* A Tree is a ‘undirected graph with no cycles’ aka a ‘connected graph with n vertices and n-1 edges’
* A rooted tree is a tree with designated root node where every node either points away (out-tree) or towards this node (in-tree).
* DAG – ‘directed graphs with no cycles’. All out trees are DAGs but not all DAGs are out trees
* A Bipartite graph – whose vertices can be split in to 2 groups U and V so that **every edge** connects between U and V. These graphs are two colourable. Graphs with ‘**no odd length cycle**’
* Complete graph – a graph where there is a ‘unique edge’ between every pair of vertices. i.e. we have edges form every vertex to another vertex. If we need to test code for worst case, test it on complete graph as it has lots of edges.
* Strongly Connected Components (SCC) = Self Contained Cycles: in a ‘directed graph’ where *every* vertex in a given cycle can reach *other every other vertex in same cycle*.
* Bridge (cut) – is any edge with its removal increases the number of connected components.
* Articulation Point (Cut vertex) – any node in the graph whose removal increases number of connected components.
* Minimum Spanning Tree – is a subset of edges of connected & edge weighted graph that ‘still’ connects all vertices together without cycles and with minimum possible edge weight.

|  |  |
| --- | --- |
| **Adjacency Matrix:**   1. Edge weight lookup is O (1) 2. Requires O(V2) space though very space efficient for **dense** graphs (lots of edges) 3. Iteration over all edges takes O(V2) time | **Adjacency List:**   1. Edge weight lookup is O(E) 2. Less space efficient for dense graphs, i.e. space efficient for sparse (less edges) graphs. 3. Iteration is efficient. |

|  |  |  |
| --- | --- | --- |
| **Shortest Path:**   * BFS (unweighted) * Dijkstra’s * Bellman Ford * Floyd Warshall | **Connectivity:**   * DFS * Union and Find |  |
| **Min Spanning Tree:**   * Kruskal * Prims * Boruvka’s | **Network Flow:**   * Ford Fulkerson * Edmonds-Karp * Dinic’s |
| **Negative Cycles:**   * Bellman Ford * Floyd Warshall | **Strongly Conn Comp**   * Trajan’s * Kosaraju |
| **Travelling Salesman**   * Held-Karp, Branch & Bound, Ant Colony Optimi. | |

**DFS:**

Time complexity: O(V+E)

|  |  |
| --- | --- |
| **private** **void** dfs(**int** at, **boolean**[] visited) {  visited[at] = **true**;  **for** (**int** adjVert : adjList.get(at)) {  **if** (visited[adjVert] == **false**) {  dfs(adjVert, visited);  }  }  } |  |
| **Connected Components by DFS:**  **void** connectedComponents() {  **boolean**[] visited = **new** **boolean**[n];  **int** color = 1;  Map<Integer, List<Integer>> map = **new** HashMap<>();  **for** (**int** i = 0; i < n; i++) {  **if** (visited[i] == **false**) {  dfs(i, visited, color++, map);  }  }  }  **private** **void** dfs(**int** at, **boolean**[] visited, **int** color, Map<Integer, List<Integer>> map) {  visited[at] = **true**;  **if** (map.get(color) == **null**) {  List<Integer> l = **new** ArrayList<>();  l.add(at);  map.put(color, l);  } **else** {  List<Integer> l = map.get(color);  l.add(at);  map.put(color, l);  }  **for** (**int** adjVert : adjList.get(at)) {  **if** (visited[adjVert] == **false**) {  dfs(adjVert, visited, color, map);  }  }  } | |

**BFS:**

Time complexity: O(V+E)

|  |
| --- |
| **void** bfs(**int** u, **int** v) {  **boolean**[] visited = **new** **boolean**[n];  **int**[] prev = **new** **int**[n];  Arrays.*fill*(prev, -1);  Queue<Integer> q = **new** ArrayDeque<>();  q.add(u);  visited[u] = **true**  **while** (!q.isEmpty()) {  **int** currVert = q.poll();  **for** (**int** adjVert : adjList.get(currVert)) {  **if** (visited[adjVert] == **false**) {  q.add(adjVert);  visited[adjVert] = **true**;  prev[adjVert] = currVert;  }  }  }  } |
| **Grid Search BFS**  **int**[][] dir = **new** **int**[][] { { -1, 0 }, { +1, 0 }, { 0, -1 }, { 0, +1 } };  Queue<Point> q = **new** ArrayDeque<>();  q.add(a);  **boolean**[][] visited = **new** **boolean**[grid.length][grid[0].length];  visited[a.x][a.y] = **true**;  **while** (!q.isEmpty()) {  Point p = q.poll();  **if** (p.equals(b)) {  **break**;  }  **for** (**int** i = 0; i < 4; i++) {  **int** newR = p.x + dir[i][0];  **int** newC = p.y + dir[i][1];  **if** (newR < 0 || newC < 0 || newR >= grid.length || newC >= grid[0].length) {  **continue**; // as its out of bounds.  } **else** **if** (grid[newR][newC] == 'X') {  **continue**;  **else** **if** (grid[newR][newC] == '.' && visited[newR][newC] == **false**) {  Point newPoint = **new** Point(newR, newC);    prev.put(newPoint, p);  visited[newR][newC] = **true**;  q.add(newPoint);  }  }  } |

**Rooting a Tree (from Graph)**

|  |
| --- |
| **private** **static** TreeNode rootTree(List<List<Integer>> graph, **int** rootNode) {  TreeNode root = **new** TreeNode(rootNode);  root = *buildTree*(root, graph);  **return** root;  }  **private** **static** TreeNode buildTree(TreeNode node, List<List<Integer>> graph) {  **for** (**int** eachAdjVertex : graph.get(node.id)) {  **if** (node.parent != **null** && node.parent.id == eachAdjVertex) {  // System.out.println("Skipping loop causing stuff...");  **continue**;  }  TreeNode newChild = **new** TreeNode(eachAdjVertex);  node.children.add(newChild);  newChild.parent = node;  *buildTree*(newChild, graph);  }  **return** node;  } |

Centre of the Tree is always middle node of a longest Path. Another option is like an onion, peel off all leaf nodes layer by layer and then you will be left at centre of the tree. Starts from outside in until you left with one or two nodes. Compute degree of each node (leaf nodes has 1)

|  |
| --- |
| **private** **static** List<Integer> findTreeCenters(List<List<Integer>> tree) {  **int** n = tree.size();  **int**[] degrees = **new** **int**[n];  List<Integer> leaves = **new** ArrayList<>();  **int** counter = 0;  **for** (List<Integer> eachNodesAdj : tree) {  **if** (eachNodesAdj.size() <= 1) {  leaves.add(counter);  degrees[counter++] = 0;  } **else** {  degrees[counter++] = eachNodesAdj.size();  }  }  **int** processedLeavesCount = leaves.size();  **while** (processedLeavesCount < n) {  Set<Integer> newLeaves = **new** HashSet<>();  **for** (Integer eachLeaf : leaves) {  **for** (Integer eachLeafAdjNode : tree.get(eachLeaf)) {  degrees[eachLeafAdjNode] = degrees[eachLeafAdjNode] - 1;  **if** (degrees[eachLeafAdjNode] == 1) {  newLeaves.add(eachLeafAdjNode);  }  }  degrees[eachLeaf] = 0;  processedLeavesCount += newLeaves.size();  leaves = **new** ArrayList(newLeaves);  }  }  **return** leaves;  } |

|  |  |
| --- | --- |
| **public** **static** String encode(TreeNode node) {  **if** (node == **null**) {  **return** "";  }  List<String> labels = **new** ArrayList<>();  **for** (TreeNode eachAdjNode : node.children) {  labels.add(*encode*(eachAdjNode));  }  Collections.*sort*(labels);  StringBuilder sbr = **new** StringBuilder();  **for** (String label : labels) {  sbr.append(label);  }  **return** "(" + node.id + sbr.toString() + ")";  } |  |

**Topological Sort:**

A graph with cycles cannot have top ordering. Only DAGs can have top ordering. We could use Trajan’s strongly connected components to find cycles. All rooted trees can have top ordering as they cannot have cycles.

Topological sort time complexity: O(V+E)

|  |
| --- |
| **DFS Style:** O(V+E)  **private** List<Integer> topologicalSort(Map<Integer, List<Edge>> graph, **int** n) {  **boolean**[] visited = **new** **boolean**[n];  List<Integer> l = **new** ArrayList<>();  **for** (**int** i = 0; i < n; i++) {  **if** (visited[i] == **false**) {  *DFS*(i, visited, graph, l);  }  }  **return** l;  }  **private** **void** DFS(**int** node, **boolean**[] visited, Map<Integer, List<Edge>> graph, List<Integer> l) {  visited[node] = **true**;  **for** (Edge adjEdgeOfNode : graph.get(node)) {  **if** (visited[adjEdgeOfNode.to] == **false**) {  *DFS*(adjEdgeOfNode.to, visited, graph, l);  }  }  l.add(0, node);  } |
| **Kahn’s Algorithm :** O(V+E)  **void** topologicalSort() {  **int**[] inbounds = **new** **int**[vertexCount];  // create an array if inbounds.  **for** (**int** i = 0; i < vertexCount; i++) {  **for** (Integer adjVert : adj[i]) {  inbounds[adjVert]++;  }  }  // queue all vertices whose inbound = 0  Queue<Integer> q = **new** LinkedList<>();  **for** (**int** i = 0; i < inbounds.length; i++) {  **if** (inbounds[i] == 0) {  q.add(i);  }  }    List<Integer> topolSort = **new** ArrayList<>();  **int** processedVertices = 0;  **while** (!q.isEmpty()) {  Integer ver = q.poll();    topolSort.add(ver);  **for** (**int** eachAdjVer : adj[ver]) {  inbounds[eachAdjVer]--;  **if** (inbounds[eachAdjVer] == 0) {  q.add(eachAdjVer);  }  }  processedVertices++;  }  **if** (processedVertices != vertexCount) {  System.***out***.println("No way jose!!! Topolgical Sort Impossible");  } **else** {  System.***out***.print(topolSort);  }  }  } |

**(SSSP) Single Source Shortest Paths on DAGs:**

1. Top sort approach:

All trees are automatically DAGs. However, since trees does not have directed edges we cannot call them as DAGs. Thing about DAGs is **single source** shortest path can be solved on DAGs with **O(V+E)** as nodes can be ordered by top sort and processed sequentially. This is the *bestest we can get* (infact linear time). Next best is Dijkstra’s (may not work for negative edge weights). But below approach works for both +ve and -ve weights. In general, longest path is NP-Hard but for DAGs it can be done in **O(V+E).** Multiply all edges by -1, do shortest path and again multiply by -1.

|  |
| --- |
| **private** **static** Integer[] dagShortestPath(Map<Integer, List<Edge>> graph, **int** start, **int** n) {  **int**[] topSort = *topologicalSort*(graph, n);  Integer[] dist = **new** Integer[n];  dist[start] = 0;  **for** (**int** i = 0; i < n; i++) {  **int** eachNodeInTopSortOrder = topSort[i];  **if** (graph.get(eachNodeInTopSortOrder) != **null**) {  **for** (Edge eachAdjEdge : graph.get(eachNodeInTopSortOrder)) {  **int** newDistance = eachAdjEdge.weight + dist[eachNodeInTopSortOrder];  **if** (dist[eachAdjEdge.to] != **null**) {  dist[eachAdjEdge.to] = Math.*min*(dist[eachAdjEdge.to], newDistance);  } **else** {  dist[eachAdjEdge.to] = newDistance;  }  }  }  }  **return** dist;  } |

1. Dijkstra’s Shortest Path:

This is also a **single source** shortest path but ***only works for non-negative edge weights***. Based on implementation O(E\*log(V)) is fairly competitive among other sssp algorithms. However using such as Binary Heaps could bring down to O( (E + V) Log(V)) and Fibonacci Heaps even could go down to O(E + V Log(V))

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| **public** **double**[] dijkstra(**int** start) {  **double**[] dist = **new** **double**[n];  Arrays.*fill*(dist, Double.***POSITIVE\_INFINITY***);  PriorityQueue<Node> pq = **new** PriorityQueue<Node>(**new** Comparator<Node>() {  **public** **int** compare(Node node1, Node node2) {  **if** (Math.*abs*(node1.value - node2.value) < 1e-6;)  **return** 0;  **return** (node1.value - node2.value) > 0 ? +1 : -1;  }  });  pq.add(**new** Node(start, 0));  **boolean**[] visited = **new** **boolean**[n];  dist[start] = 0;  visited[start] = **true**;  **while** (!pq.isEmpty()) {  Node currNode = pq.poll();    **if** (dist[currNode.id] < currNode.value) {  **continue**;  }  // get all adj nodes and ensure they are not visited  visited[currNode.id] = **true**;  List<Edge> adjEdgesFromCurrNode = **this**.graph.get(currNode.id);  **for** (**int** i = 0; i < adjEdgesFromCurrNode.size(); i++) {  Edge currEdge = adjEdgesFromCurrNode.get(i);  **if** (visited[currEdge.to]) {  **continue**; // already visited so don’t touch again.  }  **double** newCost = currEdge.cost + dist[currEdge.from];  **if** (newCost < dist[currEdge.to]) {  dist[currEdge.to] = newCost;  pq.add(**new** Node(currEdge.to, newCost));  }  }  }  **return** dist;  } |

1. Bellman Ford Algorithm

Certainly, one of the easiest single sources shortest path BUT not ideal for most of the problems due to high complexity of O(EV) hence suggested to use Dijkstra’s. However, BF can used for negative edge weight and this BF can also finds out says where those negative cycles are (Dijkstra’s can’t be used for -ve edges)

|  |
| --- |
| **private** **static** **double**[] bellmanFord(List<Edge>[] graph, **int** v, **int** start) {  **double**[] dist = **new** **double**[v];  Arrays.*fill*(dist, Double.***POSITIVE\_INFINITY***);  dist[start] = 0;  **for** (**int** i = 0; i < v - 1; i++) { // relax each edge v-1 times  **for** (List<Edge> list : graph) {  **for** (Edge eachEdge : list) {  **if** (dist[eachEdge.from] + eachEdge.cost < dist[eachEdge.to]) {  dist[eachEdge.to] = dist[eachEdge.from] + eachEdge.cost;  }  }  }  }  **for** (**int** i = 0; i < v - 1; i++) { // repeat same to find out negative cycles  **for** (List<Edge> list : graph) {  **for** (Edge eachEdge : list) {  **if** (dist[eachEdge.from] + eachEdge.cost < dist[eachEdge.to]) {  dist[eachEdge.to] = Double.***NEGATIVE\_INFINITY***;  }  }  }  }    **return** dist;  } |

**(APSP) All Pairs Shortest Paths on DAGs:**

1. Floyd Warshall Algorithm:

Not good unless you got **just few** hundred of vertices as complexity is O(V3) with space complexity of O(V2), hence not so widely used due to V3 comp. however this is all pair shortest path and also deals negative cycles too. Logic: I-J is better than I->K + K->J, moreover it works best for adjMatrices only.

|  |
| --- |
| **private** **void** floydWarshall() {  **int**[][] dist = **new** **int**[n][n];  **int**[][] next = **new** **int**[n][n];  // initialize the array. could have been done in createGrpah  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < n; j++) {  dist[i][j] = adjList[i][j];  }  }  // IF i->k plus k -> j is less than direct path of i->j then use it  **for** (**int** k = 0; k < n; k++) {  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < n; j++) {  **if** (dist[i][k] + dist[k][j] < dist[i][j]) {  dist[i][j] = dist[i][k] + dist[k][j];  next[i][j] = next[i][k]; // better to go i->k than i->j  }  }  }  }  // see still if there are negative cycles  **for** (**int** k = 0; k < n; k++) {  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < n; j++) {  **if** (dist[i][k] + dist[k][j] < dist[i][j]) {  dist[i][j] = -1 \* ***INF***;  next[i][j] = -1;  }  }  }  }  } |

**Strongly Connected Components – Self Contained Cycles:**

Strongly Connected Components = Self Contained Cycles. These are within a ‘**Directed Graph**’ every vertex in a given cycle can reach every other vertex in same cycle. SCCS in a graph are unique. The low link value of a node is the smallest node id that can be reachable from that node when doing DFS (including itself). Depending WHERE the DFS starts and the order it traverses the low link values could get wrong hence we need to maintain an invariant. To cope with random traversal, Tarjan’s method keep a set (often a stack) of valid nodes from which to update low-link values from. Nodes are added to stack of valid nodes as they are explored for the first time. Nodes are removed each time a SCC is found. To update U’s low link value as V’s low link value, there has to be a path from U->V and V must be on the stack. If we get low-lin values during DFS we get low links in O(V+E)

|  |  |
| --- | --- |
|  | **private** **void** dfs(**int** at) {  visited[at] = **true**;  ids[at] = lows[at] = id++;  stack.push(at);  **for** (**int** to: **this**.graph.get(at)) {  **if** (ids[to] == -1) {  dfs(to);  }  // 0 -> 1 -> 2 -> 0, seen a visited node called 0.  **if** (visited[to] == **true**) {  lows[at] = Math.*min*(lows[at], lows[to]);  }  }  // On recursive callback, if we're at the root node (start of SCC). Empty the seen stack until back to root. Started at 0, came back from 2->1->0  **if** (ids[at] == lows[at]) {  **for** (**int** node = stack.pop();; node = stack.pop()) {  visited[node] = **false**;  sccs[node] = sccCount;  **if** (node == at) { **break**;  }  }  sccCount++;  }  } |

|  |
| --- |
| **private** **void** getSccs() {  ids = **new** **int**[n];  lows = **new** **int**[n];  sccs = **new** **int**[n];  visited = **new** **boolean**[n];  stack = **new** ArrayDeque<>();  Arrays.*fill*(ids, -1);  **for** (**int** i = 0; i < n; i++) {  **if** (ids[i] == -1) {  dfs(i);  }  }  Map<Integer, List<Integer>> m = **new** HashMap<>();  **for** (**int** i = 0; i < n; i++) {  **if** (!m.containsKey(sccs[i])) {  m.put(sccs[i], **new** ArrayList<>());  }  m.get(sccs[i]).add(i);  }  System.***out***.println(m.values());  } |

**Existence of Eulerian paths and circuits: Hierholzer’s algorithm**

A Eulerian path/trial is a path of edges that visits **all the edges in a graph exactly once**. We can find eulerian path only if we start at right vertex. Eulerian circuit is an eulerian path which starts and ends on the same vertex. Starting at wrong node may not lead you eulerian cycle. However, if you know graph has an eulerian cycle then you can start any node (does not matter as long as it has eulerian cycle)

|  |  |
| --- | --- |
|  | **private** **void** setup() {  **this**.in = **new** **int**[n];  **this**.out = **new** **int**[n];  **for** (**int** at = 0; at < n; at++) {  **for** (**int** adj : graph.get(at)) {  out[at]++;  in[adj]++;  edgeCount++;  }  }  }  LinkedList<Integer> path = **new** LinkedList<>();  **int** edgeCount = 0; |

|  |  |
| --- | --- |
| List<Integer> **getEulerianPath**() {  setup();  **if** (!graphHasEulerianPath())  **return** **null**;  dfs(findStartNode());  **if** (path.size() != edgeCount + 1)  **return** **null**;  **return** path;  }  **int** findStartNode() {  **int** start = 0;  **for** (**int** i = 0; i < n; i++) {  **if** (out[i] - in[i] == 1) {  **return** i;  }  **if** (out[i] > 0) {  start = i;  }  }  **return** start;  } | **boolean** graphHasEulerianPath() {  **int** startNodes = 0, endNodes = 0;  **for** (**int** i = 0; i < n; i++) {  **if** (in[i] - out[i] > 1 || out[i] - in[i] > 1) {  **return** **false**;  } **else** **if** (in[i] - out[i] == 1) {  endNodes++;  } **else** **if** (out[i] - in[i] == 1) {  startNodes++;  }  }  **return** (startNodes == 0 && endNodes == 0) ||  (startNodes == 1 && endNodes == 1);  }  **void** dfs(**int** at) {  // while current node has still outgoing edges  **while** (out[at] != 0) {  // select the next unvisited outgoing edge  **int** next = graph.get(at).get(--out[at]);  dfs(next);  }  path.addFirst(at);  } |

**Minimal Spanning Trees: Prims’ Greedy algorithm**

In a given ‘undirected’ graph with weights, MST is a subset of edges which connects all vertices together while minimizing the cost (it should not create cycles otherwise it cannot be a **tree**). Prims works well on dense graphs compared to rivalries Kruskal and Boruvka. The lazy implementation takes O(E\*log(E)) where as eager version takes O(E\*log(V)), however Min Spanning forest on a disconnected graph prim cant do easily as below algo should run on each connected component.

|  |  |
| --- | --- |
| **public** **static** ArrayList<Edge> prims(ArrayList<ArrayList<Edge>> G) {  ArrayList<Edge> mst = **new** ArrayList<>();  PriorityQueue<Edge> pq =  **new** PriorityQueue<>((Object o1, Object o2) -> {  Edge first = (Edge) o1;  Edge second = (Edge) o2;  **if** (first.weight < second.weight)  **return** -1;  **else** **if** (first.weight > second.weight)  **return** 1;  **else**  **return** 0;  });  **for** (Edge e : G.get(0)) {  pq.add(e);  }  **boolean**[] marked = **new** **boolean**[G.size()];  marked[0] = **true**;  **while** (!pq.isEmpty()) {  Edge e = pq.poll();  **if** (marked[e.from] && marked[e.to]) {  **continue**;  }  marked[e.from] = **true**;  **for** (Edge edge : G.get(e.to)) {  **if** (!marked[edge.to]) {  pq.add(edge);  }  }  marked[e.to] = **true**;  mst.add(e);  }  **return** mst;  } |  |

**Graph Bipartite**

|  |  |
| --- | --- |
| **static** **public** **boolean** isBipartite(**int**[][] graph) {  **int** n = graph.length;  **int**[] colors = **new** **int**[n];  **for** (**int** i = 0; i < n; i++) {  **if** (colors[i] == 0 &&  !*dfs*(i, graph, colors, +1)) {  **return** **false**;  }  }  **return** **true**;  } | **private** **static** **boolean** dfs(**int** at, **int**[][] graph, **int**[] colors, **int** color) {  **if** (colors[at] != 0) {  **if** (colors[at] == color) {  **return** **true**;  } **else**  **return** **false**;  } **else** {  // without this you can never go out from inf loop  colors[at] = color;  **for** (**int** eachAdj : graph[at]) {  **if** (!*dfs*(eachAdj, graph, colors, **-1 \* color**)) {  **return** **false**;  }  }  **return** **true**;  }  } |