Agenda:

- DFS application:
  - Deciding DAG (Directed acyclic graphs)
  - Topological sorting
  - Strongly Connected components
- MST (Minimum Spanning Tree) (Kruskal's algorithm)

Reading:

- CLRS: 612-623
- CLRS Chapter 23.2

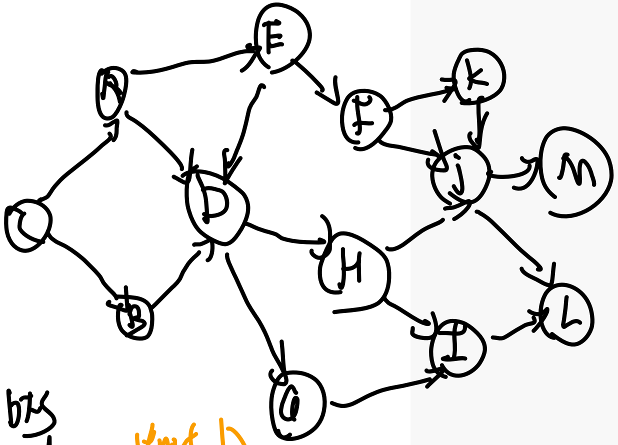
DFS Application 1: Directed Acyclic Graph (DAG)

- Thm 1. DFS has a back edge iff G contains a cycle.
  - Proof: (⇒) the back-edge (u, v) along with the tree edges connecting v to u is a cycle in G.
    (⇐) If there's a cycle let v_1 be the first node on the cycle that turns gray. So the cycle is $(v_1, v_2, .., v_k, v_1)$. At time $v_1.dtime$ the $v_1 \to v_k$ path is all white, so v_k is a descendant of v_1. Thus when the edge (v_k, v_1) is traversed, both vertices are gray, so it is a back-edge.
- Corollary: G is a DAG iff the DFS has no back-edges.
- An algorithm to determine if G is a DAG:
  Run DFS; if DFS encounters a gray-gray edge, abort and output "found a cycle"; upon DFS conclusion output "DAG".

Topological ordering in DAG's

- Suppose we have a set of tasks to be performed
- For each task we have a requirement that some of the other tasks must be done before we can perform this.
- This requirement is given as a directed graph G which is DAG (directed acyclic).
- If $(u, v) \in E$ it means we must perform u before we can perform v.
- Goal: find an ordering of the tasks (vertices of G) such that for each task all its requirements appear earlier in that ordering,

- i.e. find an ordering $v_1, \ldots, v_n$ of vertices of G such that for every edge (v_i, v_j), $i < j$. This is called a "topological soring"
- Theorem: A digraph has a topological soring if and only if it is acyclic.
- Clearly if we have a cycle we cannot have a topological ordering (why?)
- Now suppose that G is a DAG.
- We prove the theorem by induction on n. Base case $n = 1$ is trivial (any ordering will do).
- So assume that $n \geq 2$. There is a vertex in G which has no ingoing edges or else G has a cycle (why?)
- Say in-degree$(u) = 0$. Remove v from G, call the new graph G' (which has $n - 1$ vertices).
- G' is acyclic so by I.H. has a topological ordering $v_2, \ldots, v_n$.
- Since u has only outgoing edges, $u, v_2, \ldots, v_n$ is a topological ordering of G.
- The above suggests the following algorithm:



DFS order: start D

H, j, M, L, I,

topological order

H, I, j, L, M

- procedure Topological-Sort(G)

```
procedure Topological-Sort($G$)
    $S \leftarrow \emptyset$
    for each $v \in V$ do
        if in-degree$(v) = 0$ then
            S.enqueue$(v)$
    $i \leftarrow 1$
    While $S \neq \emptyset$ do
        $v \leftarrow$ S.dequeue()
        output $v$
        $i \leftarrow i + 1$
        for each $vu$ do
            Remove $vu$ (so decrease in-degree$(u)$)
            if in-degree$(u) = 0$ then
                S.enqueue$(u)$
    if $i < n$ then
        return ‘‘$G$ has a cycle’’
```

- We can maintain (for each node) the value of in-degree(v); all of these can be computed in $\Theta(n + m)$ by going through adjacency list.
- We have n iterations of the while loop; each time we remove a vertex v with out-degree $d_{out}(v)$ we have to update in-degree of all its neighbors (and if any of them becomes zero we insert that node into the queue); this update takes $d_{out}(v)$

- We can also use DFS to find a topological ordering (as in the textbook).
  - G is a DAG $\Rightarrow$ no back-edges
  - No gray-gray edges.
  - (u, v) is a gray-white edge:

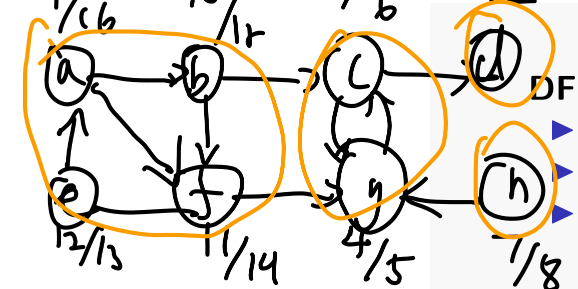
$$u.dtime < v.dtime < v.ftime < u.ftime$$

  - (u, v) is a gray-black edge:

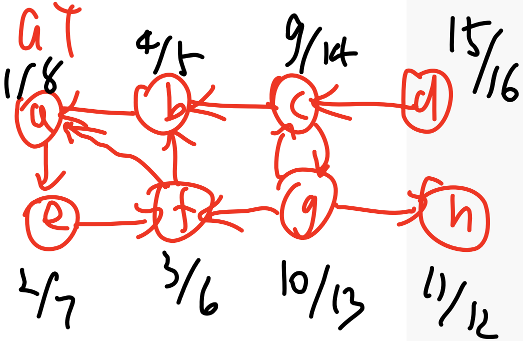
$$v.dtime < v.ftime < u.dtime < u.ftime$$

  - $dtime$ isn't consistent, but $ftime$ is:
    we must have $v.ftime < u.ftime$ for any edge (u, v)
- Sort the vertices by descending order of $ftime$ and you got a topological sort.
- Doesn't have to take extra $O(n \log(n))$. Can be done as part of the DFS algorithm
  - When a node turns black, insert it to the end of a $TopSort$ array
  - Or `Push()` it into a $TopSort$ stack
- After DFS, print the array / `Pop()` and print elements in the stack.
- Conclusion: A $O(n + m)$-time algorithm for topologically-sort a DAG or output a cycle.

9/ 10/ 3/ 1/2



~~d~~, ~~c~~, ~~g~~, ~~h~~, ~~a~~, ~~e~~, ~~f~~, ~~b~~



DFS Application 2: Finding Strongly-Connected Components

- A directed graph every edge is directed (i.e. it is an ordered pair)
- We say u reaches v if there is a directed path from u to v
- Strongly connected digraph: A digraph G is strongly connected if for every pair u, v of vertices u is reachable from v and v is reachable from u
- Recall: In a digraph G, $SCC(u)$ is the set of all nodes v that are reachable from u and that u is reachable from them.
- Recall: $v \in SCC(u)$ iff $u \in SCC(v)$
- Recall: the SCCs of G form a partition of V into $\{C_1, C_2, ..., C_k\}$.
- Moreover, draw graph G_{SCC} on k nodes: $v_1, ..., v_k$ (so that v_i represents C_i). Put en edge (v_i, v_j) iff for some $x \in C_i, y \in C_j$ such that (x, y) is an edge in G.

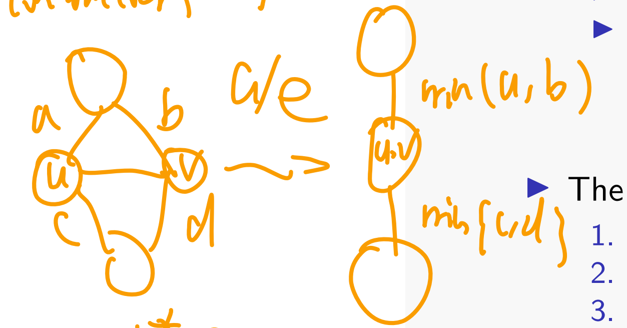
  Then G_{SCC} is a DAG.
- Moreover, C is a SCC in G iff it is a SCC in the flipped graph G^T. ((u, v) is an edge in G iff (v, u) is an edge in G^T)
- To find the SCCs of G
  1. Run DFS on G.
  2. Flip G's edges to create G^T
  3. Run DFS on G^T **but** the main DFS loop traverses nodes in a decreasing $ftime$ order
  4. SCCs of G are the trees of the DFS-forest of G^T
- Runtime $O(n + m)$.

Minimum Spanning Tree (MST) problem:

- Input: edge-weighted undirected graph
- Notions:
  - subgraph: $G' = (V, E')$, where $E' \subset E$, forest = acyclic graph, trees
  - spanning subgraph: subgraph including all the vertices
  - spanning tree: spanning subgraph which is a tree — acyclic connected must have exactly $n - 1$ edges
    e.g., BFS/DFS-tree is a spanning tree of the graph
  - minimum spanning tree: sum of weights on tree edges is minimal
- **The MST Problem: Find an MST for the input graph.**
  - ... there could be more than one ...
  - Example: all weights are the same, both BFS/DFS produce a MST ...
- The Minimum Spanning Forest problem: If the given graph is not necessarily connected: find MST for each CC.

remove edges

contraction



say MST* ∈ e

Greedy algorithms and MST problem:

- Greedy algorithms:
  - Greedy — each step makes the best choice and don't look back
  - Optimal substructure: an optimal solution to the original problem contains within it optimal solutions to subproblems. T is MST for $G = (V, E) \Rightarrow$ for any $U \subset V$ where $T[U]$ is connected, $T[U]$ is an MST.
- The general MST algorithm outline:
  1. A is a set of "safe" edges: they are contained in some MST T
  2. $A = \emptyset$ initially
  3. `while` $(|A| < n - 1)$ `do:` find a safe edge $e = (u, v)$ and set $A = A \cup \{e\}$.
- Two greedy solutions
  - Prim's Algorithm (Actually: Prim + Dijkstra + Boruvka)
    Grow T vertex-wise: A is always a MST on some $S \subset V$
  - Kruskal's (Actually: Kruskal + Boruvka, which we discuss first).
    Grow T edge-wise: A is always a minimal set of edges without a cycle (forest)

Say that $T^* \in e$

$T^* - e$ is spanning tree of G/e

$w(T') \le w(T^* - e)$

$w(T' \cup \{e\}) = w(T) + w(e)$

$\le w(T^* - e) + w(e) = w(T^*)$

Kruskal's algorithm for the MST problem:

- Input: an edge-weighted (simple, undirected, connected) graph (positive weights)
- Output: an MST
- Idea:
    - Start with a forest T on all the vertices and no edges
    - Grow the forest T to become a tree by adding one edge at a time
    - The edges are considered in non-decreasing order of their weight
    - an edge can be added if it joins two different connected components (i.e. two trees of T)
    - So an edge is added if it does not create a cycle, otherwise it is discarded
    - For each vertex we keep an index which tells the index of the "cluster" to which it belongs.
    - When we add an edge, we merge the clusters (i.e. the sub-trees) that it connects.

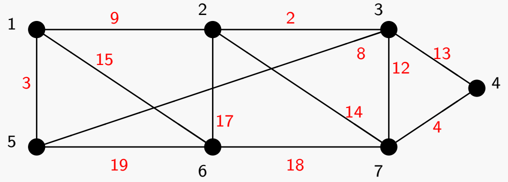
Kruskal's algorithm for the MST problem:

- procedure kruskal (G)

```
$T \leftarrow \emptyset$
for each $v \in V(G)$ do
   Define cluster $C(v) \leftarrow v$
sort edges in $E(G)$ into non-decreasing weight order
for each edge $e_i = (u, v) \in E(G)$ do
   if $C(u) \neq C(v)$ then
   $T \leftarrow T \cup \{e_i\}$
   merge clusters $C(u)$ and $C(v)$
return $T$
```

Kruskal's algorithm for the MST problem — An Example:

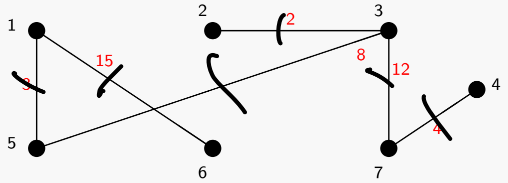
- An example:



- Sorting the edges:

2	3	4	8	9	12	13	14	15	17	18	19
(2,3)	(1,5)	(4,7)	(3,5)	(1,2)	(7,3)	(3,4)	(2,7)	(1,6)	(6,2)	(6,7)	(5,6)

- kruskalMST(G) returns:



Kruskal's algorithm for the MST problem — analysis:
Correctness:

- We prove that after every iteration of the while loop, where we have examined i edges and have a partial solution built into T, call it T_i, there is a MST T_{opt} which has all these edges of T_i ane every edge that is in T_{opt} and not in T_i are among the edges we have not examined yet, i.e. $T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, \ldots, e_m\}$
- This implies that every decision we make in every step is consistent with an optimum solution.
- This is proved by induction on i; the base case is trivial.
- Once we prove it for the very last iteration (say after considering e_m) it implies that the solution is a MST.
- So consider the induction step where we have T_i satisfying the predicate stated above and we examine edge e_{i+1}
- If we don't add e_{i+1} (because it creates a cycle) then $T_{i+1} = T_i$; we claim that in this case T_{opt} cannot have e_{i+1} either because all the edges of T_i are in T_{opt} as well and if they create a cycle with e_{i+1}, then e_{i+1} cannot belong to T_{opt} either; Thus $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.
- If we add e_{i+1} to T_i then there are two cases; the easy case is if T_{opt} also contains it (so we are consistent with the optimum solution after step $i + 1$ too).

- The critical case is when we select edge e_{i+1} to be added to T_i to obtain T_{i+1} but T_{opt} does not have it.
- In this case, $T' = T_{opt} + e_{i+1}$ has a cycle, C.
- This cycle contains at least one edge e_j that is not in T_i (why?)
- That edge must be among the edges in $\{e_{i+1}, \ldots, e_m\}$ because $T_{opt} \subseteq T_i \cup \{e_{i+1}, \ldots, e_m\}$ (so if an edge is in T_{opt} but not in T_i must be in the second set).
- this implies edge e_j is among the edges we have not considered yet, because up until edge e_i, all the decisions made were consistent with T_{opt}; i.e. $j \geq i+2$.
- So $w(e_j) \geq w(e_{i+1})$. So $T' = T_{opt} + e_{i+1} - e_j$ is also a MST that extends T_i.
- Running time analysis: how to implement "Merge clusters $C(u)$ and $C(v)$"?

Running time analysis:

- Each cluster will be an unordered linked list of vertices in that cluster
- Each vertex v also keeps the index of the cluster to which it belongs
- To find $C(v)$ it takes $O(1)$ time only (check the index)
- To merge $C(v)$ and $C(u)$: merge the smaller list into the larger one and update the index of the vertices whose list is merged.
- Thus, merging $C(v)$ and $C(u)$ takes $O(\min\{|C(u)|, |C(v)|\})$ time.

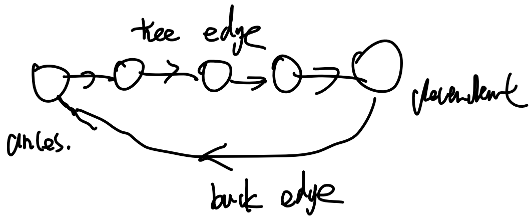
- Each time we update the reference of a vertex we can put 1 token on that node (to account for the time spent to update that node)
- So at the end of the algorithm, the total number of tokens over all the nodes is equal to the number of times the references of the nodes were updated over all calls to "Merge" operation in the while loop
- So to bound the time complexity of all "Merge" operations over all the loop iterations it is sufficient to count the total number of tokens over all the nodes.
- Observation: each time we update the reference for a vertex the size of the cluster to which it belongs at least doubles; starts from 1 and goes up to n
- Thus: number of times we update a vertex's reference is at most $\log n$.
- Therefore, the number of tokens on each node is at most $\log n$.
- Total time for all merges and cluster updates: $O(n \log n)$.
- Time for sorting edges: $O(m \log m) = O(m \log n)$, time for the while loop $O(m) + O(n \log n)$.
- Total time for Kruskal's algorithm $O((m + n) \log n)$

cycle detection

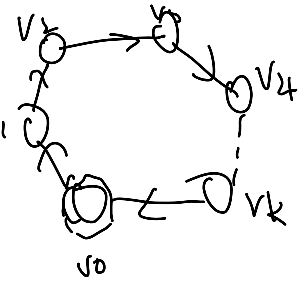
G has a cycle <~> DFS has a back edge

proof:

left:



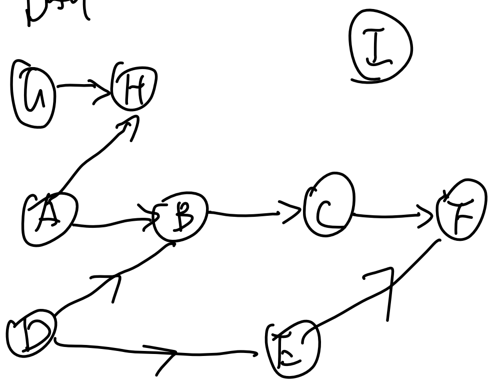
right



assume V0 is first vertex in the cycle visited by DFS

claim (Vk, V0) is back edge

DAG



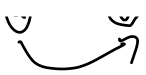
Topological sort: run DFS

output reverse of finishing times of vertices

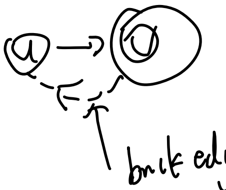
correctness: for any edge e=(u,v), v finishes before u finishes

case 1: u starts before v

(u)→(v) => visit v before u finishes



case 2: v starts before u



back edge → contradiction