|  |  |  |
| --- | --- | --- |
| **ALGORITHM FOR TS:**   1. call DFS(G) to compute finishing times f[v] for each vertex v 2. as each vertex is finished, insert it onto the front of a linked list 3. return the linked list of vertices   An edge from a -> c is a cross edge if c has already been discovered.  Node with largest finish time starts first.  NOTE: An MST may not be unique  A tree with n vertices has exactly n-1 edges  (|E| = |V| - 1)  Undirected Graphs: Can’t have self-loops  Weighted: Each edge has an associated weight  Dense: |E|  |V|2. Sparse: |E| << |V| |E| = O(|V|2)  If (u, v)  E, then vertex v and u are adjacent.  Adjacency relationship is:  Symmetric if G is undirected.  Not necessarily so if G is directed.  If G is connected:  There is a path between every pair of vertices.  |E|  |V| – 1.  Furthermore, if |E| = |V| – 1, then G is a tree.  Adjacency List: Pros: Space-efficient, supports many graphs.  Cons: Search entire list to find(u,v), theta(V) worst case.  Ujasoigho | **Greedy Algorithm: Kruskal’s Algorithm (similar to connected component)**  **Prim’s Algorithm (similar to Dijkstra’s Algorithm)**  Being greedy for local optimization with the hope it will lead to a global optimal solution, not always, but in many situations, it works.  **Dynamic:** Store result, bottom-up,  **Memoized:** Cache result, top-down,  **DAGS(Directed Acyclic Graps):**  Can there be a back edge in a DFS on a DAG?  NO! Back edges close a cycle!  A graph G is a DAG <=> there is no back edge classified by DFS(G)  Tree edges  Forward edges  Back edges  Cross edges  Note: there can be multiple topological sorts of G  MST: Some applications:  Communication networks  Circuit design  Layout of highway systems  ***w*(*T*) *=* ∑(*u,v*)∈*T**w*(*u,v*)**  **Free Tree:**  A tree with n vertices has exactly n-1 edges  (|E| = |V| - 1)  **We say that a subset *A*⊆ *E* is *viable* if *A* is a subset of edges in some MST. We say that an edge (*u,v*) ∈ *E*-*A* is *safe* if *A*∪{(*u,v*)} is viable** | Qs:  If begin is less than end  Get pivot  Qs on left  Qs on right  Select:  Random r = new Random  while(right >= left)  int pivotIndex = partition(copyArr, left, right, r.nextInt(right - left + 1) + left);  if(pivotIndex == k)  return copyArr[pivotIndex]  else if(pivotIndex < k)  left = pivotIndex + 1;  else  right = pivotIndex - 1;  return 0;  int pivotValue = a[pivotIndex];  Partition: swap(a, pivotIndex, right); int storeIndex = left;  for(int i = left; i < right; i++)  if(a[i] < pivotValue)   swap(a, i, storeIndex);  storeIndex++; swap(a, right, storeIndex)   return storeIndex; |