
# Algorithm Design VI

## Decompositions of Graphs

Guoqiang Li
School of Software, Shanghai Jiao Tong University

SHANGHAI JIAO TONG UNIVERSITY

# An Exercise



Let $B$ be an $n \times n$ chessboard, where $n$ is a power of 2. Use a divide-and-conquer argument to describe how to cover all squares of $B$ except one with $L$-shaped tiles. For example, if $n = 2$, then there are four squares three of which can be covered by one $L$-shaped tile, and if $n = 4$, then there are 16 squares of which 15 can be covered by 5 $L$-shaped tiles.

# Decompositions of Graphs

# Exploring Graphs

```
EXPLORE(G, v)
```

**input** : $G = (V, E)$ is a graph; $v \in V$

**output:** $visited(u)$ to $true$ for all nodes $u$ reachable from $v$

$visited(v) = true$;

PREVISIT($v$);

**for** *each edge* $(v, u) \in E$ **do**

  **if** *not* $visited(u)$ **then** EXPLORE($G$, $u$);

**end**

POSTVISIT($v$);

## Depth-First Search

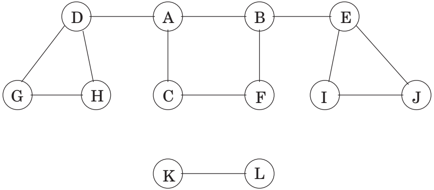
```
DFS(G)

for all v ∈ V do
    visited(v) = false;
end
for all v ∈ V do
    if not visited(v) then Explore(G, v);
end
```

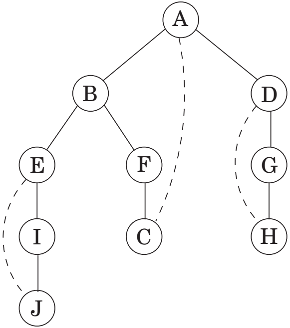
# Connectivity in Undirected Graphs

## Types of Edges in Undirected Graphs

Those edges in $G$ that are traversed by `EXPLORE` are tree edges.

The rest are back edges.



# Connectivity in Undirected Graphs



**Definition**

An undirected graph is connected, if there is a path between any pair of vertices.

**Definition**

A connected component is a subgraph that is internally connected but has no edges to the remaining vertices.

When `EXPLORE` is started at a particular vertex, it identifies precisely the connected component containing that vertex.

Each time the DFS outer loop calls `EXPLORE`, a new connected component is picked out.

# Connectivity in Undirected Graphs

DFS is trivially adapted to check if a graph is connected.

More generally, to assign each node $v$ an integer $ccnum[v]$ identifying the connected component to which it belongs.

```
PREVISIT(v)
```

$ccnum[v] = cc;$

where $cc$ needs to be initialized to zero and to be incremented each time the `DFS` procedure calls `EXPLORE`.

## Previsit and Postvisit Orderings



For each node, we will note down the times of two important events:

- the moment of first discovery (corresponding to `PREVISIT`);
- and the moment of final departure (`POSTVISIT`).

```
PREVISIT(v)
```

$pre[v] = clock;$
$clock++;$

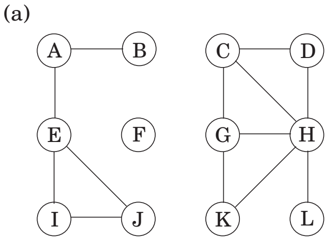
```
POSTVISIT(v)
```

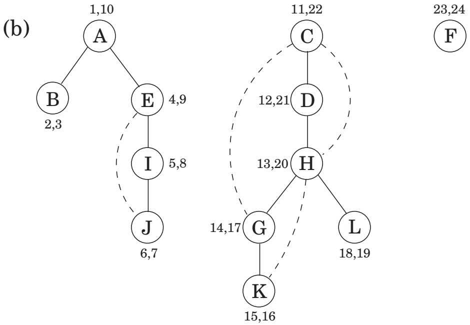
$post[v] = clock;$
$clock++;$

**Lemma**

*For any nodes $u$ and $v$, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other.*

# Previsit and Postvisit Orderings



# Connectivity in Directed Graphs

## Types of Edges in Directed Graphs

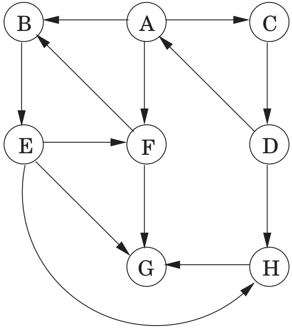


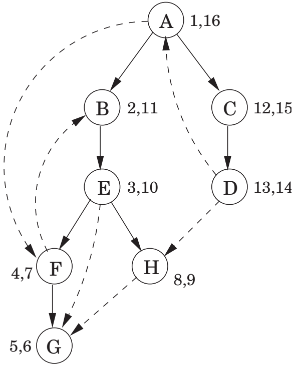
DFS yields a search tree/forests.

- root.
- descendant and ancestor.
- parent and child.
- Tree edges are actually part of the DFS forest.
- Forward edges lead from a node to a nonchild descendant in the DFS tree.
- Back edges lead to an ancestor in the DFS tree.
- Cross edges lead to neither descendant nor ancestor.

# Directed Graphs





# Types of Edges

| pre/post ordering for $(u, v)$ | | | | Edge type |
|---|---|---|---|---|
| $[_u$ | $[_v$ | $]_v$ | $]_u$ | Tree/forward |
| $[_v$ | $[_u$ | $]_u$ | $]_v$ | Back |
| $[_v$ | $]_v$ | $[_u$ | $]_u$ | Cross |

Q: Is that all?

# Directed Acyclic Graphs (DAG)



**Definition**

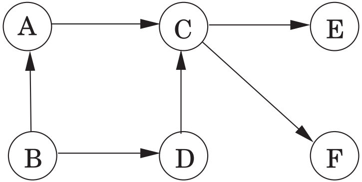
A cycle in a directed graph is a circular path

$$v_0 \to v_1 \to v_2 \to \ldots v_k \to v_0$$

**Lemma**

*A directed graph has a cycle if and only if its depth-first search reveals a back edge.*

# Directed Acyclic Graphs (DAG)



# Directed Acyclic Graphs (DAG)



Linearization/Topologically Sort: Order the vertices such that every edge goes from a earlier vertex to a later one.

Q: What types of dags can be linearized?

A: All of them.

DFS tells us exactly how to do it: perform tasks in decreasing order of their post numbers.

The only edges $(u, v)$ in a graph for which $post(u) < post(v)$ are back edges, and we have seen that a DAG cannot have back edges.

# Directed Acyclic Graphs (DAG)

SHANGHAI JIAO TONG UNIVERSITY

**Lemma**

*In a DAG, every edge leads to a vertex with a lower post number.*

# Directed Acyclic Graphs (DAG)

SHANGHAI JIAO TONG UNIVERSITY

There is a linear-time algorithm for ordering the nodes of a DAG.

Acyclicity, linearizability, and the absence of back edges during a depth-first search - are the same thing.

The vertex with the smallest post number comes last in this linearization, and it must be a sink - no outgoing edges.

Symmetrically, the one with the highest post is a source, a node with no incoming edges.

**Lemma**

*Every DAG has at least one source and at least one sink.*

The guaranteed existence of a source suggests an alternative approach to linearization:

1. Find a source, output it, and delete it from the graph.
2. Repeat until the graph is empty.

# Strongly Connected Components

# Defining Connectivity for Directed Graphs



**Definition**

Two nodes $u$ and $v$ of a directed graph are connected if there is a path from $u$ to $v$ and a path from $v$ to $u$.

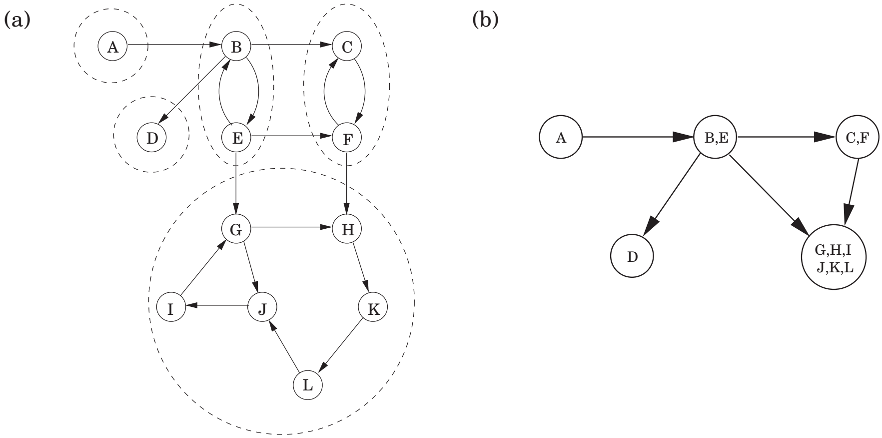
This relation partitions $V$ into disjoint sets that we call strongly connected components (SCC).

**Lemma**

*Every directed graph is a DAG of its SCC.*

# Strongly Connected Components





## An Efficient Algorithm



**Lemma**

*If the* `EXPLORE` *subroutine at node* $u$*, then it will terminate precisely when all nodes reachable from* $u$ *have been visited.*

If we call explore on a node that lies somewhere in a sink SCC, then we will retrieve exactly that component.

We have two problems:

1. How do we find a node that we know for sure lies in a sink SCC?
2. How do we continue once this first component has been discovered?

# An Efficient Algorithm



**Lemma**

*The node that receives the highest post number in a depth-first search must lie in a source SCC.*

**Lemma**

*If $C$ and $C'$ are SCC, and there is an edge from a node in $C$ to a node in $C'$, then the highest post number in $C$ is bigger than the highest post number in $C'$.*

Hence the SCCs can be linearized by arranging them in decreasing order of their highest post numbers.

## Solving Problem A

Consider the reverse graph $G^R$, the same as $G$ but with all edges reversed.

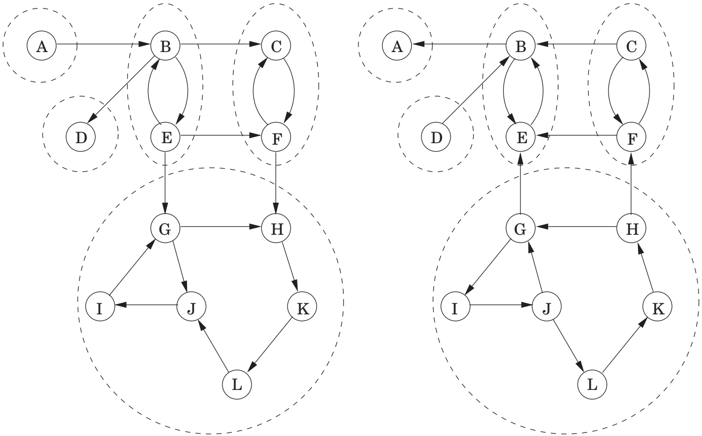
$G^R$ has exactly the same SCCs as $G$.

If we do a depth-first search of $G^R$, the node with the highest post number will come from a source SCC in $G^R$.

It is a sink SCC in $G$.

# Strongly Connected Components

Shanghai Jiao Tong University



# Solving Problem B



Once we have found the first SCC and deleted it from the graph, the node with the highest post number among those remaining will belong to a sink SCC of whatever remains of $G$.

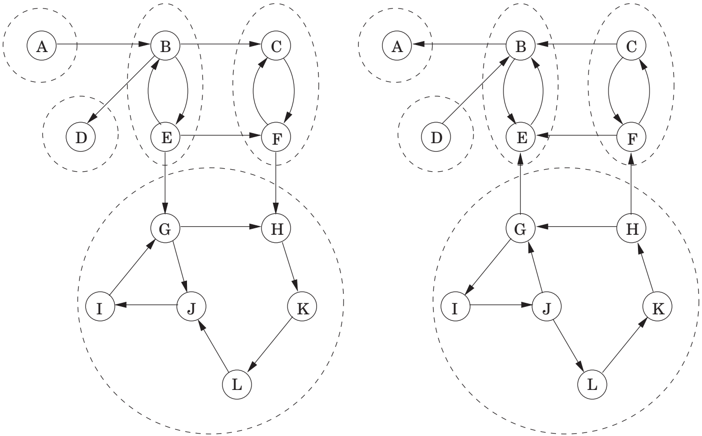
Therefore we can keep using the post numbering from our initial depth-first search on $G^R$ to successively output the second strongly connected component, the third SCC, and so on.

# The Linear-Time Algorithm

1. Run depth-first search on $G^R$.
2. Run the EXPLORE algorithm on $G$, and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1.

# Strongly Connected Components





# Think About



How the SCC algorithm works when the graph is very, very huge?

# Think About

How about edges instead of paths?

## Exercises

# Exercises 1

Suppose a CS curriculum consists of $n$ courses, all of them mandatory. The prerequisite graph $G$ has a node for each course, and an edge from course $v$ to course $w$ if and only if $v$ is a prerequisite for $w$. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). The running time of your algorithm should be linear.

## Exercises 2



Give an efficient algorithm which takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.