Introduction to Algorithms
Lecture 8 Graph and DFS

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

Course Logistic

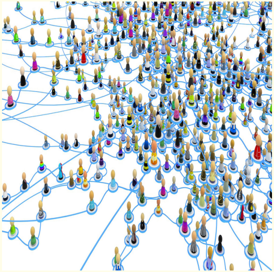
1. Midterm: 13:30 — 15:30 on April 17, no make-up exam
2. Office Hour: 3A103 (this week and next week)

Outline

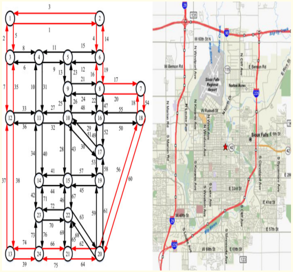
1. Introduction
2. Representations of Graphs
3. DFS on undirected graphs
4. DFS on directed graphs
5. Topological Order

Overview

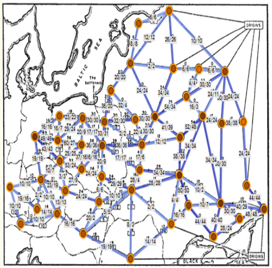
Graphs are fundamental objects in many areas:



(a) Social Network



(b) Traffic Network

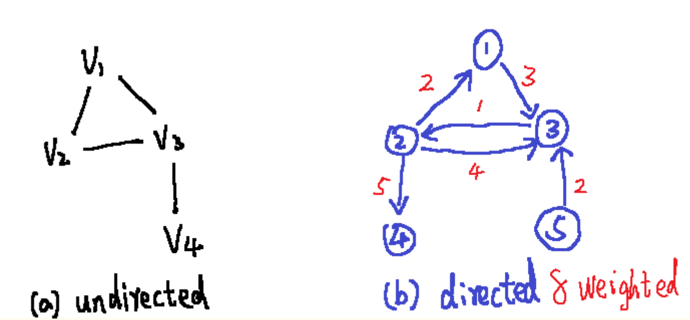


(c) Network Flow

Overview (II)

Basic notation

1. Graph G is specified by its vertex set V and edge set E
2. n vertices (a.k.a. nodes, points, terminals) in V: $\{v_1, \ldots, v_n\}$
3. m edges in E: could be undirected or directed, unweighted or weighted, ...



Its flexibility captures abstract models of many practical problems

Plan

Many interesting problems

1. Connectivity: directed vs undirected, connected components, minimal spanning trees,
2. Shortest paths: distance, negative costs, single-source vs all pairs, ...
3. Max flow, min cut, matching, ...
4. (optional) max cut, random walk, page-rank, effective resistance, ...

Our focus in the next 3 weeks!

Outline

1 Introduction

2 Representations of Graphs

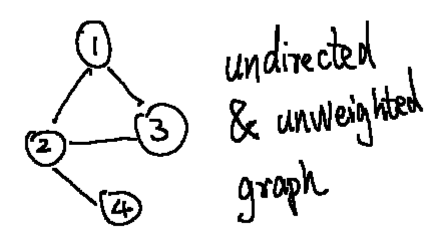
3 DFS on undirected graphs

4 DFS on directed graphs

5 Topological Order

Introduction

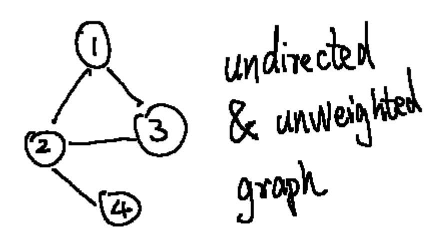
Two standard ways to store/access a graph: consider unweighted undirected graph for convenience



1. Adjacency matrix $A \in \mathbb{R}^{n \times n}$: $A[i,j] = 1$ if $(i,j) \in E$; o.w. $A[i,j] = 0$.

Introduction

Two standard ways to store/access a graph: consider unweighted undirected graph for convenience

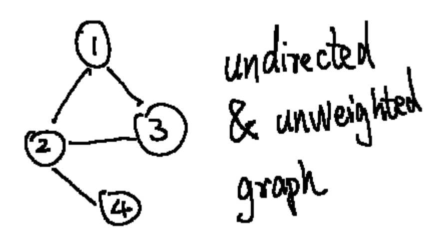


1. Adjacency matrix $A \in \mathbb{R}^{n \times n}$: $A[i,j] = 1$ if $(i,j) \in E$; o.w. $A[i,j] = 0$.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Introduction

Two standard ways to store/access a graph: consider unweighted undirected graph for convenience



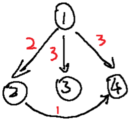
1. Adjacency matrix $A \in \mathbb{R}^{n \times n}$: $A[i,j] = 1$ if $(i,j) \in E$; o.w. $A[i,j] = 0$.
2. For each vertex v, store all its edges in an adjacency-list $Adj[v]$:

$$Adj[1] = \{2,3\}, Adj[2] = \{3,1,4\}, Adj[3] = \{1,2\}, Adj[4] = \{2\}.$$

Adjacency Matrix

General Graphs

1. For directed graph, $A[i,j] = 1$ only if $(i,j) \in E \Rightarrow A[i,j] \neq A[j,i]$
2. For weighted graph, $A[i,j] = w_e$ for edge $e = (i,j)$



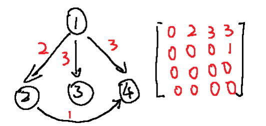
$$\begin{bmatrix} 0 & 2 & 3 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3. Many variants: $A^\top$, normalized $A[i,j] = \frac{A[i,j]}{deg(i)}$, symmetrical normalized $A[i,j] = \frac{A[i,j]}{\sqrt{deg(i) \cdot deg(j)}}, \ldots$

Adjacency Matrix

General Graphs

1. For directed graph, $A[i,j] = 1$ only if $(i,j) \in E \Rightarrow A[i,j] \neq A[j,i]$
2. For weighted graph, $A[i,j] = w_e$ for edge $e = (i,j)$



3. Many variants: $A^\top$, normalized $A[i,j] = \frac{A[i,j]}{deg(i)}$, symmetrical normalized $A[i,j] = \frac{A[i,j]}{\sqrt{deg(i) \cdot deg(j)}}, \ldots$

Pro: (1) Easy to access and maintain; (2) Elegant math expression; (3) Rich tools from matrix theory;

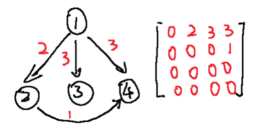
Example

What does $A^{100}[i,j]$ stand for?

Adjacency Matrix

General Graphs

1. For directed graph, $A[i,j] = 1$ only if $(i,j) \in E \Rightarrow A[i,j] \neq A[j,i]$
2. For weighted graph, $A[i,j] = w_e$ for edge $e = (i,j)$



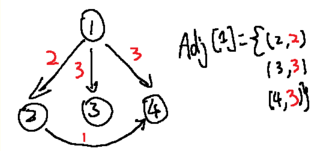
3. Many variants: $A^\top$, normalized $A[i,j] = \frac{A[i,j]}{deg(i)}$, symmetrical normalized $A[i,j] = \frac{A[i,j]}{\sqrt{deg(i) \cdot deg(j)}}, \ldots$

Pro: (1) Easy to access and maintain; (2) Elegant math expression; (3) Rich tools from matrix theory;
Con: n^2 space is wasteful when $|E|$ is small

Adjacency List

General Graphs

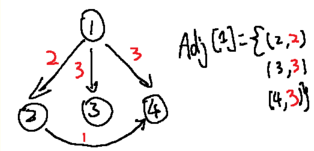
1. For directed graph, $adj[i]$ stores the list of j such that $(i, j) \in E$.
2. Maintain another list for incoming edges
3. Record weights



Adjacency List

General Graphs

1. For directed graph, $adj[i]$ stores the list of j such that $(i, j) \in E$.
2. Maintain another list for incoming edges
3. Record weights



Pro: (1) Save space; (2) Improve running time, $O(n^2) \to O(m)$, for sparse graphs
Con: (1) Hard to maintain and access — how to determine $(i, j) \in E$?
(2) No clean math notation

Summary

Usually we use adjacency matrix for dense graphs and adjacency list for sparse graphs.

Summary

Usually we use adjacency matrix for dense graphs and adjacency list for sparse graphs.

Pick the correct representation based on applications!

Outline

1 Introduction

2 Representations of Graphs

3 DFS on undirected graphs

4 DFS on directed graphs

5 Topological Order

Exploring Graphs

Given an undirected graph $G = (V, E)$, explore all reachable nodes?

Exploring Graphs

Given an undirected graph $G = (V, E)$, explore all reachable nodes?

Basic idea:

1. Keep track of all nodes discovered;
2. While there is an unexplored path, follow it

Algorithm Description

Keep track of (1) discovered vertices; (2) which edge to follow.

---

procedure DFS-EXPLORE(u)
// $visited(v) = False$ for all v in the initial stage
    $visited(u) = True$
    for each $v \in Adj[u]$ **do**
        if $visited(v) = False$ **then**
            $v.\pi = u$ // Record v's parent node
            DFS-EXPLORE(v)

Analysis: Correctness

Claim

When all vertices have $visited(v) = False$, DFS-EXPLORE(u) will mark all vertices reachable from u to *True*.

Analysis: Correctness

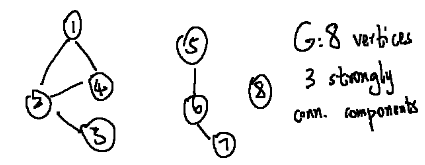
Claim

When all vertices have $\textit{visited}(v) = \textit{False}$, DFS-EXPLORE$(u)$ will mark all vertices reachable from u to *True*.

Proof Sketch: For contradiction, suppose v is not marked and $\exists$ a path from u to v. Consider the previous node w in front of v on this path ...

Connected Components

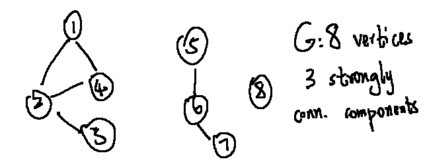
procedure DFS(G)
  for each $v \in V$ **do**
    $visited(v) = False$
    $v.\pi = NIL$
  for each $v \in V$ **do**
    if $visited(v) = False$ **then**
      DFS-EXPLORE(v)



Connected Components

---

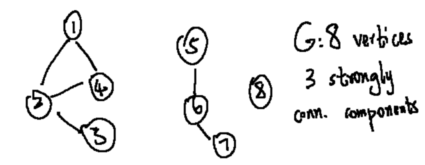
procedure DFS(G)
  for each $v \in V$ **do**
    $visited(v) = False$
    $v.\pi = NIL$
  for each $v \in V$ **do**
    if $visited(v) = False$ **then**
      DFS-EXPLORE(v)



1. To explore all vertices, restart DFS at any vertex that has not yet been visited.

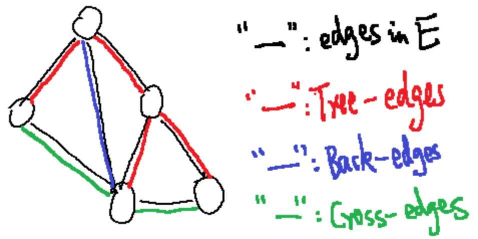
Connected Components

procedure DFS(G)
  for each $v \in V$ **do**
    $visited(v) = False$
    $v.\pi = NIL$
  for each $v \in V$ **do**
    if $visited(v) = False$ **then**
      DFS-EXPLORE(v)



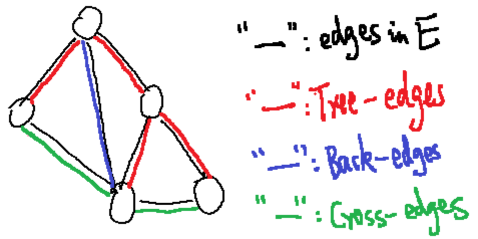
1. To explore all vertices, restart DFS at any vertex that has not yet been visited.
2. Strongly Connected Components: a *maximal* set of vertices $C \subseteq V$ s.t. for every $u, v \in C$, $u \to v$ and $v \to u$.
3. Each DFS-EXPLORE call finds a strongly connected component

Types of Edges



Define 3 types of edges according to a given tree

Types of Edges



Define 3 types of edges according to a given tree

Theorem 22.10 in CLRS for Undirected Graphs

Consider the DFS-tree based on $v.\pi$, all edges in E are either tree-edges or back-edges — there is no cross-edges

Running Time

Theorem

The running time is $O(n + m)$.

Running Time

Theorem

The running time is $O(n+m)$.

DFS-EXPLORE(u) is applied at most once for each u and $\sum_u deg(u) = 2m$.

```
procedure DFS-EXPLORE(u)
    visited(u) = True
    for each v ∈ Adj[u] do                    // Time: O(deg(u))
        if visited(v) = False then
            v.π = u
            DFS-EXPLORE(v)
procedure DFS(G)
    for each v ∈ V do
        visited(v) = False
        v.π = NIL
    for each v ∈ V do
        if visited(v) = False then
            DFS-EXPLORE(v)
```

Outline

1 Introduction

2 Representations of Graphs

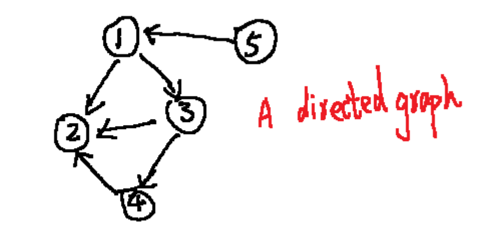
3 DFS on undirected graphs

4 DFS on directed graphs

5 Topological Order

Introduction

While everything looks straightforward for undirected graphs, it becomes more involved in directed graphs.



1. DFS
2. Strongly connected components
3. Directed Acyclic Graph

Basic DFS

Consider the same DFS procedure:

---

procedure DFS-EXPLORE(u)
    $visited(u) = True$
    for each $v \in Adj[u]$ **do** // Time: $O(deg(u))$
        if $visited(v) = False$ **then**
            $v.\pi = u$
            DFS-EXPLORE(v)

Basic DFS

Consider the same DFS procedure:

---

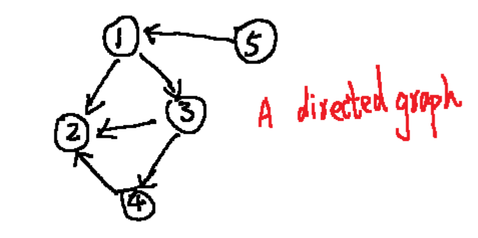
procedure DFS-EXPLORE(u)
  $visited(u) = True$
  for each $v \in Adj[u]$ **do** // Time: $O(deg(u))$
    if $visited(v) = False$ **then**
      $v.\pi = u$
      DFS-EXPLORE(v)

Fact

In directed graphs, it finds all vertices reachable from u.

Next question: How about strongly connected components?

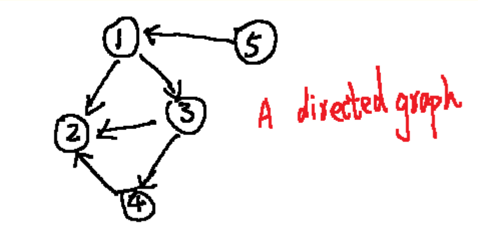
Strongly Connected Components



Question

How many strongly connected components?

Strongly Connected Components

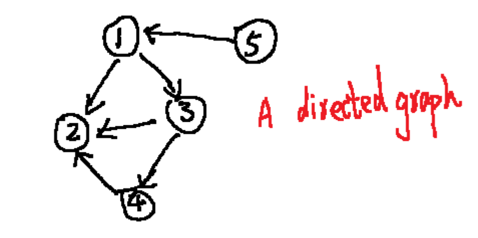


Question

How many strongly connected components?

Ans: 5, each vertex contribute a component.

Strongly Connected Components



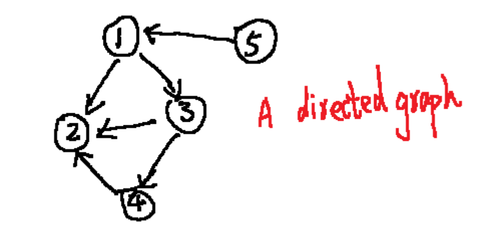
Question

How many strongly connected components?

Ans: 5, each vertex contribute a component.

1. DFS-EXPLORE does not find a connected component every time.
2. Discuss forward/backward edges and cross edges again.

Strongly Connected Components



Question

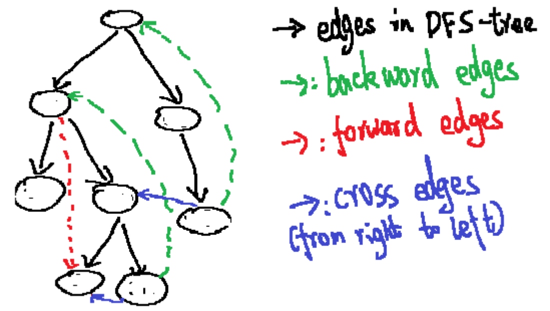
How many strongly connected components?

Ans: 5, each vertex contribute a component.

1. DFS-EXPLORE does not find a connected component every time.
2. Discuss forward/backward edges and cross edges again.
3. Define time-stamps for vertices in DFS
4. Show the algorithm to find connected components in linear time

Types of Edges

In a directed graph, DFS procedure labels all edges with 4 types:



Exception: No edge from left to right.

Observation

Strongly conn. components are defined by backward edges.

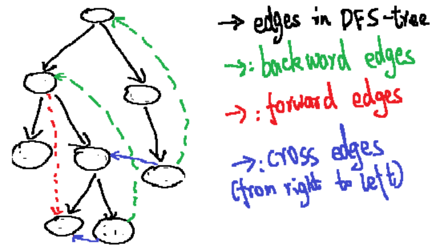
Time Stamps

Define time-stamps to (1) determine the type of each edge and (2) use backward edges to find conn. components in linear time $O(n+m)$

---

procedure DFS-EXPLORE(u)
    $t = t + 1$; $u.d = t$; // Discover u at t
    $visited(u) = True$
    for each $v \in Adj[u]$ **do**
        if $visited(v) = False$ **then**
            $v.\pi = u$
            DFS-EXPLORE(v)
    $t = t + 1$; $u.f = t$; // Finish at t
procedure DFS(G)
    $t = 0$
    for each $v \in V$ **do**
        $visited(v) = False$
        $v.\pi = NIL$
    for each $v \in V$ **do**
        if $visited(v) = False$ **then**
            DFS-EXPLORE(v)

Example

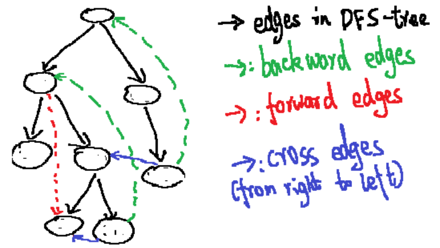


Theorem 22.7 in CLRS

For two vertices u and v,

1. $[v.d, v.f] \subset [u.d, u.f]$: v is a descendant of u
2. $[u.d, u.f] \subset [v.d, v.f]$: u is a descendant of v
3. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint: Neither u nor v is a descendant of the other in DFS

Example

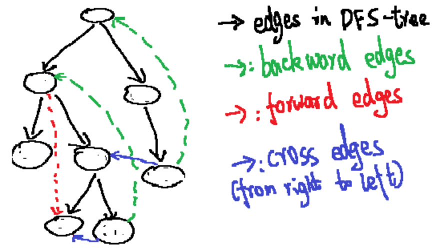


Theorem 22.7 in CLRS

For two vertices u and v,

1. $[v.d, v.f] \subset [u.d, u.f]$: v is a descendant of u
2. $[u.d, u.f] \subset [v.d, v.f]$: u is a descendant of v
3. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint: Neither u nor v is a descendant of the other in DFS

Example



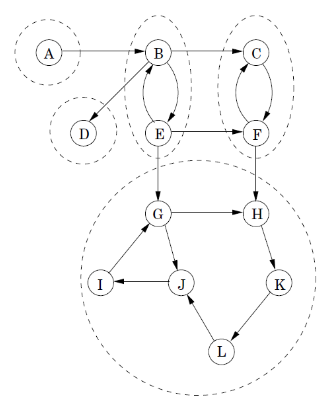
Theorem 22.7 in CLRS

For two vertices u and v,

1. $[v.d, v.f] \subset [u.d, u.f]$: v is a descendant of u
2. $[u.d, u.f] \subset [v.d, v.f]$: u is a descendant of v
3. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint: Neither u nor v is a descendant of the other in DFS

These properties hold for both directed and undirected DFS.

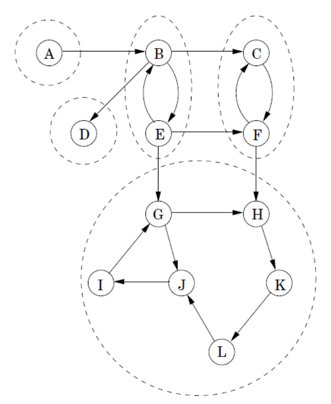
Find Connected Components



Intuition:

1. If DFS starts with a sink conn. component like D or $\{G, H, K, I, J, L\}$, it finds a correct component

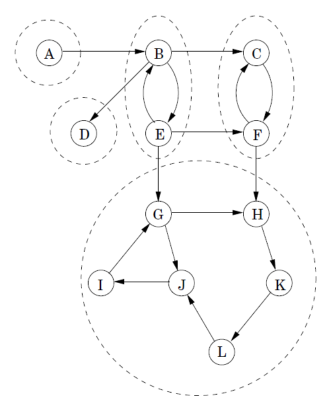
Find Connected Components



Intuition:

1. If DFS starts with a sink conn. component like D or $\{G, H, K, I, J, L\}$, it finds a correct component
2. What if DFS starts from B?

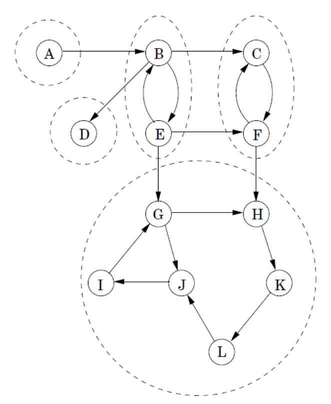
Find Connected Components



Intuition:

1. If DFS starts with a sink conn. component like D or $\{G, H, K, I, J, L\}$, it finds a correct component
2. What if DFS starts from B?
3. ☹ DFS finds $\{B, E\} \cup \{D\} \cup \{C, F\} \cup \{G, H, K, I, J, L\}$
4. Question: Find a way to start from D or $\{G, H, K, I, J, L\}$?

Find Connected Components



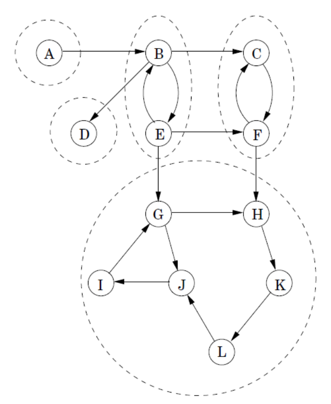
Intuition:

1. If DFS starts with a sink conn. component like D or $\{G, H, K, I, J, L\}$, it finds a correct component
2. What if DFS starts from B?
3. ☹ DFS finds $\{B, E\} \cup \{D\} \cup \{C, F\} \cup \{G, H, K, I, J, L\}$
4. Question: Find a way to start from D or $\{G, H, K, I, J, L\}$?

Observation

After Procedure DFS marks all vertices, the node u that receives the highest $u.f$ must lie in a source conn. component.

Find Connected Components



Intuition:

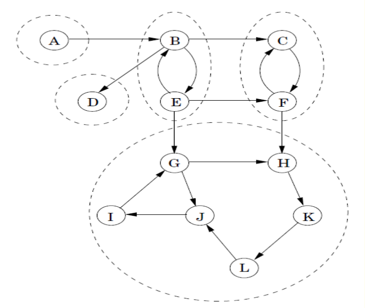
1. If DFS starts with a sink conn. component like D or $\{G, H, K, I, J, L\}$, it finds a correct component
2. What if DFS starts from B?
3. ☹ DFS finds $\{B, E\} \cup \{D\} \cup \{C, F\} \cup \{G, H, K, I, J, L\}$
4. Question: Find a way to start from D or $\{G, H, K, I, J, L\}$?

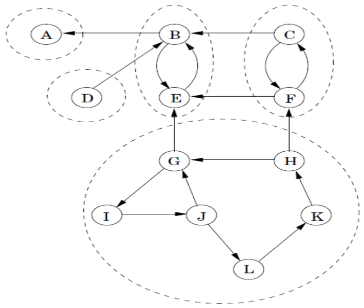
Observation

After Procedure DFS marks all vertices, the node u that receives the highest $u.f$ must lie in a source conn. component.

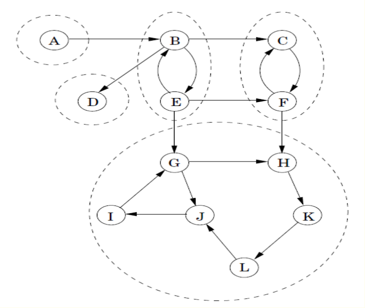
The last node must be A

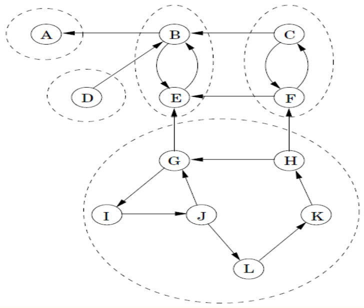
Find source conn. component by considering the reverse graph $G^\top$





Find source conn. component by considering the reverse graph $G^\top$





Description

STRONGLY-CONNECTED-COMPONENTS (G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

Analysis

Running time: $O(n+m)$.

Correctness: Lemma 22.14 in CLRS

If C and C' are two strongly connected components with an edge from C to C', then the highest $u.f$ in C is bigger than the highest $u'.f$ in C'.

Analysis

Running time: $O(n+m)$.

Correctness: Lemma 22.14 in CLRS

If C and C' are two strongly connected components with an edge from C to C', then the highest $u.f$ in C is bigger than the highest $u'.f$ in C'.

Proof: 2 cases

1. Visit C before C'
2. Visit C' before C

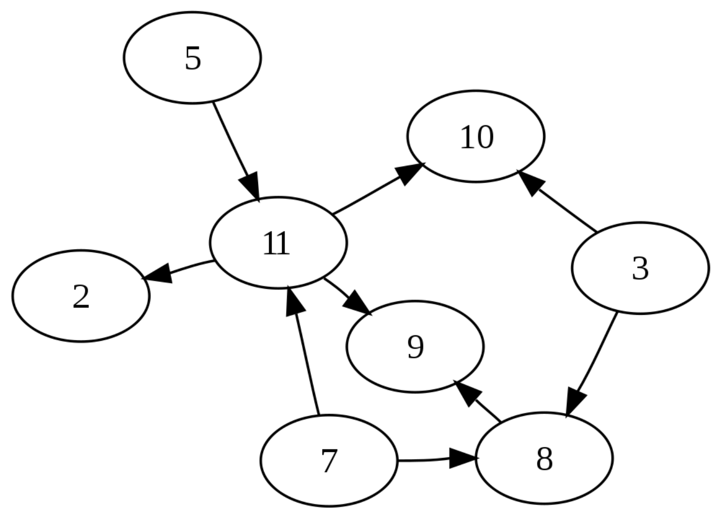
This lemma indicates that Line 3 picks a source conn. component in G, which is a sink conn. component in $G^\top$.

Outline

1. Introduction
2. Representations of Graphs
3. DFS on undirected graphs
4. DFS on directed graphs
5. Topological Order

Directed Acyclic Graph

If there is no cycle in G, all conn. components are of size 1 — called acyclic.



Topological Order

A order σ on all vertices such that u appears before v in σ whenever $(u, v) \in G$.

Applications

1. Dependency relation: compiler, resource managements, ...
2. Time order: Data processing, sociology, ...
3. Biology: Evolution, ...

Topological Order

A order σ on all vertices such that u appears before v in σ whenever $(u, v) \in G$.

Applications

1. Dependency relation: compiler, resource managements, ...
2. Time order: Data processing, sociology, ...
3. Biology: Evolution, ...

Question: Given a DAG, how to compute the order?

Algorithm

1. Call $DFS(G)$ to compute finishing time $v.f$
2. Once finish processing a note v, add it to the front of the list
3. Output the list

Algorithm

1. Call $DFS(G)$ to compute finishing time $v.f$
2. Once finish processing a note v, add it to the front of the list
3. Output the list

Analysis

1. Running time: $O(n+m)$
2. Correctness: If $\exists (u,v) \in E$, $v.f < u.f$ is always true.

Questions?