|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Running Time** | **Code** |
| BFS | O(V+E) |  |
| DFS | O(V+E) |  |
| Topological Sort (DAG) | O(V+E) | **BFS Implementation**  1.Count in-degree of all vertices. And sort them ascendingly.   2.Pick any vertex 'v' which has in-degree of 0.   3.Print 'v'. Remove the vertex 'v' and all edges coming out of it. Decrement in-degrees of all neighbors of vertex 'v' by 1.  4.Repeat steps 2 and 3 till all vertices are removed. |
| **Strongly connected components**  •It is a maximal set of vertices 𝐶⊆𝑉such that for every pair of vertices 𝑢and 𝑣in 𝐶, we have both 𝑢→𝑣and v→𝑢; that is, vertices 𝑢and 𝑣are reachable from each other. | O(V+E) |  |
| **Kruskal’s Min Spanning Trees(Greedy)**  This algorithm treats the graph as a forest and every node it has as an individual tree | 𝑂(𝐸log𝐸)=𝑂(𝐸log𝑉)  We assume that we use  the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and  path-compression heuristics, since it is the asymptotically fastest implementation  known  Moreover, since ˛.jV j/ D O.lgV / D O.lg E/, the total  running time of Kruskal’s algorithm is O.E lg E/. Observing that jEj < jV j2,  we have lg jEj D O.lg V /, and so we can restate the running time of Kruskal’s  algorithm as O.E lgV /. |  |
| **Prim’s Minimum Spanning Trees**  This strategy qualifies as greedy since at each step it adds to the tree an edge  that contributes the minimum amount possible to the tree’s weight.It treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. | Q as a binary min-heap lines 1–5 in O(V ).  the total time for all calls to EXTRACT-MIN is O(V lgV )  The **for** loop in lines 8–11 executes O(E) times  line 11 involves an implicit DECREASE-KEY operation  on the min-heap,O(lg V ) time.  **𝑂(𝑉log𝑉+𝐸log𝑉)=𝑂(𝐸log𝑉)**  –Using Fibonacci heaps:𝑂(𝐸+𝑉log𝑉) bec decrease key takes O(1) | Line 7 identifies a vertex u E Q incident on a light edge that crosses the cut  (V- Q,Q) (with the exception of the first iteration, in which u D r due to line 4).  Removing u from the set Q adds it to the set V- Q ie. Min spanning tree  The **for** loop of lines 8–11 updates the *key* and \_ attributes  of every vertex \_ adjacent to u but not in the tree, thereby maintaining the third part of the loop invariant. |
| Bellman-Ford  (Dynamic Programming) | O(VE) | Detect –ve cycles  D:\bellman2.png  D:\After1stIteration.png  D:\seconditeration2.png |
| DAG Shortest Paths Algorithm | O(V+E) |  |
| Dijkstra’s Algorithm  (Greedy ) |  | D:\Dijkstra Steps\5.jpg |
| All-pairs shortest paths | Dijkstra:  Array🡪O(V^3)  Min-Heap🡪O(VElogV)  Fib-heap🡪 O(V^2logV+VE)  Bellman-Ford🡪O(V^2\*E)  If dense🡪O(V^4) | Run single source for each vertex |
| Slow All Pairs Shortest Path  (adding edges)  Find all vertices reachable in two edges, L(2) , save the matrix, and use it to  find all vertices reachable in three edges, L(3) and so on until we find L(n-1) .  This matrix will contain the shortest path between every pair of vertices in  the graph. | Extend🡪O(n^3)  Slow🡪O(n^4) |  |
| Faster All Pairs Shortest Path  (adding edges) | O(V^3logV) |  |
| Floyd-Warshall Algorithm  (adding Vertices)  Find all vertices reachable using intermediate nodes in the range 1...1 (D(1)), save the matrix, and use it to find all vertices reachable using intermediate vertices in the range 1...2 (D(2)),  and so on until we find D(n). | O(V^3) |  |
| Transitive |  | K🡪 For edges , I 🡪 src vertex , j 🡪 dest vertex |
| Johnson’s Algorithm (re-weighting)  For each edge (u, v), assign the new weight as “original weight + h[u] – h[v]”.  Where h[u] is computed from bellman ford’s | If no negative-weight edges:  –Apply Dijkstra’s algorithm with Fibonacci heaps in 𝑂(V^2 LogV+VE) |  |
|  |  |  |