CS412 - Graph Algorithms

Date

Let $G = (V, E)$ be an undirected graph, with $n = |V|$ and $m = |E|$.

Path: in an undirected graph is a sequence $v_1, v_2, \ldots, v_k$ of nodes s.t. $\{v_i, v_{i+1}\} \in E$, $i = 1, 2, \ldots, k-1$

A path is a cycle if $k > 2$ and first $(k-1)$ nodes are disjoint.

Directed Graph: Let $DG = (V, E)$ be a directed graph with E, a set of directed edges (or arcs), where each edge is an ordered pair (u, v) of nodes, $u, v \in V$.

Connectivity: A graph is called connected if all nodes belong to the same component.

Directed Graph
- Weakly Connected Components
- Strongly Connected Components (SCC)

A directed graph is strongly connected if for every two nodes u and v, $u \rightsquigarrow v \wedge v \rightsquigarrow u$
[a path exists from u to v and vice versa].

Graph Representation: We assume an adjacency list representation unless stated otherwise.

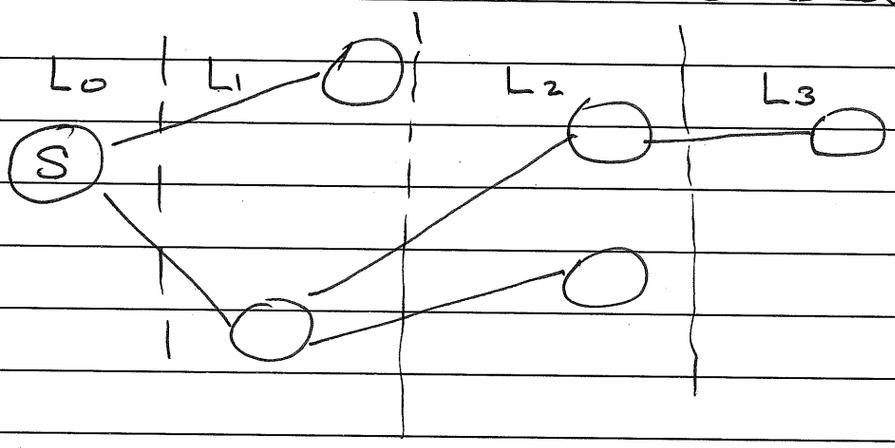
Graph traversal:
- BFS
- DFS

Runtime is $O(n+m)$ or $O(V+E)$, ie linear time in graphs.

RC

Let S be a node of an undirected graph G = (V,E)
BFS (s) finds all nodes t, t∈V/s that are reachable from node S.

Output: BFS constructs a tree from G, where all reachable nodes are divided into layers.



Layer 0 contains only the source node 'S'. Then a layer Li+1 consists of all nodes v s.t., they do not belong to layers L0, L1, ..., Li and that the distance [shortest] from S to v is i.

Analysis:

Given an undirected graph G = (V,E), let us use lists L[0], L[1], ..., corresponding to each layer L0, L1, ..., respectively.

Let Discovered [n] is an array of length n s.t.
Discovered [v] = True iff v is in the BFS tree 'T'

∵ Each node belongs to exactly one list
* Time to initialize the list and manage Discovered: O(n).

RC

Let $(u,v) \in E$. BFS will work with the edge twice:-

a) When BFS visits node u.
b) When BFS visits node v. (Sees)

∴ There are 'm' edges in G, the time will be $O(m)$.

Hence, the total running time of BFS will be $O(m+n)$ or $O(n+m)$.

BFS works exactly in case of a directed graph.

× ——— ×

Finding Connected components in an Undirected Graph $G=(V,E)$.

1. Start with an arbitrary node 'S' and run BFS(S) to construct a tree 'T₁', if $|T_1| = n$, then the graph is connected [# of components = 1]

2) Otherwise, pick a node $u \notin T_1$ and run BFS(u) to construct T_2. If $|T_1| + |T_2| = n$, stop. Repeat untill all nodes in G are visited,

RC

Date

Finding Strongly Connected Components (SCC) in a directed graph $DG = (V, E)$.

Given two nodes $u, v \in V$ in a directed graph $DG = (V, E)$, define an equivalence relation 'SCC':

> Recall, an equivalence relation R on a set S is:
> (a) Reflexive aRa (b) Symmetric: $aRb \wedge bRa$
> (c) Transitive: $aRb \wedge bRa \Rightarrow aRc$

SCC is an equivalence relation

i) Reflexive: $S \rightsquigarrow S$, $(S \in V)$

ii) Symmetric: $S \rightsquigarrow t \wedge t \rightsquigarrow S$ $(S, t \in V)$

iii) Transitive: if $S \rightsquigarrow t \wedge t \rightsquigarrow u \Rightarrow S \rightsquigarrow u$, $(S, t, u \in V)$

An equivalence relation partitions S into disjoint sets called equivalence class.

In case of a directed graph $DG = (V, E)$, the node set V is partitioned into disjoint sets (equivalence classes) or SCC by this relation

ie,

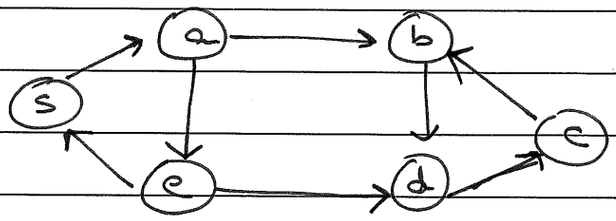
Given two nodes $u, v \in V$, either u, v are in the same SCC or in different SCCs.

Finding SCCs: Both BFS and DFS-based solutions exist

Using BFS to find SCCs:

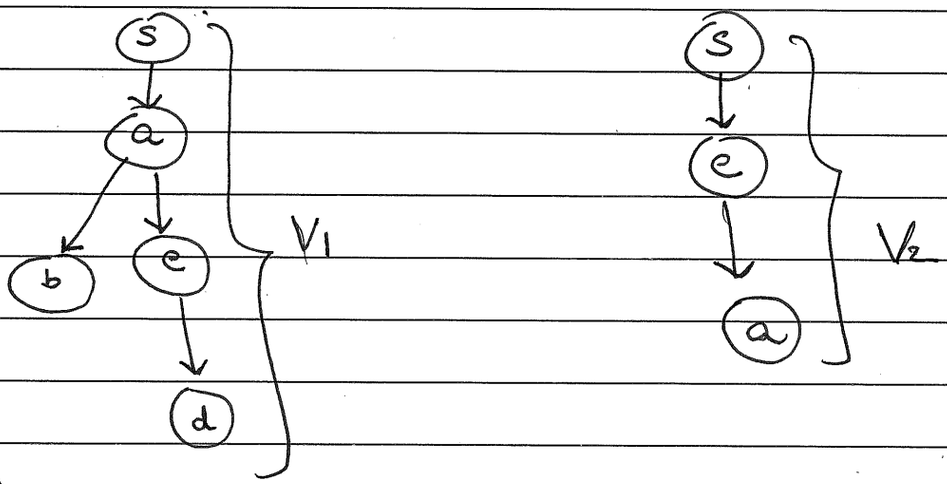
Given a directed graph DG.

DG =

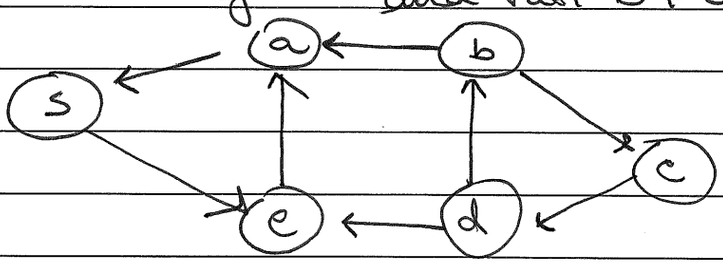


Pick a node 's'. Run BFS(s):

Now, reverse the graph. What's the cost to reverse edges of DG?



DGrev: Reversing the edges. What is the cost?
Pick 'S' again and run DFS on DGreversed

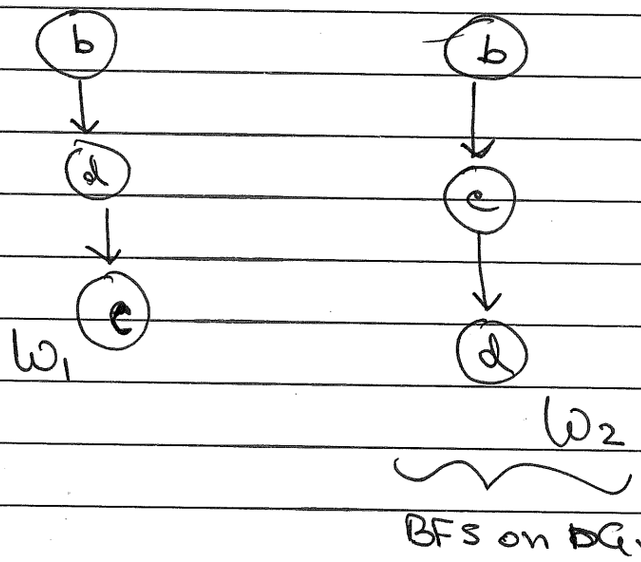


$V_1 \cap V_2$ i.e. {s, e, a} belong to one strongly-connected component

Date

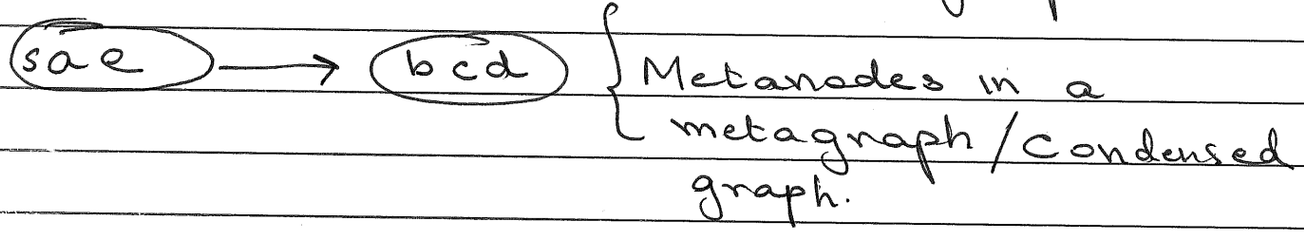
Likewise,

run BFS(b) on DG and DGrev:

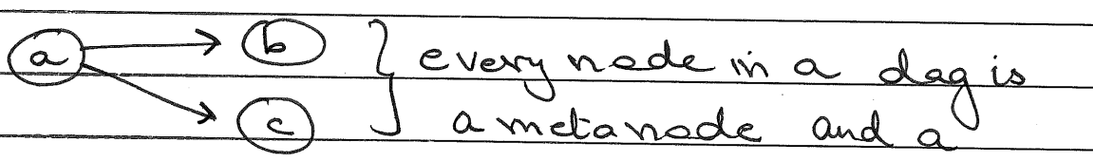


$W_1 = W_2$, therefore, {b, c, d} belong to one scc.

We can draw the actual directed graph as:



A dag (directed acyclic graph) is a directed graph (meta) of strongly-connected components of a directed graph, e.g.,



Date

i.e., Given a dag say $G = (V, E)$, every node is a SCC.

* There is at least one node that is called a Sink

* There is at least one node that is called a Source.

Q: How many SCCs exist in a dag of size 'n'?

× ——————— ×

Source: CLRS

Depth-First Traversal (DFS)

DFS explores edges out of the most recently discovered node v, s.t. it still has unexplored edges leaving it. Once all the edges have been explored, the search backtracks to explore the edges leaving the nodes from which v was discovered (i.e., v's predecessors)

* As soon as a node $v \in V$ is discovered, it is assigned a parent or a predecessor.

The Predecessor Subgraph of DFS

Defn:

$G_\pi = (V, E_\pi)$, where $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{Nil}\}$

is a DFS forest.

1. Initialization of DFS: Each node is initially white $[O(n)]$.

2. Each node is grayed when it is first discovered and colored as black (eg), when finished, i.e, the adjacency list is examined completely.

$\Rightarrow$ Each node $v \in V$ in $G = (V, E)$ ends up in exactly one DFS tree so all trees in the DFS forest are disjoint.

A DFS on a graph $G = (V, E)$
- i) creates a DFS Forest (disjoint trees)
- ii) Time-stamping of nodes.

Each node $v \in V$ has two time-stamps:
- i) v.d : records when v is first discovered. (grayed)
- ii) v.f : when v is finished (black)

↓

i.e the search finishes examining the adjacent list of node v

Timestamping in DFS helps solve many problems including finding SCCs

RC

[A time stamp is an integer between 1 and 2|V|]

For every node $v \in V$, there is a discovery time $(v.d)$ and a finishing time $(v.f)$ with $v.d < v.f$.

DFS pseudocode

```
DFS (G): 1. For each u ∈ G.V {
         2      u.color ← white        } Initialization
         3      u.pred ← Nil           } O(n).
         4.. t ← 0    // set the clock O(1)
         5. For each u ∈ G.V {
         6.     if u.color = white      } O(m)
         7.         DFS_Visit (G, u)    } # of edges
```

```
DFS-visit (G, u):
    1. time ← time + 1    //discovery time
    2. u.d ← time.
    3. u.color ← gray
    4    For v ∈ Adj [u]  { fetch from u's adjacency list

    5.       if v.color = white
    6.           v.pred ← u
    7            DFS-visit (G, v)  { Recursive call

    8.       u.color ← black
    9.       time ← time + 1
    10.      u.f ← time.   { set finishing time
```

Date

Since DFS-Visit (G, u) is called exactly once for each node $u \in V$, and

Since

$$\sum_{v \in V} Adj[v] = \underbrace{O(|E|)}_{O(m)}$$

DFS runs in $O(n+m)$ time.

Using DFS to find SCC

Given a [directed] graph $DG = (V, E)$, for any nodes $u \neq v \in V$, then two intervals $[u.d, u.f]$ and $[v.d, v.f]$ are either contained in one another or disjoint

Why? Since $[u.d, u.f]$ is the time u stays on the stack [recall DSA], the nice property from CS102 (DSA) is checking parentheses' balance using stacks

The Parenthesis Theorem

For any two vertices $u \neq v \in V$, exactly one of the three conditions hold true.

a) $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, i.e., neither u or v are descendents of each other in the DFS forest.

b) $[u.d, u.f]$ is contained in $[v.d, v.f]$, i.e., u is a descendent of v in a DFS tree.

c) The other way round RC of (b).

Date

Finding SCCs using DFS

The scheme runs in $O(n+m)$.

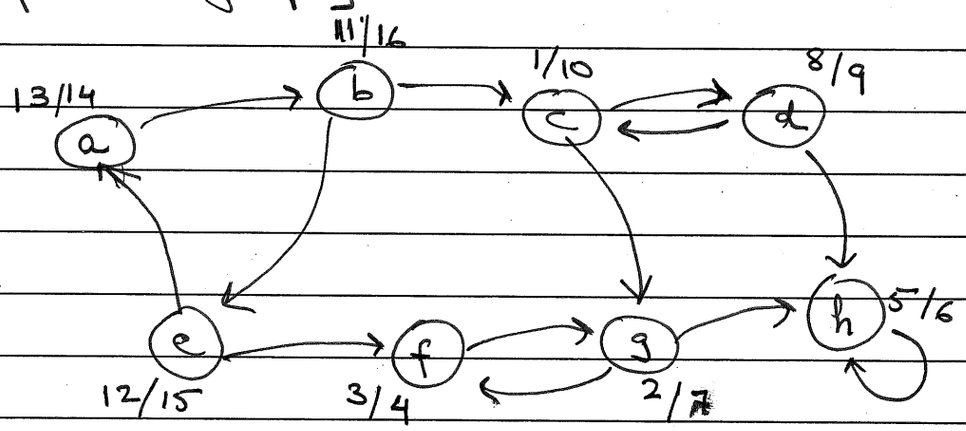
1. Run two DFS on G and G^T:

1) Call DFS(G) to compute finishing times $u.f, \forall u \in V$.
2) Compute G^T (reversed graph)
3) Call DFS(G^T) but in the main loop of DFS(G^T) consider vertices in decreasing order of u.
4) Output nodes/vertices of each DFS tree in the DFS forest as a separate SCC.

[The idea behind this comes from a property of a component graph]

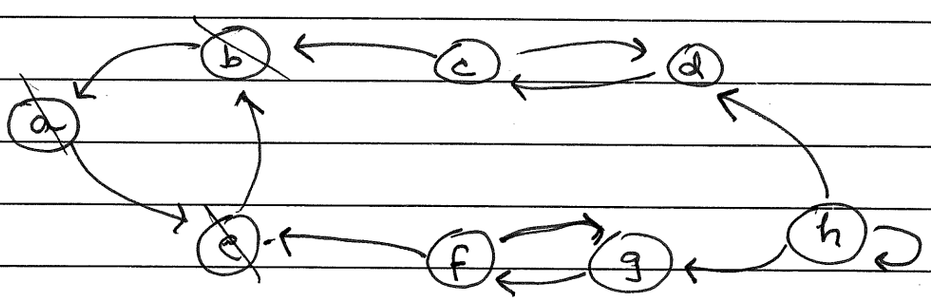
e.g.:

$G =$



Let's run DFS on G starting from node 'c'.

Now, let's run DFS on G^T (Grev), choosing nodes to start with in decreasing finishing order with node 'b'.



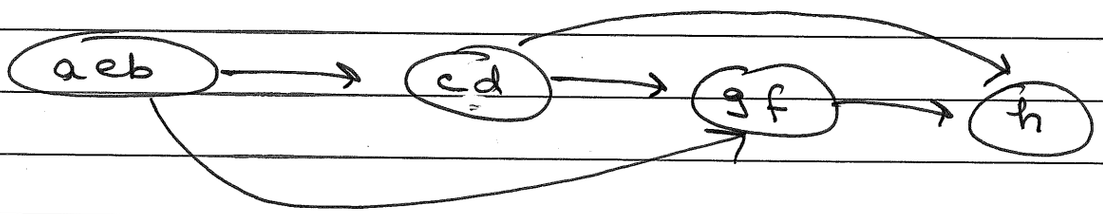
$F = \{\{b, a, e\}, \{c, d\}, \{g, f\}, \{h\}\}$

T_1 T_2 T_3 T_4

RC

There are four SCC in the directed graph G

Condensing them into meta nodes will give us a dag!



We've also linearized the graph (see next!)

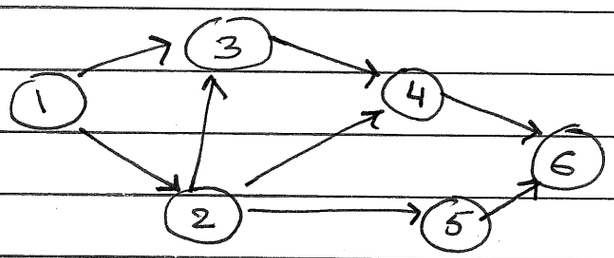
x ——— x

DAGs and Topological Ordering:

Let G = (V, E) be a dag. A topological ordering of G is an ordering of the nodes as

$v_1, v_2, \ldots, v_n$ s.t. for every edge (v_i, v_j) we have $i < j$

Example: Consider the following dag:



∵ G has no cycles, there exists a _____ directed path in G of maximum length.

Let some node v be the first node on this path. Then v has no incoming edge (a source node)

Date

and will be the first node in the topological order.

Remove V_1. The remaining graph would still be a dag.

Repeat the same for the next source node and so on.