Shortest Path

- Shortest path in **DAG**: Toposort + relax in order O(V + E)
- Longest path in **DAG**: Multiply all edges by -1, find shortest path, multiply edges by -1 again - O(V + E), OR Toposort + modify the relax function by taking dist[v] = max(dist[v], dist[u] + weight(u, v))
- BFS/DFS on a Tree: O(V+E) = O(V+V-1) = O(V)
- Shortest path in **directed** graph (with cycles) where number of hops is divisible by 3: Duplicate each node 3 times to store (A, x mod 3). Then run shortest path, Dijkstra's - O(m log n), from (start, 0) to (end, 0) - O(n + m) to build the new graph containing 3n nodes and 3m edges.
- Shortest path in **directed** graph (with cycles) where number of hops is divisible by k: Duplicate each node k times - (A, x mod k), run Dijkstra's - O(km log kn)
- Shortest path in **DAG** where number of hops is divisible by k: duplicate each node k times, run DAG_SSSP - O(km + kn) toposort + relax in order
- Dijkstra's: O((V + E)log V) - V*insert, V*extract, E*decreaseKey (with binary heap). With fibo heap – O(E + V log V). With list O(V^2)
- Bellman-Ford: O(VE)
- Shortest path in **unweighted** graph: BFS O(V + E)
- Floyd-Warshall APSP: O(V^3). In sparse graphs, O(E)=O(V), run Dijkstra's N times is faster – O(EVlog V)=O(V^2 log V)
- Longest path in weighted complete graph with non-negative edge weights: Equivalent to TSP because the longest path always includes all vertices
- Topological sort: Post-order DFS or Kahn's algorithm. Pre-order DFS does not work because children are not guaranteed to be visited in the correct order. Kahn's algorithm: keep extracting nodes with indegree of 0 and updating the indegree of 'to' nodes. Can detect cycles. Use a normal queue to keep pushing nodes of indegree 0 – O(V+E)
- Shortest path in DAG passing through edge of weight > k: Duplicate graph, nodes in first graph point to second graph if edge > k. If pass through two edges > k – duplicate the graph twice. To find whether a path exists, check if destination node in 2nd graph can be reached
- Shortest path in DAG passing through *at most* k edges: Create k more copies of the graph and add edge from copy (u, i) to (v, i+1) for each edge (u, v) in the original graph, and also add zero-cost edges from (u, i) to (u, i+1) to account for paths using less than k edges. Then run Dijkstra's taking O(km log kn)=O(m log n)
- Shortest path in DAG passing through *exactly* k edges: Same as above, but no zero-cost edges.
- Longest path in undirected, unweighted graph where index of each node in a path is strictly increasing: Convert undirected edges into directed edges (u, v) for each pair of nodes where (u < v). Then just find the longest path in the DAG.
- Find minimax distance from source to destination (minimize the maximum weight of each edge on a path) less than or equal to k:

Method	Time	Space
Binary search on values of k, run modified BFS/DFS (O(V+E)) ignoring all weights > k.	Preprocess: 0 Query: O((V+E)log k)	O(V) for BFS/DFS stack
Preprocess the component table for the graph for all values of k. Find if source and dest are in the same component in O(1). Binary search on values of k for each query	Preprocess: O(k(V+E)) Query: O(log k)	O(kV) for k component tables. Use when k is small
Modified Dijkstra's - if previous max edge along path to u is bigger than max edge in current path to u, update d[u] to the new (smaller) max weight. Max weight along path is monotonically increasing, so can handle negative weights	Preprocess: 0 Query: O((V+E)log V)	O(V) for query
Exhaustively run modified Dijkstra's on all vertices	Preprocess: O(V(V+E)logV)Query: O(1)	Preprocess O(V^2) Query O(1)
Floyd-Warshall – use depending on whether graph is sparse or dense	Preprocess: O(V^3) Query: O(1)	Preprocess O(V^2) Query O(1)
Obtain MST from Prim's/Kruskal's, run BFS/DFS on MST to obtain minimax dist from S to D by keeping track of max weighted edges along the way	Preprocess: 0 Query: O(E log V)	Preprocess 0 Query O(V) for Prim's
Preprocess the MST, run BFS/DFS for each query	Preprocess: O(E log V) Query: O(V)	Preprocess O(V) Query O(V)
Preprocess MST + Binary Search + LCA. For each vertex in T, add O(log V) ancestral skips point to it and precompute lineage widths to them. Query stage: find LCA, find width of path from S to LCA, and D to LCA	Preprocess: O(E log V + V^2) Query: O((log v)^2)	Preprocess O(V log V) Query O(1)

- Shortest Path to destination in DAG passing through mandatory vertex v: Create duplicate graph with all edges reversed, then run SSSP from mandatory vertex v
- Shortest path from source to destination in DAG with edge weights only 1 and 2: Split each edge of weight 2 into 2 edges of weight 1, then run DFS/BFS
- Shortest path from source to dest in grid where only right turns are allowed and path length is number of right turns taken: Duplicate each node 4 times, keeping track of both the location and the direction that the car is facing. Then just run SSSP from the source to dest node. Non-graph solution (not correct): the number of turns required is just the quadrant that the destination node is in, relative to the source node. Top right = 1 turn, bottom right = 2 turns, bottom left = 3 turns, top left = 4 turns. Might not produce the correct number of turns

Important Notes

- DAG_SSSP cannot be used if there are cycles in a graph
- Graphs with cycles don't have a topological order
- Unique topological ordering means all vertices have an edge to their next node (Hamiltonian path)
- DFS does not find the shortest path
- BFS does not work with weighted graphs to find shortest path
- Longest path with positive weight cycles has no solution
- Longest simple path (no cycles) is NP-hard, no efficient solution
- Each state in a puzzle can be represented by one node, edges are unweighted. Use BFS to find shortest path from unsolved to solved
- Each edge relaxed in Dijkstra's at most once. est[u] >= shortest dist from s to v. BF can be terminated early if one round of relax operations does not reduce any estimate further.
- Bellman-Ford is guaranteed to give correct estimates after (n-1) iterations, but (n-1) is not the **smallest** iteration possible.
- Shortest path can be found in DAGs, even if there are negative weights
- Problems requiring state transitions can be modelled effectively with graphs

MST

- Prim's with AVL PQ: O((E+V) log V) - bottleneck is (cost of decreaseKey) * (# of times of decreaseKey)
- Prim's with adj matrix: O(n^2 + mlog n + nlog n) each node has to scan V-1 neighbours at worst. m*(PQ operations, log n). n*(Extract min, log n)
- Prim's with normal queue: decreaseKey O(1), findMin O(n), insert/remove n nodes = O(n^2), m * O(1) for decrease key. Overall O(n^2 + m)
- Kruskal's: O(E log E) = O(E log V) - bottleneck is sorting
- Bounded edge weights (e.g. 0..10): Kruskal's with counting sort - O(E), Prim's with fixed size PQ – O(E), cost of decreaseKey() decreased to O(1)
- Max Product Spanning Tree weights bounded 0 to 1: (tut9) Take negative log of weights and run MST OR modify Prim's with max heap to take next biggest edge – O(E log V)
- Minimum Product Spanning Tree: Take log weights and run SSSP
- Tree Properties: **(P1)** No cycles. **(P2)** If you cut an MST, the 2 pieces are MSTs. **(P3)** For every cycle, the max weight edge **IS NOT** in the MST. **(P4)** For every partition, the min weight edge across the cut **IS** in the MST.
- For every vertex, the min outgoing edge **IS** in the MST (Boruvka)
- The smallest edge of a graph **IS NOT** guaranteed to be in the *shortest path* graph.
- For every vertex, the max outgoing edge might sometimes be in the MST (simple example - star graph)
- The smallest edge *in a cycle* **IS NOT** guaranteed to be in the MST. Can happen if two cycles share an edge.

Hashing

- Chaining: expected O(1) + n/m (items/bucket) search **with S*imple Uniform Hashing Assumption*** (SUHA), worst case O(n). Insert O(1) append to back of linked list. Expected max chain length: O(log n), θ(log n/log log n). Using 2 hash functions, max chain length is O(log log n). When (n == m), still can insert items and search efficiently
- Open Addressing: Expected cost of searching = 1/(1-a), a = n/m, ***with Uniform Hashing Assumption***. If no uniform hashing, worst case search is O(n). If table is 1/4 full, there will be clusters of size θ(log n). When (n == m), cannot insert any more items and cannot search efficiently. Chaining outperforms OA as table gets more full. Tombstone values (e.g. DEL) needed to mark deleted items so that search does not produce false negatives.

- SUHA vs UHA: **SUHA** means each key has an equal probability of mapping to each bucket. **UHA** means that the probe sequence is assumed to be a random permutation of [0..m-1]
- Table Resizing: (**Inserting**) If (m2 == m1 + 1) - O(n^2) resize. If (m2 == m1^2) - O(n^2). If (m2 == 2*m1) - O(n) resize, O(1) amortized cost of insertion. Generalized for a constant k, if (m2 == k*m1), cost to resize is O(n), cost to insert is amortized O(1). (**Deleting**) If (m2 == m1/4), amortized O(1) delete
- Amortized cost: if every insert and search operation takes *amortized* O(1), *any* sequence of *k* insert and *l* search operations will take O(*k* + *l*) time. Amortized cost is not average cost – the current total cost / number of ops must be <= k at all times for amortized O(k)
- Cuckoo Hashing: Insertion expected O(1) amortized, Lookup O(1) worst case. Uses 2 hash tables and 2 hash functions. Inserting new item will 'kick out' previous items into the other hash table. If caught in an infinite cycle, need to resize and rehash. Expected O(log n) resizing operations for n inserts. Amortized O(1/n) per insertion for rehashing. Better than chaining (uses less space, less overhead than linked list), provides O(1) insert and delete
- Probability that hash function has **no** collisions: m!/m^n*(m-n)!
- Fingerprint Hash Table (FHT): Doesn't store key, only store 0/1 vector. Reduced space (only store 1 bit per slot), but more space (bigger table) needed to reduce chance of collision
- Probability of no false positive in a Fingerprint Hash Table (FHT): (1-1/m)^n ≈ (1/e)^(n/m). To achieve rate of false positive < p, we need n/m ≤ log(1/(1-p))
- Expected number of pairs of distinct keys that collide: n(n-1)/2m
- Collision resolved by chaining, probability that last 5 buckets are empty after three insertions into table of size 50: (9/10)^3. If OA was used instead, it would be (45x44x43)/50^3
- Probability that 3 keys hash to the same bucket under SUHA: 1/m^2
- 1/2 probability of hashing to first 1/4 of bucket, 1/2 probability of hashing to last 3/4 of bucket. Collision resolved by chaining: Expected length of chain: n/2 items are in first 1/4 buckets, n/2 items are in last 3/4 buckets. Expected length of chain = (n/2)/(1/4) * 1/2 + (n/2)/(3/4) * 1/2 = n + n/3 = (4/3) * n

Union Find

	find	*union*
Quick-find	O(1)	O(n)
Quick-union	O(n)	O(n)
Weighted-union	O(log n)	O(log n)
Path compression	O(log n)	O(log n)
Weighted-union with path compression	O(a)	O(a)

- Union only connects two trees together, path compression is done during find() operation. If path has not been compressed, find() will take worst case O(log n) instead of O(a(n))
- WU+PC: For m union/find operations on n objects, O(n + m*a(m, n))

Heaps

- Bubble down from n-1 to 0: O(n). TC = summation(nodes * height).

$$\sum_{i=0}^{\log N} \frac{N}{2^i} \log\left(\frac{N}{2^i}\right) \leq N \log N.$$

- Bubble up from 0 to n-1: O(n log n) - (n/2^i) nodes at each level, which can bubble up at most (height-1) = log(n/2^i). TC = Summation(nodes * log(nodes))

$$\sum_{i=0}^{\log N} \frac{N}{2^i} \cdot i \leq N \sum_{i=0}^{\infty} \frac{i}{2^i} = N \cdot 2$$

- ^ This is the same as performing an operation on each node of a binary tree where the cost of performing an operation on the node is equal to the height of the node
- Heap sort: extract O(log n) * n times = O(n log n)
- Left child: 2n+1, Right child: 2n+2, Parent: floor((n-1)/2)
- Last level is full, children as far left as possible. Priority[parent] >= priority[children]
- Amortized cost for operation on each node of perfectly balanced binary tree where operation takes time equal to number of descendants of node: θ(n log n / n) = θ(n)
- ^ when operation takes time equal to height of node: θ((log n)^2 / n) = θ(1)

Dynamic Programming

- Optimal Substructure: Big problem can be broken down into smaller problems
- Overlapping Subproblems: Each subproblem is used to solve *multiple* bigger problems
- LIS: Run DAG + bfs on all nodes – n*O(V+E)=n*O(n^2)= O(n^3). DP solution: S[i] = max(S[j]) + 1. Base case: S[n] = 0
- Lazy Prize Collecting: P(v, k) = prize collected at node v, taking k steps. P(v, 0) = 0. P(v, k) = max(P(w1, k-1) + weight(v, w1), P(w2, k-1) + weight(v, w2)...)). k rows, each row takes E time to solve – O(kE)
- Vertex Cover: Each node is either in the vertex cover at the subtree rooted at v, or not in the vertex cover – 2V subproblems. O(V) to solve each subproblem. S(v, 0) = size of subtree if node not in vertex cover. S(v, 1) = size of subtree if node in vertex cover. S(leaf, 0) = 0. S(leaf, 1) = 1. S(v, 0) = sum(cover all children). S(v, 1) = 1 (to cover v) + sum(children either covered or uncovered, whichever is smaller).
- Floyd-Warshall APSP: Run Dijkstra's on every node: O(VElog v). APSP – O(V^3) (refer to shortest path section).
- Longest Sawtooth Subsequence: A(i) = A(i-1) if xi < xi-1, else 1 + D(i-1). D(i) = D(i-1) if xi > xi-1, else 1 + A(i-1). Base cases: A(1)=1, D(1)=1

Others

- Replacing k elements (k << n) in a sorted array and running insertion sort takes O(k*n)=O(n)
- Build AVL tree from sorted array - O(n), pick median as root
- Worst case insertion sort on array with 2 elements: O(n^2) - e.g. 01010101
- Quicksort 3 way partitioning on array with k duplicates: O(nk log n)
- 3/4 weight balanced tree is not height balanced (e.g. 8/6/1)
- 3/4 weightbalanced tree is balanced (h = log n). Recurrence: h(n) = 1 + h(3/4n). Height is bounded by log(base 4/3)n + 1
- Maintain AVL tree sorted by names with insert, delete, find

Time Complexity

Sterling's Approximation: O(log(n!)) = O(nlog n)
O(sqrt n) > O((log n)^2) > O(log n)
O(n^(1+1/logn)) = O(n) as 1/log n tends to infinity
T(n) = T(n/3) + 7n = O(n)
T(n) = 3T(n/3) + 3n = O(n log n)
T(n) = 2T(n/2) + O(n) = O(n log n)
T(n) = 2T(n/2) = O(n)
T(n) = 2T(n/2) + O(1) = O(n)
T(n) = T(n/c) + O(1) = O(log n) for c > 1
T(n) = T(n/2) + O(n) = O(n)
T(n) = 2T(n-1) + O(1) = O(2^n)
T(n) = 2T(n/2) + O(n log n) = O(n(log n)^2)
T(n) = 2T(n/4) + O(1) = O(sqrt n)
T(n) = T(n-c) + O(n) = O(n^2)

Master's Theorem

T(n) = aT(n/b) + cn^k, T(1) = c
If (a < b^k), T(n) = θ(n^k)
If (a == b^k), T(n) = θ(n^k log n)
If (a > b^k), T(n) = θ(n^(log base b a))

Augmented Trees

- Order Statistic Tree: Find order of element in array. Augment: store size of subtree rooted at node. O(log n) insert and find. Sorted array O(n) insert, O(1) find. Unsorted array O(1) insert, O(n) find.
- Interval Tree: Find interval that point exists in. Augment: Sort by left endpoint, store the maximum right endpoint for subtree rooted at each node. To find all overlaps, run search k times – O(k log n). Best known solution O(k + log n) with canonical subsets
- Range Query: Find all points/nodes within a given range. Solution 1: build x-tree with each node pointing to a y-tree with all the same nodes in that subtree but sorted by y-coordinates. O(log^2 n + k) query time. O(n log n) space. Building tree O(n log n). Insert/delete not supported, because each rotation takes O(n). K-dimensional trees query O(log^d n), buildTree O(n log^(d-1) n), space O(n log^(d-1) n). Applications: database searching
- Trie: O(L) search where L is the length of the string. Uses O(text size) space. Trie tends to use more space than AVL tree, but performs faster than AVL tree. Use special characters/flags node to indicate the end of a trie