, if it's bigger than the current node, search the right subtree; if it's smaller than the current node, search the left subtree; if it's equal to the current node, return the current node
- Insert works in the exact same manner as search to find a parent node that has an empty space
- Deletion has a few cases:
  - If the node to be removed is a leaf, simply remove it
  - If the node has one child, remove it and replace it with its child
  - If the node has both children, replace it with its *successor*, which is the leftmost node in its right subtree; the successor will either have 1 child or no children
- For all operations in a BST, the worst-case runtime is directly proportional to its height
  - For an unbalanced BST, in the worst case the height is the same as its size; however if we keep the tree balanced, we can do all the operations in $O(\log n)$ time

Lecture 5, Sep 25, 2023

Balanced BSTs – AVL Trees

- We have learned previously that the runtime complexity of BST algorithms are dependent on the tree height, so in the worst case they can run in $O(n)$ time for an extremely unbalanced tree
  - In the best case the height scales as $\Theta(\log n)$; how can we achieve this?
- Note height is the number of edges in the longest path from a node to a leaf; the height of a tree is defined as the height of the root
  - A trivial tree with only a root node has a height of 0
  - We shall define the height of an empty tree as -1

Definition

The *balance factor* (BF) of a tree is the height of its right subtree minus the height of its left subtree.

In an *AVL tree*, the balance factor of every node is between -1 and +1, inclusive.

- Intuitively we would like the BF of every node to be close to 0, but we can't quite get there since the tree isn't complete
- We can get pretty close by using an AVL tree (named after Adelson-Velski-Landis)
- If the BF is less than 0, then we say it is left-heavy; if BF is greater than 0, it is right-heavy; otherwise it is balanced
- Properties of AVL trees:
  - An AVL tree of n nodes has height $\Theta(\log n)$ (less than $1.44 \log_2(n+2)$)
  - Given an AVL tree, we can do inserts and deletes while maintaining the tree's balance in $\Theta(\log n)$ time
- Practically, to store each node in memory we have a pointer to the parent and both children, as well as a key and balance factor
- For $\textsc{Search}(T, x)$ there is no difference from the regular BST search
- For $\textsc{Insert}(T, x)$, first insert as in a regular BST, and then apply a series of rotations to restore balance
  - General idea: by inserting a node, the balance factor of all its parents will potentially change by 1
    * We update the balance factors by going from the newly inserted node back to the root
      - If we're coming from a right child, we increase the BF by 1; if we're coming from a left child, we decrease it by 1
    * If now the balance factor is no longer in the range $[-1, 1]$, then we rebalance by applying rotations
  - Note if we change a balance factor to 0, it does not actually change the height of the node or any of its parents (since we just balanced it), so we can stop going up the chain
  - If we encounter a right-heavy node with a right-heavy right child, we will left-rotate
    * After a rotation, the BF of the nodes that are not involved in rotations remain the same
    * The node that we applied the rotation to will have its BF reduced by 2; its former right child will have its BF reduced by 1
    * After a rotation, the height of the tree does not change, so we can stop going up the tree to update BFs
  - If we encounter a right-heavy node with a left-heavy right child, we will first right-rotate on its right child (to make the right child right-heavy), and now left-rotate on the parent
  - In the symmetric cases of left-heavy node, left-heavy left child, or left-heavy-node, right-heavy left child, the operations are symmetric

Lecture 6, Sep 27, 2023

AVL Trees Continued

- In general for insertion, to rebalance there are 4 distinct cases:
  - Left-heavy node with left-heavy left child: rotate right on node
  - Left-heavy node with right-heavy left child: rotate left on left child, then rotate right on node
  - Right-heavy node with right/left-heavy right child: symmetric to the above cases
- In all 4 cases we can prove that the BST property is kept, the BFs are restored, and the height of the balanced tree is the same as before when the new node was inserted
- To insert:
  - Insert the node into the tree; set its balance factor to 0
  - Go from the node to the root, and for each node in the path:
    * Adjust the BF: if coming from the left, decrease BF, if coming from the right, increase BF
      - If the BF becomes 0, stop going up the tree

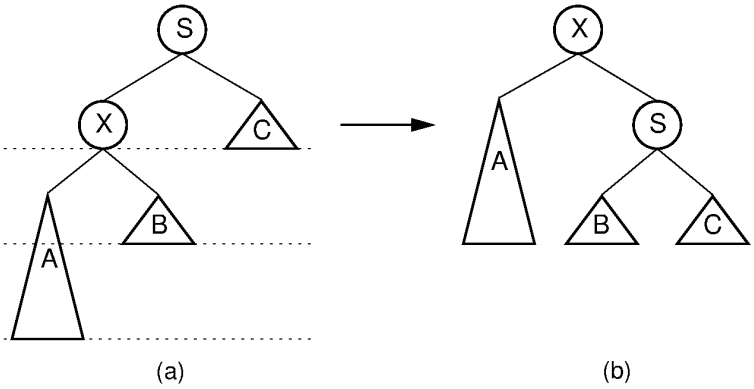

Figure 3: AVL tree, left-heavy node with left-heavy child.

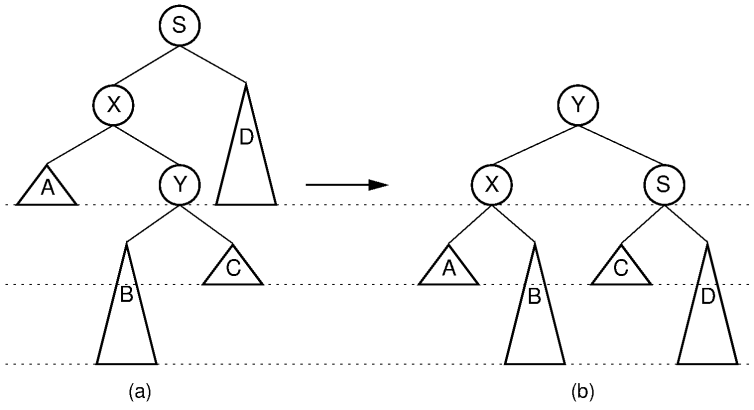

Figure 4: AVL tree, left-heavy node with righ-heavy child.

        * If the BF is outside the range $[-1, 1]$, rebalance as one of the cases above
            - After rebalancing, stop going up the tree
        * If the BF is nonzero and no rebalance was needed, keep going up the tree
- Balancing takes constant time and we do it at most once per insert
- Since we have to first insert the node and then possibly update the BF of all its parents, the complexity of insert is $O(\log n)$
- For delete, the operations are similar but now since the tree gets shorter, we do at most $O(\log n)$ rotations

Lecture 7, Oct 2, 2023

Augmenting Data Structures

- Sometimes we need to modify an existing data structure to perform additional operations; this is called *augmenting* the data structure
    1. Determine which additional info to store for your operations
    2. Check that this additional information can be cheaply maintained during each operation – don't accidentally change the runtime complexity of the structure's operations!
    3. Use the additional information to efficiently implement the new operations you need
- Example: Dynamic Order Statistics: maintain a dynamic set S of elements with distinct keys, supporting search, insert, delete, in addition to:
    - SELECT(k): find the element with rank k, i.e. the kth smallest element in S
    - RANK(x): determine the rank of the element x, i.e. its order in the set
    - We can augment an AVL tree!
        1. At each node x, we will store the size of the subtree rooted at x
            * For every other node, the size is the sum of the sizes of its two children plus one
            * For leaf nodes the size is 1, for nil nodes the size is 0
        2. This information is cheap to maintain, because to update the size of a node, we simply set it as the sum of the sizes of its children plus one
            * On insertion, increase the size of every parent node by 1, all the way up till the root, opposite on deletion
            * On a rotation, update the size of each node that underwent a rotation as the sum of the sizes of its children plus one
        3. We can use this information to efficiently implement SELECT(k) and RANK(k):
            * Observation: if a node has n nodes in its left subtree, then there are n nodes smaller than it in the left subtree, so it has *relative* rank $n + 1$ in its own subtree
            * For SELECT(k):
                - The rank of the current node is the size of the left subtree plus one
                - If k is equal to the current rank, return the current node since it has the correct rank already
                - If k is less than the current rank, recurse on the left subtree
                - If k is greater than current rank, recurse on the right subtree, but instead of k, the rank we want is now k minus the current rank
                - This has complexity $O(h) = O(\log n)$ since in the worst case it goes down the entire tree
            * For RANK(x):
                - If the current node is x, return the current rank
                - If the key at x is less than the key at the current node, recurse on the left subtree
                - If the key at x is greater than the key at the current node, recurse on the right subtree and return the result plus the current rank
                - This has complexity $O(h) = O(\log n)$ since in the worst case it goes down the entire tree
    - Focus on the modifications that you made to the original data structure, because it is assumed that you already know how the original operations work!

Lecture 8, Oct 4, 2023

Hash Tables

- A hash table is an implementation of a dictionary that uses *hashing*
- Idea: If the set of possible keys U is small, we can use a direct access table $T[0..u-1]$, so that item with key $k \in U$ is stored at $T[k]$
  - This gives us $\Theta(1)$ search, insert, and delete, at the cost of using a lot of memory – each possible key must have a slot associated with it, regardless of whether there's something stored there
  - If U is large, then it will be impractical or impossible to do this
  - Can we do better?
- Let m be the size of the hash table $T[0..m-1]$ and let n be the number of keys in S (we typically choose m so that it is $\Theta(n)$)

Definition

Given a set U of possible keys and a hash table of size m, a *hash function* h is a function that takes a key in U to an index in the table, i.e.

$$k \in U \to h(k) \in \{\, 0, 1, \ldots, m-1 \,\}$$

We say that the key k *hashes* into the slot $h(k)$ of T.

- The basic idea is that given a key k, we will store it at the slot $h(k)$
- Since m is smaller than the size of U, inevitably we will eventually have two distinct k hashing to the same $h(k)$; this is called a *hash collision*
  - One way to get around this is hashing with chaining, where each slot in T stores a linked list of all the keys with the same hash; on a collision, simply add the item to the start of the list
- With hash chaining, we have $\Theta(1)$ insertion, $\Theta(1)$ deletion (assuming we already have a pointer to the element), but $\Theta(n)$ search since in the worst case, all the keys can end up being hashed to the same slot
- Intuitively we want a hash function that spreads out the keys; we state this as the Simple Uniform Hashing Assumption (SUHA): any key $k \in U$ is equally likely to hash into any of the m slots of T, independent of all other keys
  - The probability of k hashing into i is $\dfrac{1}{m}$ regardless of i
- Starting from an empty T and inserting n keys, by SUHA we expect $\dfrac{n}{m}$ keys in each slot on average; $\dfrac{n}{m} = \alpha$ is called the *load factor* of the table
  - If we perform a search for k, either the key is not in the table or it is; if the key is in the table, we expect $\dfrac{\alpha}{2}$ comparisons on average; if it is not, we expect α comparisons
  - Therefore the search on average takes $\Theta(\alpha)$
- As long as we keep n within a constant factor of m, we have constant $\Theta(1)$ time on all operations!
  - n grows with each insertion, but if it gets too large we can resize the table
- One way to implement such a hashing function is to simply take $h(k) = k \mod m$ where m is a prime number; then if we assume k is uniformly distributed, this hash function will satisfy SUHA

Lecture 9, Oct 11, 2023

Bloom Filters

- Bloom filters are like space-efficient "probabilistic dictionaries"
- Instead of storing the entire key, we will store only the hashes of a key in a set S
- The following operations are supported:
  - $\textsc{Insert}(S, x)$: insert x into S

  - SEARCH(S, x): search for x in S; this can have two results: "false" (in which case $x \notin S$ for sure) or "likely true" (in which case it is likely $x \in S$, but it could be a false positive)
- A bloom filter consists of an array $BF[0..m-1]$ of m bits, initially all set to 0, and t independent hash functions $h_1, \ldots, h_t$ that all map to the range $[0, m-1]$
  - We will assume that these hash functions are all SUHA, i.e. any key is equally likely to be hashed into any slot
- On INSERT(S, x), we hash x using all t functions, resulting in t indices; the bits at all these indices are set to 1
- On SEARCH(S, x), we hash x using all t functions, and check that all the bits at those indices are 1; if this is true, then x is likely in S, otherwise it is definitely not in S
- Suppose we insert n keys into an empty Bloom filter with m bits and t independent hash functions all satisfying SUHA; what is the probability that searching for a key not in the filter will return a positive result?
  - Consider an arbitrary index i in the filter; the probability that a key hashes to i for each hash function is $\frac{1}{m}$, or $1 - \frac{1}{m}$ to miss i
  - Therefore with t has functions, the probability of i remaining zero is $\left(1 - \frac{1}{m}\right)^t$, since all hash functions are independent
  - After n keys are inserted, the probability of i remaining zero is now $\left(1 - \frac{1}{m}\right)^{nt}$
    * Assuming $\frac{1}{m}$ is small, then we can approximate this as $\left(e^{-\frac{1}{m}}\right)^{nt} = e^{-\frac{nt}{m}}$
  - For a false positive we require that all t indices that x hashes to are 1; however the probability that each individual index is 1 is technically not independent
  - In practice, we can assume that these events are independent to get a (pretty good) approximation that the probability of a false positive is $\left(1 - e^{-\frac{nt}{m}}\right)^t$
- How do we find the optimal size of t?
  - Fix the ratio $\frac{m}{n}$, and minimize $\left(1 - e^{-\frac{nt}{m}}\right)^t$ with respect to t
  - The optimal t turns out to be $\ln(2)\frac{m}{n} \approx 0.69\frac{m}{n}$
  - Substituting this back gives us $0.62^{\frac{m}{n}}$ as the chance of a false positive
  - e.g. allocating 8 bits per element gives us an optimal t of 5.52 (which we round to 6 has functions), giving us about 2% chance of false positives

Lecture 10, Oct 13, 2023

Randomized Quicksort

- We've seen algorithms that rely on the input being random to achieve a good average runtime; what if we make random choices inside the algorithm ourselves, so that even if the input is not random, we still get good performance?
- Suppose we want to sort a set of S distinct keys in increasing order; we can use recursive quick sort (RQS):
  1. If S is empty or has only one element, then return it
  2. Select a "pivot" key p uniformly at random among S (i.e. each key is equally likely to be selected as pivot)
  3. Compare the pivot with every element of S; split into two subsets $S_<$ which are keys less than p, and $S_>$ which are keys greater than p
  4. Recursively sort $S_<$ and $S_>$; output $S_<, p, S_>$ in order
- Note:
  - Two keys are compared only if one of them is selected as a pivot
  - Two keys are compared at most once, since a pivot cannot compare with keys after partitioning

  - If two keys are split by a pivot, they will never be compared
- Consider fixing some input S with n keys; let C be the number of pairwise key comparisons done by RQS
  - In the worst case, we choose the biggest or smallest element of the set as the pivot each time, so each recursive call reduces n by 1
    * $C = (n-1) + (n-2) + \cdots + 2 + 1 = \Theta(n^2)$
  - What is the expected value of C? i.e. in the average case, over all the possible pivot selections, how many comparisons do we get?
  - Let $z_1 < z_2 < \cdots < z_i < \cdots < z_j < \cdots < z_n$ be the keys of S in ascending order
    * Let $c_{ij} = 1$ if RQS compares z_i, z_j, or 0 otherwise
    * We call this an *indicator random variable*
    * We have $C = \sum_{1 \le i < j \le n} c_{ij}$
  - So what is $\mathbb{E}[c_{ij}]$?
    * $\mathbb{E}[c_{ij}] = 1 \cdot P(c_{ij} = 1) + 0 \cdot P(c_{ij} = 0) = 0$
    * We will later prove $c_{ij} = \dfrac{2}{j - i + 1}$
    * Substituting this back in and expanding the sum, we get that $H_n = 1 + \dfrac{1}{2} + \dfrac{1}{3} + \cdots \in O(\log n)$
- Consider the special case $i = 1, j = n$, then for $i = 1, j = n$, we have $\dfrac{2}{n}$ of them being compared
  - These two keys will only be compared if the first or last element s chosen as pivot
  - For any i and $j = i + 1$, the probability of comparison is guaranteed, since they are right next to each other
    * The only way these elements are separated is if one of them becomes a pivot
  - Consider the set of keys $Z_{ij} = \{ z_i, \ldots, z_j \}$
    * For this set to stay on the same side, the pivot must be its range
    * This set has size $j - i + 1$
    * Initially Z_{ij} is entirely contained in S
    * RQS keeps splitting S until it gets to subsets of size one or zero; as long as the pivot is not in the set, then the set will stay together and be untouched
    * Consider the first time RQS selects a pivot in Z_{ij}
      - If z_i or z_j is selected as the pivot, they will be compared once and never again
      - If $z_i < p < z_j$ then z_i or z_j will never be compared
    * Therefore the probability of z_i, z_j being compared is the probability of selecting z_i or z_j as the pivot, given $p \in Z_{ij}$
      - The size of Z_{ij} is $\dfrac{1}{j - i + 1}$ and we have two possibilities
      - So given any two keys, the probability of comparison is $\dfrac{2}{j - i + 1}$
- Hence, C is $O(n \log n)$

Lecture 11, Oct 16, 2023

Disjoint Sets (Union/Find)

- Consider a situation where we have n distinct elements named $1, \cdots, n$; initially each element is in its own set, $S_1 = \{ 1 \}, \ldots, S_n = \{ n \}$
  - Each set is represented by some element x
- We want to support the operations:
  - $\textsc{Union}(S_x, S_y)$: create a new set $S = S_x \cup S_y$ and return the representative element of S
  - $\textsc{Find}(x)$: find the set containing the element x and return its representative element
- Using such a data structure we can test if two elements are in the same set by checking if $\textsc{Find}(x) = \textsc{Find}(y)$
- Consider a sequence σ of m $\textsc{Find}$ operations and n $\textsc{Union}$ operations; we would like to analyze the

complexity

- We can implement this using a disjoint-set forest:
  - Each set is represented by a tree, where the root node contains the set representative
  - Each node contains one element
  - Each non-root node points to its parent
  - Since we only need a parent pointer, this can be efficiently implemented using an array of n elements, with index i containing the parent of i
- The operations can be implemented as follows:
  - To find, we simply traverse up the tree until we reach the root
  - To merge, we find the root of both sets, and make one of them a child of the other
  - With this, in the worst case we can get a complexity of $O(mn)$ since merging sets can create a chain of m nodes
- To improve this, we can perform weighted union (WU) by size, i.e. every time we merge, we make the larger tree the parent
  - With this, any tree T created during the execution of σ has height at most $\log_2(n)$
    - Lemma: any tree T of height h created during the execution of σ has at least 2^h nodes
      - Base case: for $h = 0$, any tree of height 0 contains at least $2^0 = 1$ node
      - Inductive step: suppose the lemma holds for some h; we will show that it holds for $h + 1$
        - The tree must have been created by merging two trees, one of height h and one of height $h + 1$
        - By the inductive hypothesis the height h tree has at least 2^h elements
        - Since the smaller tree is the child, the height h tree must be the child, so the height $h + 1$ tree must be bigger and has more than 2^h nodes
        - Therefore overall the tree has at least $2^h + 2^h = 2^{h+1}$ nodes
    - Since $2^h \leq |T| \leq n$, we have $h \leq \log_2 n$
  - Therefore the worst case cost is $O(m \log n)$
- Another more effective technique is path compression (PC): after a FIND(x) operation, before returning, the parent of x is set to be the representative element of the set containing x, so that future FIND operations take a shorter path
  - This increases the cost of FIND, but makes future operations cheaper
  - This is called *amortization*

Important

Differences between our data structure and the one described in CLRS:

- We assume that x and y in the UNION operation are the representatives of their respective sets (as opposed to CLRS which does not require this).
- In our analysis it is assumed that we have n elements and m FIND operations (as opposed to m total FIND and UNION operations in CLRS)
- In our disjoint set forest, we are using a weighted union heuristic, i.e. union-by-size (as opposed to union-by-rank in CLRS)

Lecture 12, Oct 18, 2023

Disjoint Sets (Continued)

- Recall that we can optimize a disjoint set forest using weighted union (WU) and path compression (PC) techniques; what is the complexity of a sequence σ of $n - 1$ UNION operations and $m > n$ FIND operations?
- Define the function 2^{*n}:
  - $2^{*0} = 1$
  - $2^{*n+1} = 2^{2^{*n}}$ for $n \geq 0$
  - This function grows as: $2^{*0} = 1, 2^{*1} = 2, 2^{*2} = 4, 2^{*3} = 16, 2^{*4} = 65536, 2^{*5} \approx 1 \times 10^{19\,729}, \ldots$

- Define the inverse 2^{*n} as $\log^* n = \min\left\{ k : 2^{*k} \geq n \right\}$, i.e. the first value of k such that 2^{*k} is greater than n
  - For all intents and purposes this is basically constant
- We claim that with WU and PC, executing σ takes $O(m \log^* n)$ time (proof is left as an exercise to the reader)
- Is there some sequence of σ that takes at least $m \log^* n$ time? Can we execute σ in $O(m)$ time?
  - The real complexity is actually between the two – not quite linear, but growing slower than $m \log^* n$
  - The actual complexity is $\Theta(m\alpha(m, n))$ where $\alpha(m, n)$ is the inverse Ackermann function
  - The lower bound of $\Omega(m\alpha(m, n))$ applies for any disjoint set data structure, not just the forest implementation
  - This took 25 years to derive

Lecture 13, Oct 23, 2023

Amortized Complexity

- Given a data structure with some operation, $T(n)$ is the worst case of doing *any* sequence of n operations, over all possible such sequences
- $\dfrac{T(n)}{n}$ is the "average" cost of an operation in the worst case
  - This is referred to as the *amortized cost* of an operation
- There are two ways to compute the amortized cost:
  - Aggregate analysis: computing $T(n)$ directly by summing the cost of n operations
    - Try to do a sequence of operations and see if we can find a periodic pattern
  - Accounting method: "charging" some amount per operation (which can be more or less than the actual cost of that operation), so that the overcharge is used as "credit" towards future more expensive operations that are undercharged
    - We require that the total charges on any sequence of n operations is greater than or equal to the actual cost of doing these operations; this is the *credit invariant*
- Example: incrementing a k-bit binary counter
  - The counter starts at $A = 0$
  - Cost is the number of bits that get flipped with each increment
  - $T(n)$ is then the total number of bits flipped by a sequence of n increments
  - Solve by aggregate analysis:
    - We notice a pattern: bit 0 is flipped on every increment, bit 1 is flipped every other increment, bit 2 is flipped every 4 increments and so on
    - Given n increments, bit i is flipped $\left\lfloor \dfrac{n}{2^i} \right\rfloor$ times
    - The total cost is $\displaystyle\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{k-1} \frac{n}{2^i} = 2n$
    - Therefore the amortized cost is $\dfrac{T(n)}{n} \leq 2$, or $\Theta(1)$
  - Solve by the accounting method:
    - Notice that every increment will do some sequence of flipping 1s to 0s, and one single flip from 0 to 1
      - After the flip from 0 to 1 we no longer have a carry so the flipping stops
    - Whenever we flip from 0 to 1, we can charge for 2 flips, one for the actual operation and one more for the eventual flip from 1 back to 0
      - We attach the overcharge (credit) to the 1 that we just created
      - Next time when we need to flip the 1 back to a 0, we can use the credit that we attached to the 1 instead
      - This maintains the credit invariant, since we start the counter at 0
    - At any point the total credit we have is equal to the number of 1s in the counter; therefore

whenever we do an increment, we can use the existing credit to flip all the bits from 1 to 0; this maintains the credit invariant
    * Since we charge 2 for each operation, the amortized cost is 2, or $\Theta(1)$

Lecture 14, Oct 25, 2023

Amortized Analysis Example: Dynamic Tables

- Consider a table T occupying a contiguous region in memory, which supports the insert and delete operations; let $\alpha(T)$ be the load factor (ratio of items stored to size)
- When we insert to reach $\alpha(T) = 1$, we have to expand the table by allocating a new table larger than T and then copy over all the elements
  - Typically, we double the size of the table every time it is full
  - With this, $\alpha(T) \geq \frac{1}{2}$ (we don't waste more than half of the table)
  - Each insertion where the table is full costs 1 for insertion and the cost to copy the entire table
- What is the amortized cost per insertion?
  - Starting with aggregate analysis:
    * Starting from an empty table of size 1, what is the cost of n insertions?
    * Notice that after n insertions, we will have expanded the table $\lfloor \log_2 n \rfloor$ times; each expansion costs the same as the size of the table at that point
    * Therefore the total cost of n inserts is n (insertions) plus the sum of all powers of 2 smaller than n (expansions)
    * The cost is $n + \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = n + (2^{\lfloor \log_2 n \rfloor} - 1) \leq n + 2n = 3n$
    * Therefore the amortized cost is $O(1)$
  - Now with the accounting method:
    * Let c_i be the actual cost of the i-th operation and $\hat{c}_i$ be the cost charged for that operation
    * We need $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all n, or equivalently the credit $\sum_i \hat{c}_i - \sum_i c_i \geq 0$ at all times
    * For a heuristic, consider just having finished an expensive operation so we have no credit, and think about when the next expensive operation will come
      - The expensive operation would be a table expansion; right after an expansion to a table of size n we have half of the entries, $\frac{n}{2}$, being filled
      - The next expensive operation will come in $\frac{n}{2}$ operations when the table is filled completely, which will cost n – giving us an extra charge of $\frac{n}{\frac{n}{2}} = 2$
      - Since inserting the item itself has a cost of 1, we charge 3 per insertion
    * Consider charging 3 per insertion; 1 will go towards the insertion of the element, and the rest 2 are stored as credit
      - 1 credit is attached to the element itself and the last one is attached to another element in the table
      - By doing this, whenever the table is filled completely we will have a credit on every single element
      - Now we can do the copying with the credits attached to the elements, so the credit invariant is maintained
    * For a sequence σ of n inserts, we charge 3 per insert to maintain the credit invariant, so the total cost is $3n$, giving us an amortized cost of $O(1)$
- When we delete elements such that $\alpha(T)$ is too low, we reallocate the table to reduce the amount of memory wasted, so that $\alpha(T) \geq c$, a constant, and to keep the amortized cost per operation constant
  - A naive approach would be to half the size of the table when $\alpha(T) < \frac{1}{2}$

    * This does ensure the load factor is at least 1/2, but if we have alternating insert and delete, we will be doubling and halving the size constantly
    * Consider a sequence σ of $\frac{n}{2}$ inserts, followed by an alternating sequence of 2 inserts and 2 deletes; now every 2 operations will give a cost of $\frac{n}{2}$, so we get $\Omega(n^2)$ complexity or $\Omega(n)$ per operation
  - A better approach would be to half the table when $\alpha(T) \leq \frac{1}{4}$
    * When we move to a smaller table, we will have half the space being filled
    * Regardless of deletion or insertion, the load factor will be 1/2 after moving; this will simplify analysis
    * Consider a sequence σ of inserts and deletes and use the accounting method
      - After an expensive operation, the table will be half full
      - The next expensive operation will be either an expansion or contraction
        - For an expansion, we need another $\frac{n}{2}$ insertions; the expansion costs n, so the averaged out cost is 2 extra per insertion
        - For a contraction, we need another $\frac{n}{4}$ deletions; the contraction costs $\frac{n}{4}$, so the averaged cost is 1 extra per deletion
      - Therefore we charge 3 per insertion and 2 per deletion
    * The amortized cost is then $O(1)$ per operation

Lecture 15, Oct 30, 2023

Graphs

- A *graph* is a collection of n nodes or vertices V and m edges E connecting two nodes
  - In an *undirected* graphs the edges are unordered pairs $(u, v) \in V$; in a *directed graph* they are ordered and directionality matters
  - Note self connections are possible
  - Note that m is roughly bounded by n^2
- Graphs can be stored in an adjacency list, an array in which each element represents a node, containing a subarray that contains all the nodes connected to that node
  - This representation takes $O(n + m)$ space; it is linear in both nodes and edges
  - For a sparse graph, this is very efficient
  - To check whether there is an edge between i and j takes $O(n)$ in the worst case (since every node can be connected to every other node)
- Another way is to use an adjacency matrix, an $n \times n$ matrix such that A_{ij} is 1 if there is an edge from i to j, or a 0 if there is no edge
  - Undirected graphs have symmetric adjacency matrices
  - This representation takes $O(n^2)$ space, so it's very inefficient for sparse graphs
  - The advantage is that we can query whether there is an edge between i and j in $O(1)$ time
- A *graph search* is a systematic exploration of a graph
  - Different graph search algorithms explore the nodes in different order
  - Graph searches can reveal structural properties of the graph, e.g. connectedness, presence of cycles, shortest paths
- Breath-first search (BFS) searches nodes in the order of their discovery
  - Newly discovered nodes are inserted into a queue to maintain FIFO order
  - The next node to explore is taken from the head of the queue
  - While the algorithm is exploring the graph it maintains 3 properties about each node:
    * $color[v]$, which is white if it's undiscovered, grey if discovered but unexplored (i.e. in the queue), and black if explored
    * $p[v] = u$, the parent of each node (i.e. the node v was discovered while exploring u)
    * $d[v]$, the length of the discovery path from the starting node (i.e. the number of edges in the

path from the starting node to v)
  - Note $d[v] = d[u] + 1$
  - The complexity is $O(n + m)$ since we go through the adjacency list
  - The result of BFS, $p[v]$, is called the *BFS tree*; this tree contains the shortest path from the starting node to every other node

Lecture 16, Nov 1, 2023

Correctness of BFS

- Denote $\delta(s, v)$ as the length of the shortest path from the starting node s to a node v
- Suppose we execute BFS on a graph G; we wish to prove that, for every node $v \in V, d[v] = \delta(s, v)$, i.e. the length of the discovery path by BFS is the length of the shortest path, or that the discovery path is the shortest path
- Lemma 0: $\delta(s, v) \leq d[v]$ (trivially true)
- Lemma 1: if a node u is inserted into Q before v, then $d[u] \leq d[v]$
  - Proof by contradiction: suppose that this lemma is false, and let u, v be the first of such a pair so that $d[u] > d[v]$
  - $v \neq s$, because no vertices enter before s
  - $u \neq s$, because $d[s] = 0$, so if $u = s$, then that would imply $d[v] < 0$
  - Therefore both u and v must have been discovered by some other nodes, u', v'
  - $d[u'] = d[u] - 1, d[v'] = d[v] - 1 \implies d[u'] > d[v']$, so u', v' must be different
  - Since u was inserted into Q first, u' must have been in the queue before v'
  - Hence we have u' in the queue before v' and $d[u'] > d[v']$, and both were inserted before u, v
  - But we assumed that u, v were the first pair that violated the lemma, so since u', v' came before, $d[u'] \leq d[v']$, leading to a contradiction
- Theorem: after BFS, for every $v \in V, d[v] = \delta(s, v)$
  - Proof by contradiction: suppose there exists $x \in V$ such that $d[x] \neq \delta(s, x)$
  - Pick v to be the close node from s that that $d[v] \neq \delta(s, v)$
  - By Lemma 0, $d[v] > \delta(s, v)$
  - Consider an actual shortest path from s to v; let (u, v) be the last edge on that path; then $\delta(s, u) = \delta(s, v) - 1$
  - Since $\delta(s, u) < \delta(s, v)$, u cannot violate the lemma and therefore $d[u] = \delta(s, u)$
  - $d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$
  - Now consider the color of v just before u is explored
    * Case 1: v is white
      - When we explore u, we will set $d[v] = d[u] + 1$ since v is still white and connected to u
      - But $d[v] > d[u] + 1$ so this is a contradiction
    * Case 2: v is black:
      - Since v has already been explored before u, it has entered the queue before u
      - By Lemma 1, this means $d[v] \leq d[u]$, leading to a contradiction
    * Case 3: v is grey:
      - Since v is grey before we've explored u, it must have been discovered by a node $w \neq u$
      - Therefore w must have been in the queue before u, so by Lemma 1 $d[w] \leq d[u]$
      - Since w discovered v, $d[v] = d[w] + 1$
      - $d[w] \leq d[u] \implies d[w] + 1 \leq d[u] + 1 \implies d[v] \leq d[u] + 1$ which is a contradiction
- Therefore the BFS discovery path from s to v is a shortest path
- Note we could have proven this by induction

Lecture 17, Nov 13, 2023

Depth First Search

- In DFS, the last discovered node is the first explored
- Like BFS, we will track the color: white/grey/black, and parent $p[v]$
- DFS also tracks $d[v]$, the "discovery time" of v, and $f[v]$, the "exploration complete time" of v
  - To keep track of time we use a counter which is incremented by 1 every time we discover a node and finish exploring a node
- DFS Algorithm:
  - Initialization: set every color to white, $d[v], f[v]$ to infinite, $p[v]$ to nil, time to 0
  - For every node v, if v is white, call the recursive DFS-Explore(G, v) on the node
- The DFS-Explore(G, u) procedure explores all nodes reachable from u:
  - Color u grey, increment $time = time + 1$, set $d[v] = time$
  - For each node v connected to u, if v is white, set $p[v] = u$ and call DFS-Explore(G, v)
  - Once all connected nodes are explored, color u black, increment $time = time + 1$, set $f[v] = time$
- Note that $f[v]$ for the final node is exactly $2V$, since every node contributes 2 to the time
- The time complexity is also $O(V + E)$ like BFS
- The tree formed by all edges in $p[v]$ (i.e. all discovery edges) is the DFS forest, similar to the BFS forest
  - Edges in the original graph that are included in the DFS forest are tree edges
- A non-tree edge can be one of 3 types:
  - Forward edge: going from an ancestor to a descendant in the DFS forest
  - Back edge: going from a descendant to an ancestor in the DFS forest
  - Cross edge: neither forward nor backward edges (i.e. between two different subtrees)
- Note some properties of $d[v]$ and $f[v]$:
  - If u is an ancestor of v, $d[u] < d[v] < f[v] < f[u]$
  - For any 2 nodes u and v, we can never have $d[u] < d[v] < f[u] < f[v]$
  - If there is an edge from u to v, we will always have $d[v] < f[u]$
  - For any tree or forward edge (u, v), $d[u] < d[v] < f[v] < f[u]$ since u is an ancestor of v
  - For any back edge (u, v), $d[v] < d[u] < f[u] < f[v]$ since v is an ancestor of u
  - For any cross edge (u, v), $d[v] < f[v] < d[u] < f[u]$
    - Since u and v have no relation, and we cannot have $d[u] < d[v] < f[u] < f[v]$, we must have either $d[v] < f[v] < d[u] < f[u]$ or $d[u] < f[u] < d[v] < f[v]$, but the latter is impossible since there is an edge (u, v)
- Note DFS is not unique and it depends on the node we starts from

Lecture 18, Nov 15, 2023

Applications of DFS – Cycle Detection

Theorem

White-Path-Theorem (WPT): For all graphs G and all DFS of G, v becomes a descendant of u if and only if at the time $d[u]$ the DFS discovers u, there exists a path from u to v that consists of entirely white nodes.

- Proof of WPT: consider any G and any DFS of G
  - Suppose v is a descendant of u:
    - Let the discovery path from u to v be $u \to u_1 \to \cdots \to u_k \to v$
    - At the time $d[u]$ when u is discovered, all the other nodes on the path have not yet been discovered
    - Therefore all the nodes in the discovery path are white, so there exists a white path from u to v
  - Suppose at time $d[u]$ when u is discovered, there exists a white path from u to v

    * Claim: all nodes in this path, including v, will become descendants of u
    * Suppose there is at least one node along this path that does not become a descendant of u; let z be the closest node to u in the path that does not become a descendant of u
    * Let w be the node before z in this graph; since w is closer to u than z, it must be either u or a descendant of u
    * We know $d[u] < d[z]$ because z is white when u was discovered
    * We know $d[z] < f[w]$ because there is an edge from w to z
    * We know $f[w] \leq f[u]$ because w is either a descendant of u, or u itself
    * Therefore $d[u] < d[z] < f[w] \leq f[u]$; because z was discovered between the discovery and exploration of u, so $d[u] < d[z] < f[z] < f[u]$, but this means z is a descendant of u
    * This leads to a contradiction, so all nodes in the path will become descendants of u
- Now that we have proven WPT we can use it for cycle detection

Theorem

A directed graph G has a cycle if and only if any/every DFS of G has a back edge.

- Proof: consider any directed G and any DFS of G
  - Suppose the DFS of G has a back edge (v, u)
    * v is a descendant of u in the DFS, so there is a discovery path u to v
    * Hence we have a cycle by going from u to v via the discovery path, and then back to u via the back edge
  - Suppose G has a cycle C
    * Let u be the first node in C that the DFS discovers
    * Let v be the node right before u in C
    * At time $d[u]$ when the DFS discovers u, all nodes in the path from u to v in C are still white, including v (since u is the first node in the entire cycle that is discovered)
    * By WPT, v eventually becomes a descendant of u in the DFS
    * When v becomes a descendant of u, we explore it, it has an edge to u; this is a back edge, since v is a descendant of u
- After we do a DFS, how do we figure out whether an edge (u, v) is a back edge?
  - When we go down the edge, we can check that both u and v are grey
  - If we're already done the DFS, then we can check whether $d[v] < d[u] < f[u] < f[v]$

Lecture 19, Nov 20, 2023

Minimum Spanning Trees (MSTs)

- Recall that a *tree* is any undirected graph that contains no cycles
- Note some tree facts:
  - A tree with n nodes has $n - 1$ edges
  - Adding any edge to a tree creates a cycle containing that edge
  - Removing any edge from that cycle gives you a tree

Definition

Given a connected graph G, a *spanning tree* of the graph is any tree T that connects (i.e. spans) all the nodes of G.

If G is a weighted graph, then a *minimum spanning tree* (MST) of G is any tree T that has the minimum possible weight, defined as the total weight of all its edges.

- The spanning tree will be a subgraph of G, and is not unique; the MST is also non-unique if there are repeated edge weights

- The MST construction problem is the problem of constructing an MST T of G, given a weighted undirected connected graph $G = (V, E)$
- A *spanning forest* of G is a "fragment" of a spanning tree of G (i.e. the spanning tree with some edges removed)
  - It is a set of trees $T_1 = (V_1, E_1), \ldots, T_k = (V_k, E_k)$, such that $\{ V_1, V_2, \ldots, V_k \}$ is a partition of V and $E_i \subseteq E$

Theorem

MST Construction Theorem: Suppose some MST T of G contains the spanning forest $T_1, \ldots, T_k$ of G. Let (u, v) be an edge G of minimum weight between T_i and the other trees of the spanning forest (i.e. $u \in V_i, v \in V \backslash V_i$), then there must be an MST T^* that contains $T_1, \ldots, T_k$ and the edge (u, v).

- This means starting from the trivial forest (where there are no edges), we can repeatedly apply this algorithm to build up the MST edge by edge, and stop when the number of edges is exactly $n - 1$
- Proof:
  - Suppose some MST T of G contains the spanning forest $T_1, \ldots, T_k$ of G
  - Let (u, v) be an edge of G of minimum weight that connects T_i and any other tree in the forest
  - If the edge is a part of T, then $T^* = T$ and we're done
  - Otherwise, let $T' = T + (u, v)$ (i.e. add the new edge to T); this will create a cycle containing (u, v), so T' is no longer a tree
  - The existence of this cycle means there is another edge (u', v') such that $u' \in V_i$ and $v' \in V \backslash V_i$ (i.e. (u', v') goes back from the rest of the world to V_i)
  - Since (u, v) has minimum weight, the weight of (u, v) must be less than or equal to the weight of (u', v')
  - Let $T^* = T' - (u', v') = T + (u, v) - (u', v')$
    * Note that T^* must be a spanning tree, since removing an edge from the cycle gives back a tree
    * T^* contains $T_1, \ldots, T_k$ and (u, v)
    * Also, the weight of T^* is less than or equal to T; but T is an MST, so T^* must have the same weight and therefore it is an MST

Lecture 20, Nov 22, 2023

Kruskal's MST Algorithm

- Kruskal's algorithm builds up the MST edge-by-edge
- Sort all edges in the forest in ascending order, and keep a spanning forest (starting from the trivial forest); for every edge in order of cost, if it connects two disjoint forests, include it in the MST; stop when we have exactly $n - 1$ edges
  - The algorithm can be proven correct by induction, making use of the MST construction theorem
- Use a union-find structure to keep track of the partition of nodes into minimum spanning forests
- Often times we don't need to go through all edges; to optimize the algorithm we can use a heap and extract only the edges as necessary
  - The initial call to HEAPIFY() takes only $O(m)$, and each subsequent extraction takes around $O(\log m)$
  - In the worst case where we need to look at every edge, this still takes $O(m \log m)$, but if we only need a fraction of edges this can be substantially faster
- Formally: given a connected, undirected, weighted graph $G = (V, E)$ where $V = \{ 1, 2, \ldots, n \}$ and E is an array of m weighted edges:
  1. Turn E into a min-heap
  2. For $i = 1$ to n, MAKE-SET(i) (initialize disjoint sets)
  3. Initialize the set of MST edges with an empty set
  4. While we have less than $n - 1$ edges in the MST, do:
     1. Extract the edge (u, v) with minimum weight from E

    2. Find the representatives of the sets containing u, v; if they are different:
        1. Merge the two sets
        2. Add the edge (u, v) to the MST
- Complexity analysis:
  - Building the min-heap takes $O(m)$
  - Making the n sets takes $O(n)$
  - In the worst case, the loop runs m times since we have to go through all edges
  - All the heap extractions then take $O(m \log n)$ (note $O(m \log n) = O(m \log m)$, since $m \leq n^2$)
  - We also have $n - 1$ unions and at most $2m$ finds, giving a complexity of $O(m \log^* n)$ (using a disjoint-set forest with path compression and union by size)
  - Therefore the total complexity is $O(m \log n)$

Lecture 21, Nov 27, 2023

The Travelling Salesman Problem and Approximate Algorithms

- Travelling Salesman Problem: Given an undirected complete graph $G = (V, E)$ (i.e. edge between every pair of nodes) with non-negative weights, find a *tour* of G of minimum total edge cost
  - A *tour* is a cycle that visits every node of G exactly once
  - Such a minimum cost tour is known as a *TSP tour*
- The brute force algorithm is to find all tours of G and select the minimum cost one, but this takes exponential time
- The TSP problem is NP-hard, i.e. it is at least as hard as all NP-complete problems, so it's very unlikely that we can find a polynomial time algorithm
- The Δ-TSP (also known as triangle-TSP or metric-TSP) is a special case of the TSP problem where the edge costs satisfy the triangle inequality, i.e. for all nodes u, v, w of G, we have $c(u, w) \leq c(u, v) + c(v, w)$
  - e.g. if nodes are locations in space and the edge costs are Euclidean distances, then we have a Δ-TSP problem
  - This is easier to solve than TSP in general, but it is still NP-hard
  - However, we can find a polynomial-time approximate algorithm, which finds a tour whose cost is within a certain constant factor of the optimal tour
- Lemma: For any complete undirected G, the cost of an MST of the graph is always less than or equal to a minimum cost tour
  - Consider any TSP tour of G, which will be a cycle; remove one edge, and now we have a spanning tree
  - Therefore there will always be spanning trees that cost less than the TSP tour of G, so the MST will also cost less than the TSP tour
- The approximate Δ-TSP algorithm is as follows:
  1. Find an MST of G (this takes $O(m \log n)$ time with e.g. Kruskal's algorithm)
  2. Do a full walk of the MST (i.e. like a DFS); this gives a sequence of nodes C
     - C is a cycle that uses each MST edge twice, so it has twice the cost of the MST
     - Note that C is a cycle but not a tour, since it visits some nodes multiple times
  3. Transform C into a tour C^* by using shortcuts to remove repeated nodes
     - Whenever we come back to a node that we have previously visited, we instead skip it and move to the next node directly
     - Using the triangle inequality assumption, we know that this does not increase the cost
     - Since we never increased the cost by using shortcuts, the cost C^* is less than the cost of C, which is twice the cost of the MST
     - By the lemma above, we know the MST cost $<$ TSP tour cost, so the cost of C^* is less than twice the TSP tour cost
- Better algorithms can improve this factor of 2 to 1.5

Lecture 22, Nov 29, 2023

Analyzing Problem Complexity – Decision Trees

- Given a general problem, what is the cost of solving the problem, by the best possible algorithm?
  - This is the problem complexity
- Example problem: sorting m distinct algorithms
  - We know we can do it in $O(n \log n)$ comparisons since we have algorithms that achieve this
  - Can we do better?
- Theorem: Any comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons in the worst case
  - A comparison-based sorting algorithm is an algorithm that is only allowed to compare two elements in the input, and make decisions based on the result
  - This prohibits e.g. bucket sort or counting sort, since these use the actual value of the element
  - Therefore, heapsort and mergesort are asymptotically optimal
- Any such sorting algorithm $\mathcal{A}$ executing on a finite input can be described by a *decision tree*, a binary tree where at each node we have a comparison, and each of the two possible outcomes of the comparison gives a subtree
  - Each internal node of the tree has a label $i : j$, which represents a comparison between elements a_i and a_j
    - The left subtree, $\leq$, denotes all possibilities where $a_i \leq a_j$; similarly the right subtree $>$ denotes $a_i > a_j$
  - Every leaf of the tree represents one possible solution of the problem, i.e. a permutation of the input list
- Using the decision tree we can prove the above theorem:
  - Let $\mathcal{A}$ be any comparison-based sorting algorithm to sort n distinct integers
  - Let $T_{\mathcal{A}}$ be its corresponding decision tree
  - For each input permutation π of integers $1, 2, \ldots, n$, $T_{\mathcal{A}}$ must have a distinct reachable leaf representing the sorting of π, therefore $T_{\mathcal{A}}$ must have at least $n!$ leaves
  - Let h be the height of $T_{\mathcal{A}}$; since $T_{\mathcal{A}}$ is a binary (or any n-ary tree) of height h, it has at most 2^h leaves
  - Therefore $h \geq \log(n!)$, which is $\Theta(n \log n)$, so h is $\Omega(n \log n)$
  - Since we need h comparisons to reach a leaf in the worst case, $\mathcal{A}$ also does $\Omega(n \log n)$ comparisons in the worst case

Lecture 23, Dec 4, 2023

Analyzing Problem Complexity – Adversary Approach

- Example problem: Find both the minimum and maximum of a set S of n distinct integers
- Naive algorithm: scan S twice, first to find the maximum, and then find the minimum
  - Finding the max takes $n - 1$ comparisons exactly
  - Finding the minimum then takes $n - 2$ (since we don't have to compare against the max we just found)
  - Therefore total is $2n - 3$ comparisons
- Improved algorithm: divide S into $\frac{n}{2}$ pairs and find the maximum and minimum of each pair; then scan all the maxes to find the max, and all the mins to find the min
  - Initially $\frac{n}{2}$ comparisons to find the max and min of each pair, then $\frac{n}{2} - 1$ comparisons each to find the max and min
  - Therefore the total is $\frac{3n}{2} - 2$ comparisons
- Another algorithm is a divide-and-conquer approach of first dividing the set into 2, finding the min and max of each set, and then compare those
  - This gives the same number of comparisons as the above algorithm, however

- Theorem: Any comparison-based algorithm to solve this problem makes at least $\frac{3n}{2} - 2$ comparisons in the worst case
- We prove this by using an *adversary argument*: given any algorithm, the adversary will come up with an input that forces the algorithm to do at least $\frac{3n}{2} - 2$ comparisons
- At any point, we can categorize every element in the input set into 4 subsets: N – never compared, W – won every comparison so far, L – lost every comparison so far, M – won some and lost some comparisons
  - Initially, the size of N is n, while every other set has size 0
  - When the algorithm finishes, the size of N will be 0, the size of W and L are both exactly 1, and the size of m is $n - 2$
- Intuitively, the maximum will win all comparisons, and the minimum will lose all comparisons; all other nodes have mixed comparison results
  - The adversary wants to delay the creation of mixed comparison results as much as possible
  - Note the adversary must not create cycles as to keep the input valid
  - The rough idea is we want elements in W to keep winning comparisons, and elements in L to keep losing, to delay populating M for as long as possible
- Adversary's strategy:
  - Compare N to N: assign arbitrarily, increasing W by 1 and L by 1
  - Compare N to W: N loses, increasing L by 1 and keeping W the same
  - Compare N to L: N wins, increasing W by 1 and keeping L the same
  - Compare N to M: N wins, increasing W by 1 and keeping M the same
  - Compare W to W: one wins, increasing M by 1 and decreasing W by 1
  - Compare W to L: W wins, keeping both the same
  - Compare W to M: W wins, keeping both the same
  - Compare L to L: one wins, increasing M by 1 and decreasing L by 1
  - Compare L to M: M wins, keeping both the same
  - Compare M to M: assign arbitrarily, keeping both the same
- Claim: by following this strategy, we can always produce inputs that are consistent (i.e. no cycles) and forces the algorithm to take at least $\frac{3n}{2} - 2$
  - Starting from n elements all in N, any algorithm must:
    1. Create $n - 2$ elements in M
       * This only happens when we compare W to W or L to L
       * We need exactly $n - 2$ comparisons of this type to create the elements we need in M
    2. Create n elements in W or L ($n - 2$ of which will be changed to M, with the last 2 remaining)
       * The best way to do this is by comparing N to N, which creates 1 of each W and L
       * Therefore we need $\frac{n}{2}$ comparisons to create the n that we need
  - Therefore the algorithm must perform at least $n - 2 + \frac{n}{2} = \frac{3n}{2} - 2$ comparisons to reach the result