CS2040S
Data Structures and Algorithms

AY2021/22 Semester 2

Notes by Jonathan Tay

Last updated on April 24, 2022

Contents

1 Sorting

1.1 Bubble Sort

Repeatedly swaps adjacent elements that are out of order until there are no swaps in an iteration, or after n iterations.

At the end of iteration j, the largest j items are correctly sorted in the final j positions of the array.

In the best case of a sorted array, this takes $O(n)$, and $O(n^2)$ otherwise. BubbleSort is stable.

1.2 Selection Sort

Maintains a sorted prefix, and repeatedly finds the smallest element in the unsorted remainder and swaps it with the first element in the remainder.

At the end of iteration i, the smallest i items are correctly sorted in the final i positions of the array. In all cases, this takes $O(n^2)$ comparisons, and is unstable.

1.3 Insertion Sort

Maintains a sorted prefix, and repeatedly inserts unsorted elements into the correct place in the sorted prefix.

At the end of iteration i, the first i items are in sorted order. In the best case of a sorted array, this takes $O(n)$, in the average case of a random permutation $\theta(n^2)$, and in the worst case of an inverse sorted array, $O(n^2)$.

InsertionSort is stable as long as we do not swap identical elements.

1.4 Merge Sort

Using divide-and-conquer, we split the array into halves, recursively sorting both halves and then merging the two.

`merge` takes n iterations to move all elements to a final array, and each iteration takes $O(1)$ time to compare and copy elements. This gives a running time of $O(n) = cn$.

We get the recurrence for `merge-sort`:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

Therefore, we get an overall time complexity of $O(n \log n)$. `merge` also requires allocating space for the final array, which gives a total space complexity of $O(n \log n)$.

Merge sort is not in-place, but is stable if `merge` is implemented properly.

1.5 Quick Sort

Using divide-and-conquer, we partition the array around a pivot, and then recursively sort the two subarrays.

If there are no duplicates, we partition the array A into a new array B by using two pointers `lo` and `hi` starting from both ends of the array.

By iterating over A, if the element at $A[i]$ is less than the pivot, we copy it to `B[lo++]`, or `B[hi--]` otherwise.

Two invariants will hold:
$\forall i <$ `lo`, `B[i]` $<$ `pivot` and $\forall j >$ `hi`, `B[j]` $>$ `pivot`.

This implementation of `partition` takes $O(n)$ time, but is not in-place.

1.5.1 In-place Partitioning

```
partition(A, n):
    swap(pivot, A[0])
    lo = 1
    hi = n − 1
    while lo < hi:
        while A[lo] < pivot and lo < hi: lo++
        while A[hi] > pivot and lo < hi: hi−−
        if (lo < hi) swap(A[lo], A[hi])
    swap(A[lo], A[0])
    return lo
```

Two invariants will hold at the end of every iteration: $\forall i \geq$ `hi`, `A[i]` $>$ `pivot` and $\forall\ 1 \leq j <$ `lo`, `A[j]` $<$ `pivot`.

1.5.2 Three-way Partitioning

Partition an array with duplicates in-place using four regions.

Three invariants will hold at the end of every iteration:
$\forall\ i <$ `lo`, `A[i]` $<$ `pivot`
$\forall$ `lo` $\leq j <$ `mid`, `A[j]` $=$ `pivot`
$\forall\ k >$ `hi`, `A[i]` $>$ `pivot`

Regardless of which `partition` algorithm is used, `quick-sort` is not stable.

It has a recurrence of `T(n) = 2T(n/2) + O(n)`. In the worst case (e.g. A[0] pivot), this is $O(n^2)$. However, the expected running time with high probability is $O(n \log n)$.

2 Searching

2.1 Binary Search

Find the index of an element in a sorted array if it exists.

```
int binary_search(A, key, n):
    lo = 0
    hi = n − 1
    while lo < hi:
        mid = lo + (hi − lo) / 2
        if key ≤ A[mid]: hi = mid
        else: lo = mid + 1
    return A[lo] == key ? lo : −1
```

`A[lo]` $\leq$ `key` $\leq$ `A[hi]` is a **loop invariant**, and on iteration k, `hi` $-$ `lo` $\leq \frac{n}{2^k}$.

This has a time complexity of $O(\log n)$.

2.2 Peak Finding

Find the index of a local maximum in an unsorted array.

A **peak** is a local maximum in `A` such that `A[i]` $\geq$ `A[i-1]` and `A[i]` $\geq$ `A[i+1]`. We also assume that `A[-1]` = `A[n]` = `-MAX-INT`.

```
int find_peak(A, key, n):
    if A[n/2 + 1] > A[n/2]:
        return find_peak(A[n/2 + 1 : n], n/2)
    else if A[n/2 − 1] > A[n/2]:
        return find_peak(A[1 : n/2 − 1], n/2)
    else A[n/2] is a peak:
        return n/2
```

There will always exist a peak in `A[lo:hi]`, and every peak in `A[lo:hi]` is a peak in `A[1:n]`.

The recurrence relation is $T(n) = T(\frac{n}{2}) + \theta(1)$ which gives a time complexity of $O(\log n)$.

2.3 Quick Select

Find the k-th smallest element in an unsorted array.

Instead of sorting the entire array, we only do the minimum amount of sorting needed by recursing once on the correct side of a pivot. Recursing on both sides is Quick Sort!

```
quick_select(A, l, r, k):
  loop
      if l = r: return A[left]
      pivot = partition(A, l, r, random(l, r))
      if k = pivot: return A[k]
      else if k < pivot: r = pivot − 1
      else: l = p + 1
```

This has a time complexity of $O(n)$.

3 Hashing

3.1 Symbol Table

An abstract data type for key-value pairs with O(1) insertion, search, and deletion. Successor and predeccessor operations are unsupported.

Implementing a symbol table with an array (direct access table) indexed by integers is impractical for non-integer keys, and uses too much space.

Implementing a symbol with an AVL tree is slow, with lookup operations taking $O(\log n)$.

Furthermore, a dictionary with an AVL tree is a better choice for sorting in $O(n \log n)$ rather than $O(n^2)$ with a symbol table.

3.1.1 Hash Functions

Given a universe U of possible keys, we need to map them to a smaller number n of actual keys in m buckets.

A hash function $h : U \mapsto \{1, 2, \ldots, m\}$ is used to store key k in bucket $h(k)$. The time complexity depends on h and bucket access, but is usually assumed to be $O(1)$.

Two distinct keys k_1 and k_2 **collide** if $h(k_1) = h(k_2)$. It is impossible to choose a hash function that does not collide, due to the **pigeonhole principle**.

3.1.2 Chaining

Colliding keys are placed in the same bucket via a linked list.

For a table of size m and a linked list size of n, the total space used is $O(m + n)$.

For a hash function h with cost $\text{cost}(h)$, `insert` takes $O(1 + \text{cost}(h))$, while `search` takes $O(n + \text{cost}(h))$.

By the **simple uniform hashing assumption** where keys are equally likely to map to every bucket and are mapped independently, we can avoid the case where all keys hash to the same bucket.

Linearity of expectation states that $E(A+B) = E(A)+E(B)$. So, the expected number of items per bucket is $\frac{n}{m}$.

Therefore `search` is $1 + \frac{n}{m} = O(1)$ in expectation, but $O(n)$ in the worst case, and `insert` is still $O(1)$.

If we insert n items, the expected maximum cost is $O(\log n)$ and actually $\theta(\frac{\log n}{\log \log n})$.

3.1.3 Open Addressing

On collision, probe a sequence of buckets until we find an empty one.

We re-define h as $h(key, i) : U \mapsto \{1, 2, \ldots, m\}$.

However, we cannot simply `null` items in `delete`, and instead use a **tombstone value** which allows `insert` to overwrite it.

A good $h(k, i)$ must enumerate all possible values buckets, such that for every bucket j there is some i such that $h(k, i) = j$, meaning it is some permutation of $\{1, 2, \ldots, m\}$. Otherwise, it would wrongly declare a table to be full even when there is still space.

This is true for **linear probing**. However, LP does not obey the UHA (not SUHA!) where every key is equally likely to map to every permutation, independent of other keys, due to **clustering**.

When the table is $\frac{1}{4}$ full, clusters of size $\theta(\log n)$ form which ruins $O(1)$ performance. For a load $\alpha = \frac{n}{m}$, assuming uniform hashing, each operation is expected to take $\leq \frac{1}{1-\alpha}$.

With **double hashing**, $h(k, i) = f(k) + i \cdot g(k) \bmod m$, if $g(k)$ has no common factors with m other than 1, then $h(k, i)$ hits all buckets.

Open addressing saves space, rarely allocates, and has better cache performance, but is more sensitive to hash function choice and load.

3.2 Table Resizing

If a hash table is too small, there are too many collisions, and if it is too large, we waste space.

Upon resizing from sizes m_1 to m_2, we need to choose a new hash function h re-compute all n hashes, and copy them over. This takes $O(m_1 + m_2 + n)$.

If we increment table size by 1, this takes $O(n)$, meaning `insert` takes $O(n)$.

If we double table sizes, each resize costs $O(n)$, but the average cost of insertion is now $O(1)$. Squaring works as well, but space usage is inefficient.

3.2.1 Amortization

An operation has amortized cost $T(n)$ if $\forall k \in \mathbb{Z}^+$, the cost of k operations $\leq k \cdot T(n)$.

Therefore, resize tables only if $n = m$, then $m = 2m$ and if $n < \frac{m}{4}$, then $m = \frac{m}{2}$.

In another example, incrementing a binary counter takes amortized $O(\log n)$ time.

4 Binary Heaps

A **complete binary tree** is a binary tree where there are 2^i nodes at depth i except the last level which is **left-justified**.

A binary heap is implemented with an *implicit data structure* without pointers, using a **heap array**.

To access children of node i, we use index arithmetic: the left child is at index $2i + 1$, and the right child is at index $2i + 2$. To access the parent: $\lfloor \frac{i-1}{2} \rfloor$.

The **Max-Heap property** states that the key of every vertex is greater than or equal to the values of its children.

`insert` appends a new node to the heap array, and calls `swim` to swap the node upward with its ancestors until it is in the correct position.

`delete-max` swaps the last node in the heap array with the root, and calls `sink` to swap the root downward with the larger of its descendants until it is in the correct position.

Both `insert` and `delete-max` are $O(\log n)$ time.

`build` can be done in $O(n)$ time if we start from the end of the array and only use `sink`.

4.1 Priority Queue

An abstract data structure is a queue with elements served in order of their assigned priorities.

A binary heap is an implementation for a priority queue which must be a complete binary tree and obeys the MaxHeap property.

5 Trees

Connected acyclic graphs where two vertices are connected by only one path, with $|V| - 1$ edges in total.

5.1 Binary Search Trees

A tree which is either empty or has `left` and `right` children binary trees.

The **binary search tree property** states that all keys in the `left` sub-tree are less than the parent key, and all keys in the `right` subtree are greater than the parent key.

The **height** of a BST is the number of edges on the longest path from the root to the leaf:

$$h(\texttt{v}) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ -1 & \text{if } v \text{ is null} \\ \max(h(\texttt{v.left}),\ h(\texttt{v.right})) + 1 & \text{otherwise} \end{cases}$$

The **minimum** and **maximum** keys are found by traversing the left and right subtrees respectively.

Searching for a specific key is done by comparing the key with the parent key, recursively searching `v.left` if less than, `v.right` if greater than, and stopping if equal.

Inserting is akin to searching, but instead adding a new node at a leaf.

Deleting is trivial if if the node to be deleted has one or no children. Otherwise, the node to be deleted is replaced by the minimum of the right subtree (its successor):

```
successor(v):
    if v has a right child:
        return the minimum key in v.right
    else:
        walk up the tree to a node w
        where w.parent.left = w, or w is the root
        return w
```

All these operations are $O(h)$. However, trees with the same keys need not have the same shape, and height depends on shape, which is determined by insertion order.

There are $n!$ insertion orders, and $\approx 4^n$ possible shapes for a binary tree.

5.2 AVL Trees

Augmented BST which stores the tree height in each node.

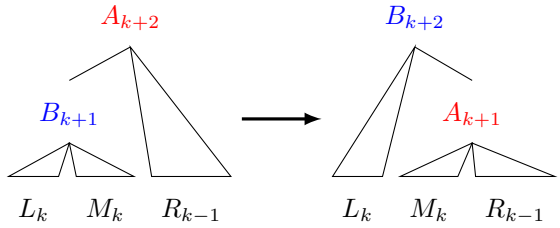
This augmented height field must be updated with every `insert` and `delete`.

A binary search tree is **balanced** iff $h = O(\log n)$, such that all operations take $O(\log n)$ time.

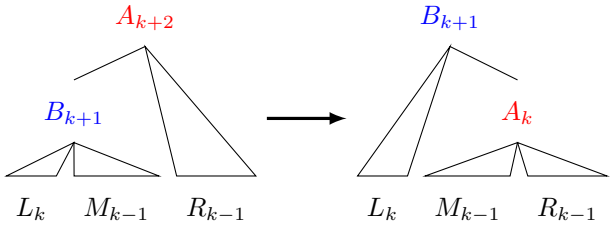
A node v is **height-balanced** iff $|\texttt{v.left.height} - \texttt{v.right.height}| \leq 1$. A height-balanced tree with n nodes has height $h < 2 \log n$, and one with height h has at least $n > 2^{\frac{h}{2}}$ nodes.

5.2.1 Rotations

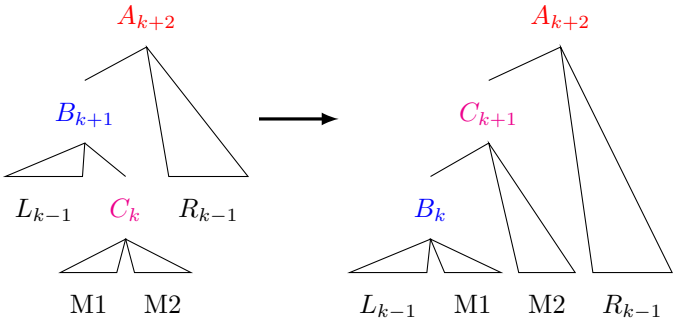
Used to rebalance trees after insertion and deletion. Left and right rotations are inverses of each other.



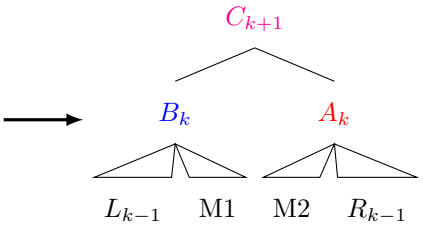
Case 1: A is left-heavy but B is balanced → right-rotate.



Case 2: A is left-heavy but B is left-heavy → right-rotate.



Case 3: A is left-heavy but B is right-heavy → left-rotate B, but A and C are still out of balance! M1 and M2 have a height of either $k-1$ or $k-2$.



Case 3 (continued): Fixing the imbalance with a right-rotate about A.

After an `insert`, at most 2 rotations are needed, but `delete` may require up to $O(\log n)$ rotations.

This is because `delete` reduces height and rotations reduce height, so it is not sufficient to only fix the lowest imbalanced node in the tree.

5.3 Dynamic Order Statistics

A tree data structure supporting not only insertion and deletion, but also selecting the k-th element.

Augment a balanced tree with **weight** information w on each node (not rank!):

$$w(\texttt{v}) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ w(\texttt{v.left}),\ w(\texttt{v.right}) + 1 & \text{otherwise} \end{cases}$$

```
select(k):
    rank = v.left.weight + 1
    if k = rank: return v;
    elif k < rank: return v.left.select(k)
    else k > rank: return v.right.select(k − rank)

rank(v):
    rank = v.left.weight + 1
    while v is not null:
        if v is left child: do nothing
        else: rank += v.parent.left.weight + 1
        v = v.parent
    return rank
```

Weights need to be updated with every rotation caused by `insert` and `delete`.

5.4 Dictionary

An abstract data type akin to a symbol table, but with support for successor and predecessor queries.

Implemented with balanced trees.

5.5 Interval Trees

A binary tree data structure where every node is an interval, and the tree is ordered by the lower bound of the interval.

We will need to augment each node with the **maximum upper bound** of the intervals in both subtrees, and also update this after every rotation.

```
interval_search(x):
    while c is not null and x is not in c.interval:
        if c.left is null: c = c.right
        elif x > c.left.max: c = c.right
        else c = c.left
    return c.interval

all_overlaps(x):
    repeat until no more intervals:
        interval = interval_search(x)
        add interval to list
        delete interval from tree
    add all intervals back to tree
```

`interval-search` takes $O(\log n)$ time, and `all-overlaps` takes $O(k \log n)$ time.

5.6 Range Trees

A binary tree data structure supporting orthogoanl range searching to determine the number of points within a range.

Unlike regular trees, range trees store points only in the leaves, and internal nodes store the largest value of its left subtree.

```
find_split(lo, hi):
    v = root
    loop:
        if hi ≤ v.key: v = v.left
        elif lo > v.key: v = v.right
        else: break
    return v
```

For a 1D range query, once the split node is found with `find-split`, we traverse both the left and right subtrees as long as they are within the `lo-hi` range.

An invariant is that the search interval for a left-traversal at node v includes the maximum item in the subtree rooted at v.

This takes $O(k + \log n)$ time, where k is the number of points found. However, it takes $O(n \log n)$ time to pre-process (build) the tree, and $O(n)$ space.

If we instead only wanted to know the number of points within a range, we only need to augment the tree with the number of nodes in each subtree rather than traversing entire subtrees.

This generalizes to d-dimensions by storing a $d-1$-dimensional range tree in each node of a 1D range tree and constructing the $d-1$-dimensional range tree recursively.

However, rotations take $O(n)$, so we don't support `insert` and `delete`. In general, queries take $O(\log^d n + k)$, building the tree takes $O(n \log^{d-1} n)$, and the tree takes $O(n \log^{d-1} n)$ space.

5.7 (a, b) Trees

The value for a is constrained: $2 \leq a \leq \frac{b+1}{2}$.

Nodes in (a, b)-trees can store **multiple keys**, where $a - 1 \leq n_{\text{keys}} \leq b - 1$, with the exception of the root node which may have 1 key at minimum.

Therefore, internal nodes have between a and b children inclusive, leaf nodes have none, and the root node can have at least 2 and at most b children.

We can see that every non-leaf node must have one more child than its number of keys. These keys must be in sorted ascending order, and form a **key range**.

For a non-leaf node with k keys $v_1, v_2, \ldots v_k$ and $k+1$ children $t_1, t_2, \ldots t_{k+1}$:

$\forall v \in t_1,\ v \leq v_1$
$\forall v \in t_{k+1},\ v > v_k$
$\forall i \in [2, k],\ \forall v \in t_i,\ v \in (v_{i-1}, v_i]$

Additionally, all leaf nodes must be at the same depth.

B-trees are simply $(B, 2B)$-trees.

`search` is trivial: descend through sub-trees by comparing the key with the node's key range.

`insert` is trivial unless there is an overflow, in which case we `split` the node and insert the new key.

`split` takes the median key in a node and conjoins it with its parent, and the keys to the left and right of the median key become its children. The parent might overflow, and if so we `split` it again, and so on.

`delete` swaps the node to be deleted with its predecessor's or successor's leaf, before deleting. Then, the leaf node may become underfull. If the sibling can donate one key, we move the parent's key down and the sibling's key up. Otherwise, we `merge` both siblings.

`merge` moves the parent down between the two children and conjoins all of them into one node. The parent might become underfull, and if so we `merge` its parent, and so on.

All operations take $O(\log n)$ time.

6 Graphs

6.1 Topological Sort

A **topological sort** imposes a total ordering on all vertices, such that edges only point foward.

Only **directed acyclic graphs** have a topological order. Topological order is **not unique**.

6.1.1 Post-order Depth-first Search

Vertices are prepended to the list only after all of their children have been visited.

6.1.2 Kahn's Algorithm

Removes vertices without dependencies from the graph.

Each vertex tracks their in-degree, and is enqueued when it reaches zero. When dequeued, the vertex is removed and its children's in-degrees are decremented.

Both algorithms have a time complexity of $\mathbf{O(V + E)}$ and a space complexity of $\mathbf{O(V)}$.

6.2 Connected Components

Vertices v and w are in the same **connected component** iff there is a path from v to w.

Iterate over each vertex, and if it is unvisited, perform either BFS or DFS to flood-fill. This has a time complexity of $\mathbf{O(V + E)}$.

In the case of a directed graph, v and w are in the same **strongly connected component** iff there is a path from v to w and also from w to v.

6.2.1 Kosaraju's Algorithm

On the first pass, maintain a set of visited vertices and a stack of finished vertices. Explore vertices with DFS, pushing onto the stack when all of their children have been visited.

On the second pass, reverse the graph. Pop a vertex from the stack and flood-fill with DFS to get the SCC. Pop vertices from the stack until we get an unvisited vertex, then repeat the previous step.

This has a time complexity of **O(V + E)**, and a space complexity of **O(V)**.

6.3 Cycle Detection

A **cycle** is a path that visits a vertex more than once.

In a directed graph, keep parent pointers and sets for unvisited, visiting, and visited vertices. Flood fill vertices with DFS, and if we find a vertex in the visiting set, then there is a cycle.

7 Shortest Paths

7.1 Bellman-Ford Algorithm

Computes the shortest path for graphs with negative weights. Can be used to detect negative weight cycles.

Let $\delta(u, v)$ be the weight of the shortest path from u to v. By the triangle inequality, $\delta(s, v) \leq \delta(s, u) + \delta(u, v)$.

For each vertex v, we maintain an estimate $d(v)$ of the distance from a source s, and a parent pointer $\pi(v)$. All estimates are initialized to ∞, and $d(s) = 0$.

The algorithm performs $|V| - 1$ iterations over the graph, during which each edge is relaxed:

```
relax(u, v):
    if d(v) > d(u) + weight(u, v):
        d(v) = d(u) + weight(u, v)
        π(v) = u
```

Relaxation mainatains the invariant $d(v) \geq \delta(s, v)\ \forall v \in V$, i.e., we never under-estimate.

If an entire iteration does not change any estimates, then we can terminate early. After n iterations, the n-hop estimate on the shortest path (not every path!) is correct, i.e., $d(v_n) = \delta(s, v_n)$

Also, if P is the shortest path from s to v, and if P goes through u, then P is also the shortest path from s to u.

If we run a $|V|$-th iteration and the estimates still change, then we have found a negative weight cycle.

This algorithm has a time complexity of **O(EV)**.

7.2 Dijkstra's Algorithm

Computes the shortest path for graphs with only non-negative weights, including cyclic graphs.

For each vertex v, we maintain the distance $d(v)$ from a source s, and a parent pointer $\pi(v)$.

We also maintain a priority queue Q of vertices ordered by their distance s, i.e. the d-values. $d(s)$ is initialized to 0.

```
dijkstra():
    initialize() // O(|V| log |V|)
    while Q is not empty: // O(|V|)
        u = extract_min(Q) // O(log |V|)
        for each edge (u, v): // O(|E|)
            relax(u, v) // O(log |V|)
```

The relax operation is the same as in the Bellman-Ford algorithm, except that `decrease-key` is used to update the d-values.

The time complexity of this algorithm with an AVL tree is **O(E log V)**, but **O(E + V log V)** with a Fibonacci heap. The complexities are simplified as $|E| = \theta(|V|^2)$.

Graphs can be reweighted by multiplying the edge weights by a constant factor, but addition by a constant factor will break Dijkstra's algorithm.

We can also find the longest path by negating the edge weights, as long as the graph is acyclic.

If we wanted to find the path with shortest weight products rather than sum, it is easier to take the logarithm of the weights, rather than modify the relax operation.

7.3 Relaxation in Topological Order

Computes the shortest path for directed acyclic graphs.

Since the path is relaxed in order, the distance is correct after relaxation.

This has a time complexity of **O(V + E)**.

8 Union-Find

An abstract data type for disjoint sets and connected components.

`union` combines two objects, and `find` checks whether there is a path between two objects.

A naive implementation associates each object with an identifier. `find` checks whether their identifiers are the same in $O(1)$. `union` has to iterate over all identifiers and combine them in $O(n)$.

A better implementation associates each object with a parent. `find` now checks whether they have the same highest common ancestor (HCA) in $O(n)$. `union` has to find both their HCAs to link together in $O(n)$.

An optimization using **weighted union** associates a tree size with each object, so `union` links the smaller tree to the larger one in $O(\log n)$. Since the heights of the trees of size n is at most $\log(n)$, `find` takes $O(\log n)$ time.

The final optimization adds **path compression**. After each time the root is found, every node on the path is linked to the root. Both `union` and `find` take $O(\alpha(m, n))$ amortized time each for m operations, and $O(\log n)$ in the worst case.

With path compression but without weighted union, both `union` and `find` take $O(\log n)$ time.

9 Minimum Spanning Trees

An acyclic subset of edges connecting all vertices, with minimum total weight.

MSTs cannot be used to find shortest paths.

A **cut** of a graph partitions the vertices into two disjoint subsets. An edge crosses a cut if it has one vertex in each of these two subsets.

MSTs have four properties:

1. MSTs are acyclic.
2. If an MST is cut, both subtrees are MSTs.
3. For every cycle, the maximum weight edge is not in the MST, but the minimum weight edge may or may not be.
4. For every vertex (and cut), the minimum weight edge is in the MST.

9.1 Prim's Algorithm

We maintain a set S for the vertices in the MST, a priority queue Q of vertices ordered by their distance d from the growing MST, as well as a parent pointer $\pi(v)$ for each vertex v.

We initialize $d(v)$ to ∞ for every vertex v, and start at some arbitrary vertex s such that $S = \{s\}$ and $d(s) = 0$.

Then, we repeatedly dequeue the vertex u with the smallest distance d, removing it from Q and adding it to S.

For every edge (u, v) in the graph, we relax it as so:

```
relax(u, v):
    w = weight(u, v)
    if d(v) > w:
        d(v) = Q.decrease_key(u, w) // O(log |V|)
        π(v) = u
```

This works because each added edge is the minimum across some cut, therefore each edge is in the MST.

Every vertex is only added and removed once from Q, taking $O(|V| \log |V|)$ in total.

Each edge is only relaxed once (calling `decrease-key`), taking $O(|E| \log |V|)$ in total.

Therefore, this has a time complexity of **O(E log V)**.

9.2 Kruskal's Algorithm

We will use union-find to keep track of the vertices in the MST.

We sort the edges by weight in ascending order and iterating through them. If the edge connects two vertices in the same connected component (with `find`), we ignore it. Otherwise, we add it to the MST and call `union`.

This works because the ignored edge would have been the heaviest edge on the cycle, which violates the MST property. Also, all other lighter edges have already been considered.

This has a time complexity of **O(E log V)** since $\log |E| = O(\log |V|^2) = O(2 \log |V|) = O(\log |V|)$.

With integer weights, $O(n)$ counting sort may be used such that the time complexity is improved to $O(|E|)$. With a $O(n^2)$ sort, the time complexity is $O(|E|^2)$.

9.3 Variants

If all edge weights are equal, any spanning tree found by DFS or BFS will be an MST since both the spanning tree an MSTs have exactly $V - 1$ edges.

Reweighting a graph has no effect on the MST.

A directed MST is a much harder problem, but in the special case with one root, an MST is the tree of all edges with the minimum weight, which takes $O(E)$ time.

A maximum spanning tree can be found by negating each edge weight, or by running Kruskal's in reverse.

10 Dynamic Programming

Construct an optimal solution to a problem from optimal solutions to smaller subproblems.

10.1 SRTBOT

1. **S**ubproblem definition.
2. **R**elate subproblem solutions recursively.
3. **T**opological order on subproblems (DAG).
4. **B**ase case(s) for subproblems.
5. **O**riginal problem, solved via subproblems.
6. **T**ime complexity.

Use **memoization** to re-use solutions to subproblems with a dictionary mapping subproblems to their solutions.

Recursive function computes the solution if and only if it is not already stored.

If the the input is some arbitrary sequence x of length n, good subproblems to consider are:

1. the $\theta(n)$ **prefixes** $x[:i]$,
2. the $\theta(n)$ **suffixes** $x[i:]$, and
3. the $\theta(n^2)$ **substrings** $x[i:j]$.

The use of subsequences as subproblems, where some

elements can be omitted, is undesirable since there would be an exponential number of them.

10.2 Longest Common Subsequence

Given two subsequences A and B, find the longest common subsequence L of A and B.

When there are **multiple inputs**, subproblems can be generated by taking the cross product of the subproblem spaces.

S: $L(i,\ j) = \text{LCS}(A[i:],\ B[j:])\ \forall i \in [0, |A|]\ \forall j \in [0, |B|]$

R: $L(i,\ j) = \begin{cases} 1 + L(i+1,\ j+1) & \text{if} A[i] = B[j] \\ \max\{L(i+1,\ j),\ L(i,\ j+1)\} & \text{otherwise} \end{cases}$

T: for $i = |A| \dots 0$ for $j = |B| \dots 0$

B: $L(|A|,\ j) = L(i,\ |B|) = 0$

O: $L(0, 0)$

T: $\theta(|A| \cdot |B|)$ subproblems $\cdot\ \theta(1)$ each $= \theta(|A| \cdot |B|)$

We can use **parent pointers** to reconstruct the solution.

10.3 Longest Increasing Subsequence

Given a sequence A, find the longest strictly increasing subsequence L of A. Generalizes to non-strict as well.

S: $L(i) = \text{LIS}(A[i:])$ that starts with $A[i]$ (**constraint**).

R: $L(i) = 1 + \max\{L(j) \mid i < j < |A|,\ A[i] < A[j]\} \cup \{0\}$

T: for $i = |A| \dots 0$

B: $L(|A|) = 0$

O: $\max\{L(i) \mid 0 \le i \le |A|\}$

T: $\theta(|A|)$ subproblems $\cdot\ \theta(|A|)$ choices $+\ \theta(|A|) = \theta(|A|^2)$

10.4 All-Pairs Shortest Path (APSP)

Find the shortest path between all pairs of vertices in a graph, using the Floyd-Warshall algorithm.

First, number every vertex from 1 to $|V|$.

S: $d(u, v, k)$ = weight of the shortest path from u to v using only the vertices $\in \{u, v\} \cup \{1, 2, \dots, k\}$ $\forall u, v \in V\ \forall k \in [0,\ |V|]$

R: $d(u, v, k) = \min \begin{cases} d(u, v, k-1) & k \in SP \\ d(u, k, k-1) + d(k, v, k-1) & k \notin SP \end{cases}$

T: increasing k for $k = 0 \dots |V|$ for $u \in V$ for $v \in V$

B: $d(u, v, 0) = \begin{cases} 0 & \text{if } u = v \\ w(u, v) & \text{if } (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$

O: $d(u, v, |V|)$ assuming no negative weight cycles

T: $\theta(|V|^3)$ subproblems $\cdot\ \theta(1)$ each $= \theta(|V|^3)$

10.5 Partitioning (Rod Cutting)

Given a rod of length $L \in \mathbb{Z}^+$ and a value $v(l)$ for a rod of length $l \in [1,\ L]$, what is the max-value partition for the rod?

S: $X(l) =$ max-value partition of length l, $\forall l \in [0, L]$

R: $X(l) = \max\{v(p) + X(l-p) \mid 1 \le p \le l\}$

T: increasing l for $l = 0, 1 \dots, L$

B: $X(0) = 0$

O: $X(L)$

T: $\theta(L)$ subproblems $\cdot\ \theta(|L|)$ choices $= \theta(|L|^2)$

10.6 Subset Sum

Given a multiset $A = \{a_0, a_1, \dots, a_{n-1}\}$ with n positive integers, and a target sum T, does any subset $S \subseteq A$ sum to T?

This is a **decision problem** where the answer is binary.

S: $X(i, t)$ = does any subset $S \subseteq A[i:]$ sum to T $\forall i \in [0, n]\ \forall t \in [0, T]$?

R: $X(i, t) = \text{OR} \begin{cases} X(i+1, t) & a_i \notin S \\ X(i+1, t - a_i) \text{ if } a_i \le t & a_i \in S \end{cases}$

T: decreasing i for $i = n, n-1, \dots 0$

B: $X(n, t) = \begin{cases} \text{true} & \text{if } t = 0 \\ \text{false} & \text{otherwise} \end{cases}$

O: $X(0, T)$

T: $\theta(nT)$

This runs in **psuedo-polynomial** time.