Algorithms Design and Analysis

Day 5 Traversing a Graph

2015, AUT-CJLU

Part I: Depth First Search

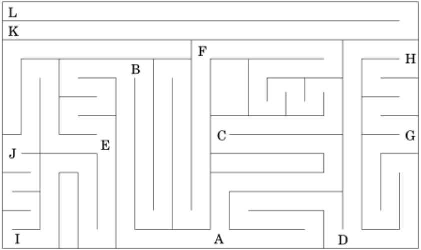
Graph Traversal

Question

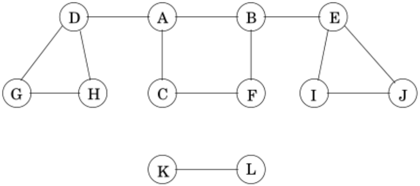
If I use a digraph to store a collection of data, how can I search for information in the graph?

Answer

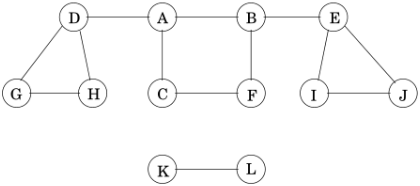
Traverse through each node of the graph.



Graph Traversal



Graph Traversal



String: Keep track of the path we are currently on

Chalk: Mark a node after we have finished visiting it

Simulating String and Chalk

Question

Can we use an algorithm to simulate the "string+chalk" procedure to traverse a graph?

Simulating String and Chalk

Question

Can we use an algorithm to simulate the "string+chalk" procedure to traverse a graph?

Strategy: Each node is processed in two stages:

- Stage 1. A node is discovered (preprocessed): the first time it is visited
- Stage 2. A node is finished (postprocessed): the last time it is visited

Simulating String and Chalk

Question

Can we use an algorithm to simulate the "string+chalk" procedure to traverse a graph?

Strategy: Each node is processed in two stages:

- Stage 1. A node is discovered (preprocessed): the first time it is visited
- Stage 2. A node is finished (postprocessed): the last time it is visited

Graph Traversal Problem

INPUT: A (representation of) digraph G
OUTPUT: Enumeration of all nodes in the digraph

We would like a traversal algorithm that reveals also the link topology of the graph.

Depth First Search

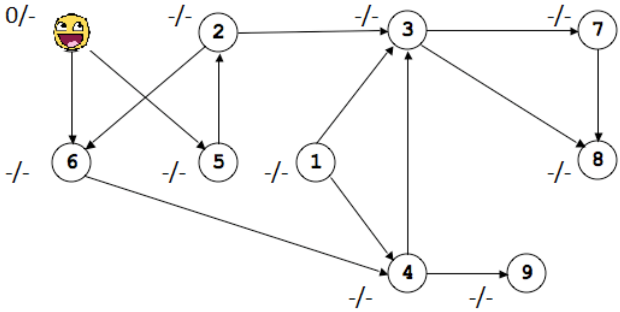
Strategy: Each node is processed in two stages:

- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited

Depth First Search

Strategy: Each node is processed in two stages:

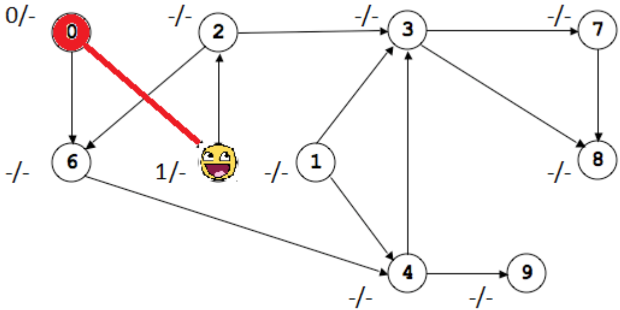
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

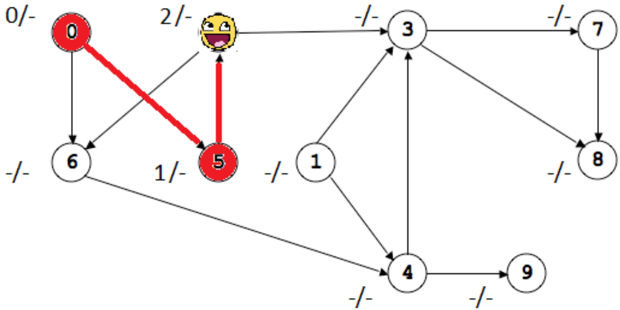
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

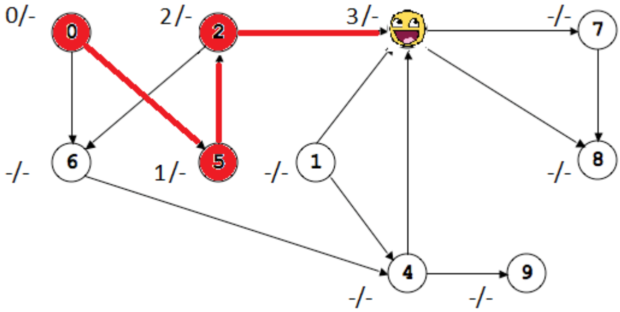
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

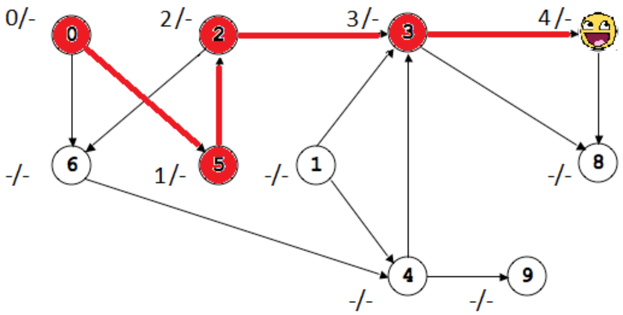
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

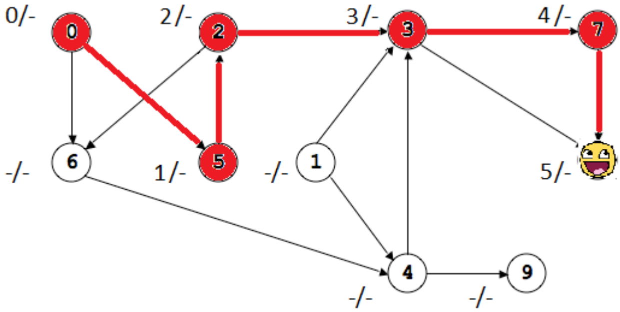
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

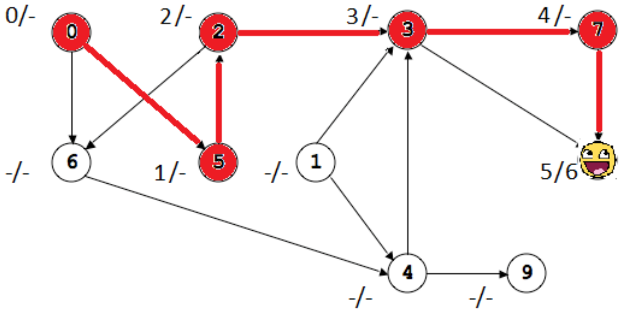
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

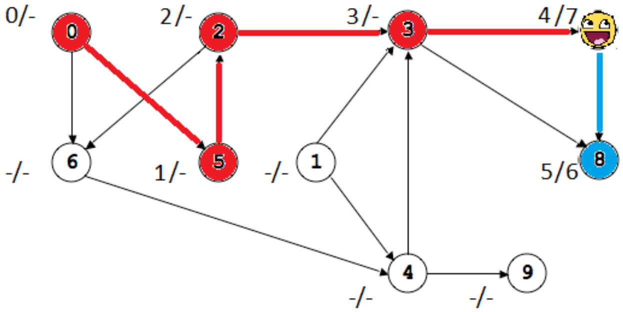
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

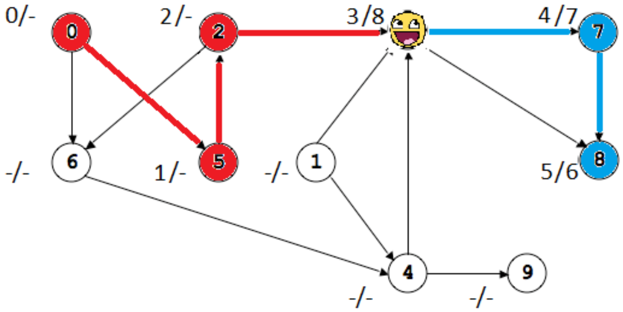
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

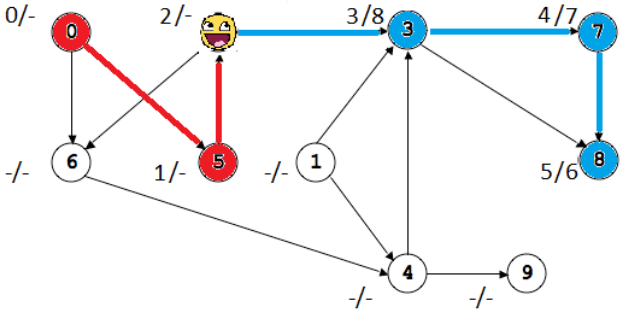
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

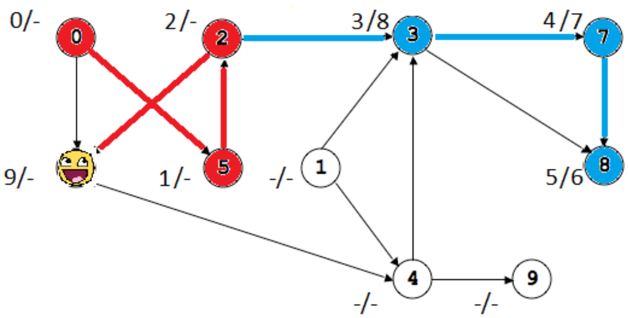
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

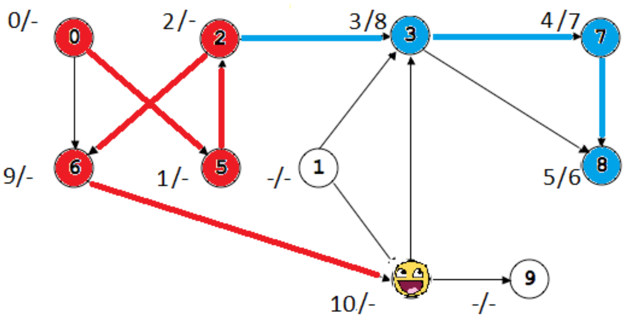
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

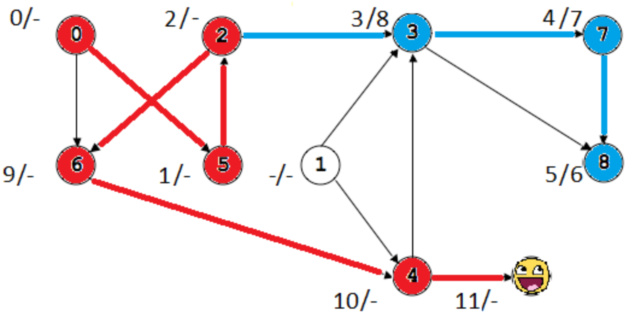
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

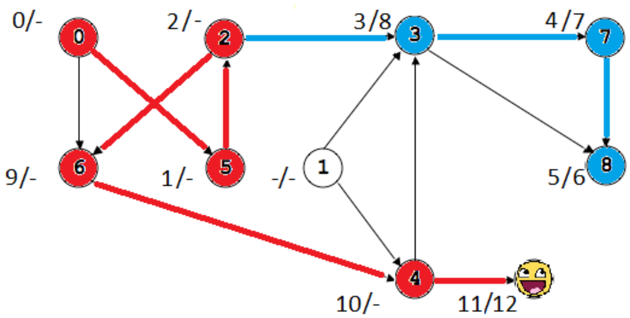
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

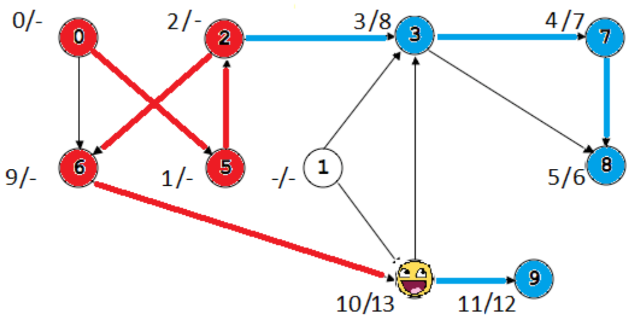
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

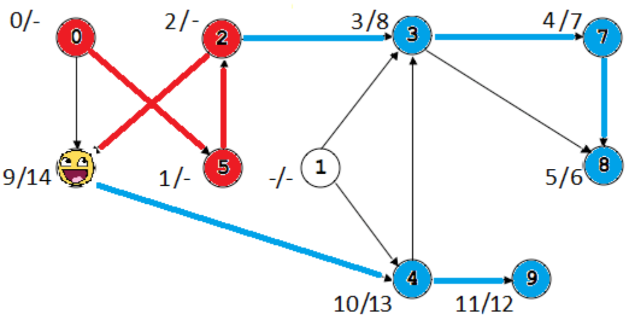
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

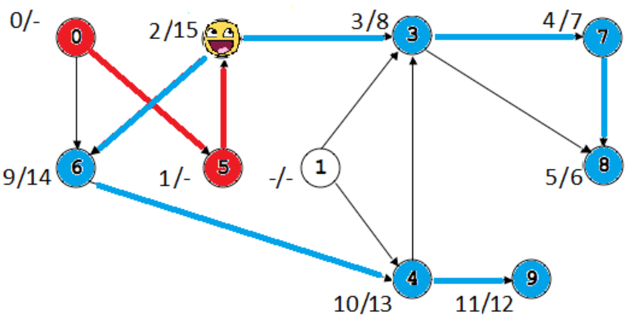
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

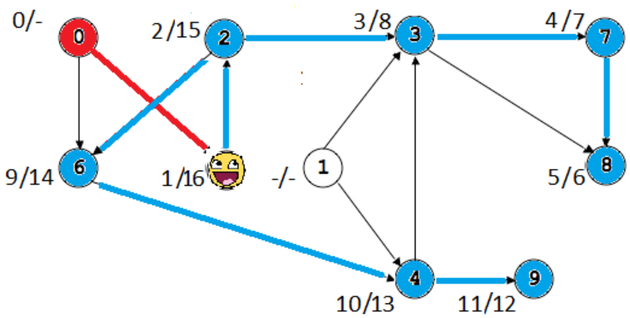
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

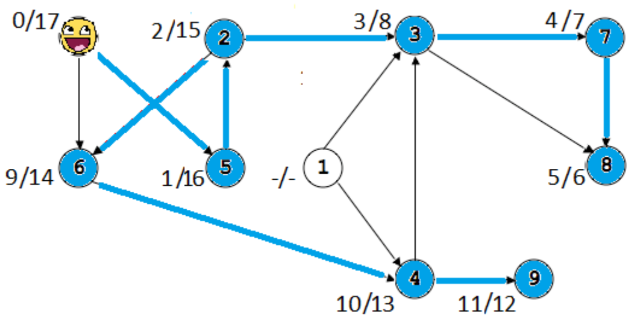
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

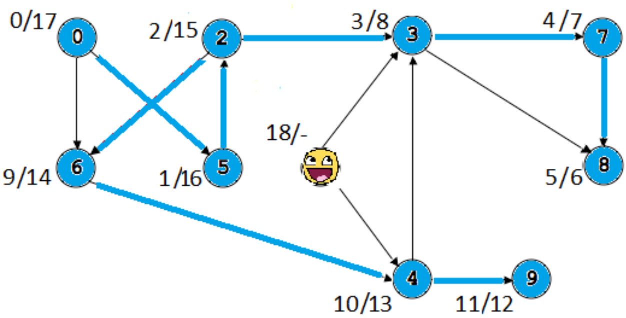
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

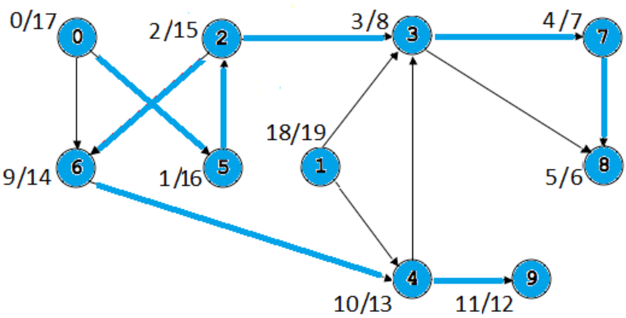
- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search

Strategy: Each node is processed in two stages:

- Stage 1. A node is discovered: the first time it is visited
- Stage 2. A node is finished: the last time it is visited



Depth First Search: Recursive Implementation

Maintain visited(v) for every $v \in V$.

Algorithm explore(G, v)

INPUT: A digraph G and a node v

```
visited(v) ← true
call discover(v) (perform operations to discover v)
for (v,u) ∈ E do
    if ¬ visited(u) do
        call explore(G,u)
call finish(v) (perform operations to finish v)
```

Algorithm dfs(G)

INPUT: A digraph G

```
for v ∈ V do
    visited(v) ← false
for v ∈ V do
    if ¬ visited(v) do
        call explore(G,v)
```

Depth First Search: Stack Implementation

Note: The current path changes in a FILO order.

Algorithm explore_stack(G, v)

INPUT: A digraph G, and a starting node v
create an empty stack S
push v to S
visited$(v) \leftarrow$ *true*

```
while  S ≠ ∅ do
    u ← top element of S
    call discover(u)
    w ← first node such that (u,w) ∈ E and visited(w) is
false
    if w does not exist then do
        call finish(u)
        pop u from S
    else do
        push w to S
        visited(w) ← true
```

Depth First Search: Complexity

Analysis

Depth First Search: Complexity

Analysis

- Discover and finish each node
- Visiting the out-neighbours of each node

Depth First Search: Complexity

Analysis

- Discover and finish each node: $O(n)$
- Visiting the out-neighbours of each node: $O(n+m)$ (with adj.list); $O(n^2)$ (with adj.matrix)

Depth First Search: Complexity

Analysis

- Discover and finish each node: $O(n)$
- Visiting the out-neighbours of each node:
  $O(n+m)$ (with adj.list); $O(n^2)$ (with adj.matrix)

Fact

The DFS algorithm takes $O(n+m)$ time with adjacency list and $O(n^2)$ with adjacency matrix.

DFS and Reachability

Definition: Reachability

We say a node u is reachable from a node v in a graph G if there is a path that starts at v and ends at u.

DFS and Reachability

Definition: Reachability

We say a node u is reachable from a node v in a graph G if there is a path that starts at v and ends at u.

Fact.

Suppose we run explore(G, v) on input graph G and node v in G, any node u is visited by the algorithm if and only if it is reachable from v.

DFS and Reachability

Definition: Reachability

We say a node u is reachable from a node v in a graph G if there is a path that starts at v and ends at u.

Fact.

Suppose we run explore(G, v) on input graph G and node v in G, any node u is visited by the algorithm if and only if it is reachable from v.

Why?

DFS and Reachability

Definition: Reachability

We say a node u is reachable from a node v in a graph G if there is a path that starts at v and ends at u.

Fact.

Suppose we run explore(G, v) on input graph G and node v in G, any node u is visited by the algorithm if and only if it is reachable from v.

Why?

1. If u is visited, then u is reachable.
   True, as we only followed edges in G.

DFS and Reachability

Definition: Reachability

We say a node u is reachable from a node v in a graph G if there is a path that starts at v and ends at u.

Fact.

Suppose we run explore(G, v) on input graph G and node v in G, any node u is visited by the algorithm if and only if it is reachable from v.

Why?

1. If u is visited, then u is reachable.

   True, as we only followed edges in G.

2. If u is reachable, then u is visited.

DFS and Reachability

Definition: Reachability

We say a node u is reachable from a node v in a graph G if there is a path that starts at v and ends at u.

Fact.

Suppose we run explore(G, v) on input graph G and node v in G, any node u is visited by the algorithm if and only if it is reachable from v.

Why?

1. If u is visited, then u is reachable.
   True, as we only followed edges in G.
2. If u is reachable, then u is visited.

Proof. Suppose w is reachable but not visited.
   Then there is a path $v \rightsquigarrow w$.
   Take the last visited u on the path ($v \rightsquigarrow u \to u' \rightsquigarrow w$).
   Then we must visit u' from u. Contradiction.

Depth First Search and Search Forest

Definition [Search Forest]

- A forest is a collection of trees.
- DFS defines a forest in the digraph. We call this forest the DFS forest.
- The DFS forest contains all paths DFS used to visit nodes in G.

Depth First Search and Search Forest

Definition [Search Forest]

- A forest is a collection of trees.
- DFS defines a forest in the digraph. We call this forest the DFS forest.
- The DFS forest contains all paths DFS used to visit nodes in G.

Question

How could we identify the search forest while running DFS?

Depth First Search and Search Forest

Definition [Search Forest]

- A forest is a collection of trees.
- DFS defines a forest in the digraph. We call this forest the DFS forest.
- The DFS forest contains all paths DFS used to visit nodes in G.

Question

How could we identify the search forest while running DFS?

Solution

Maintain a timer in the algorithm, and two times $pre(u)$ and $post(u)$ for each node u

Depth First Search and Search Forest

pre(u) **and post**(u)

procedure discover(v)	procedure finish(v)
$\text{pre}(v) \leftarrow \textit{clock}$	$\text{post}(v) \leftarrow \textit{clock}$
$\textit{clock} \leftarrow \textit{clock} + 1$	$\text{clock} \leftarrow \textit{clock} + 1$

Depth First Search and Search Forest

pre(u) and post(u)

procedure discover(v)	procedure finish(v)
$\text{pre}(v) \leftarrow \textit{clock}$	$\text{post}(v) \leftarrow \textit{clock}$
$\textit{clock} \leftarrow \textit{clock} + 1$	$\text{clock} \leftarrow \textit{clock} + 1$

Observation

If u is an ancestor of v, then

$$pre(u) < pre(v) < post(v) < post(u)$$

If v is an ancestor of v, then

$$pre(v) < pre(u) < post(u) < post(v)$$

If neither case, then

$$pre(u) < post(u) < pre(v) < post(v) \text{ or}$$
$$pre(v) < post(v) < pre(u) < post(u)$$

Depth First Search and Search Forest

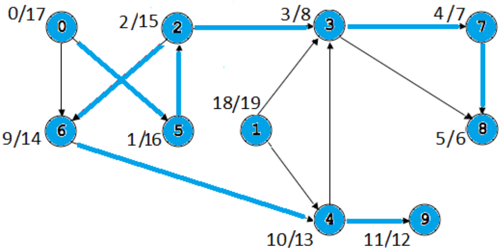
Therefore we can represent the search forest in parenthesis form:

DFS-Forest(G)

Write down a sequence of symbols $(n$ and $n)$, where $n \in \{1, \ldots, n\}$ such that:

If $pre(u) = k$, then the kth symbol is $(u$
If $post(u) = k$, then the kth symbol is $u)$

e.g. (0 (5 (2 (3 (7 (8 8) 7) 3) (6 (4 (9 9) 4) 6) 2) 5) 0) (1 1)

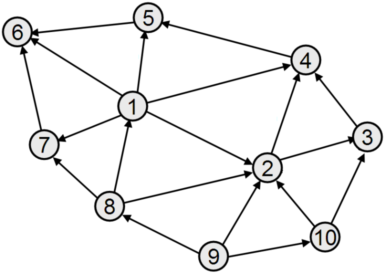


Part IV: Cyclicity and Linearisations

DFS and DAGs

Definition [DAG]

A directed acyclic graph (dag) is a digraph that does not contain a cycle.



Question

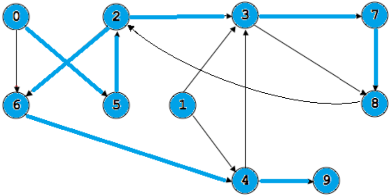
Given a digraph, decide if the digraph is a dag.

DFS and DAGs

Definition [Edge Classification]

Let T be the DFS forest in G. There are four types of edges in G:

- If (u, v) belongs to the search forest, (u, v) is a tree edge;
- Otherwise if u is an ancestor of v in T, (u, v) is a forward edge;
- Otherwise if v is an ancestor of u in T, (u, v) is a back edge;
- Otherwise (u, v) is a cross edge.



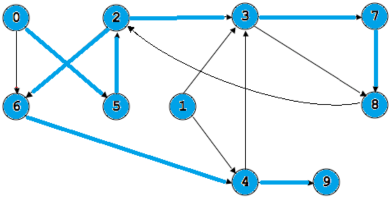
Tree edges: (0,5)(5,2),(2,6),(2,3),(3,7),(7,8),(6,4),(4,9)

DFS and DAGs

Definition [Edge Classification]

Let T be the DFS forest in G. There are four types of edges in G:

- If (u, v) belongs to the search forest, (u, v) is a tree edge;
- Otherwise if u is an ancestor of v in T, (u, v) is a forward edge;
- Otherwise if v is an ancestor of u in T, (u, v) is a back edge;
- Otherwise (u, v) is a cross edge.



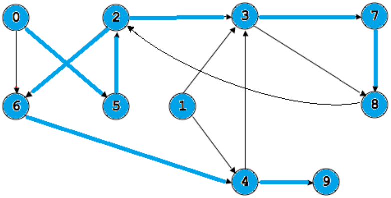
Tree edges: (0,5)(5,2),(2,6),(2,3),(3,7),(7,8),(6,4),(4,9)
Forward edges: (0,6),(3,8)

DFS and DAGs

Definition [Edge Classification]

Let T be the DFS forest in G. There are four types of edges in G:

- If (u, v) belongs to the search forest, (u, v) is a tree edge;
- Otherwise if u is an ancestor of v in T, (u, v) is a forward edge;
- Otherwise if v is an ancestor of u in T, (u, v) is a back edge;
- Otherwise (u, v) is a cross edge.

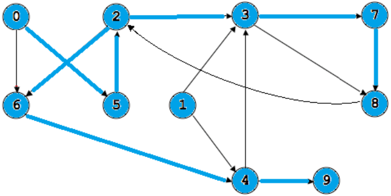


Tree edges: (0,5)(5,2),(2,6),(2,3),(3,7),(7,8),(6,4),(4,9)
Forward edges: (0,6),(3,8) Back edges: (8,2)

DFS and DAGs

Definition [Edge Classification]

Let T be the DFS forest in G. There are four types of edges in G:
- If (u, v) belongs to the search forest, (u, v) is a tree edge;
- Otherwise if u is an ancestor of v in T, (u, v) is a forward edge;
- Otherwise if v is an ancestor of u in T, (u, v) is a back edge;
- Otherwise (u, v) is a cross edge.



Tree edges: (0,5)(5,2),(2,6),(2,3),(3,7),(7,8),(6,4),(4,9)
Forward edges: (0,6),(3,8) Back edges: (8,2)
Cross edges: (4,3),(1,3),(1,4)

DFS and DAGs

Fact.

Let G be a digraph. Then the following are equivalent:

(1). G is a DAG

(2). the DFS forest has no back edge.

DFS and DAGs

Fact.

Let G be a digraph. Then the following are equivalent:

(1). G is a DAG

(2). the DFS forest has no back edge.

Proof.

DFS and DAGs

Fact.

Let G be a digraph. Then the following are equivalent:

(1). G is a DAG

(2). the DFS forest has no back edge.

Proof.

$\Rightarrow$ Suppose G is a dag, then the search forest doesn't have a back edge as otherwise, there will be a cycle.

DFS and DAGs

Fact.

Let G be a digraph. Then the following are equivalent:

(1). G is a DAG

(2). the DFS forest has no back edge.

Proof.

$\Rightarrow$ Suppose G is a dag, then the search forest doesn't have a back edge as otherwise, there will be a cycle.

$\Leftarrow$ Suppose G is not a dag, then there is a cycle C in G.
Let v be the first node discovered by the DFS in C.
Let (u, v) be the edge in C that goes into v.
Then in the search tree v is an ancestor of u.
Then (u, v) is a back edge. $\square$

DFS and DAGs

Fact

The following algorithm runs in time $O(n + m)$ and decides whether any given digraph G is a dag.

DFS and DAGs

Fact

The following algorithm runs in time $O(n + m)$ and decides whether any given digraph G is a dag.

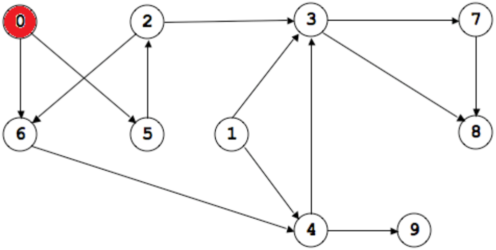
Algorithm: acyclic(G)

INPUT: A digraph G
OUTPUT: Return if G is a dag
Run DFS(G) with the following modification:
    Whenever discover a node u, do
        for every edge (u, v) out of u
            if $pre(v) < pre(u)$ and $post(v)$ is undefined
                Declare G has a cycle and return
Declare that G is a dag.

DFS and Linearisations

Definition [Linearisations]

A linearization or (topological sort) of a digraph G is a list of all nodes in G such that if G contains an edge (u, v) then u appears before v in the list.



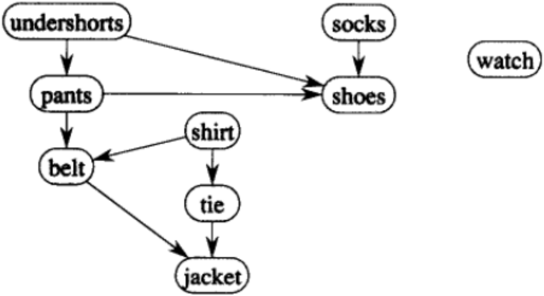
Topological Sorts:

0,5,2,6,1,4,3,7,9,8

1,0,5,2,6,4,9,3,7,8

DFS and Linearisations

In what order should I put on my cloths?

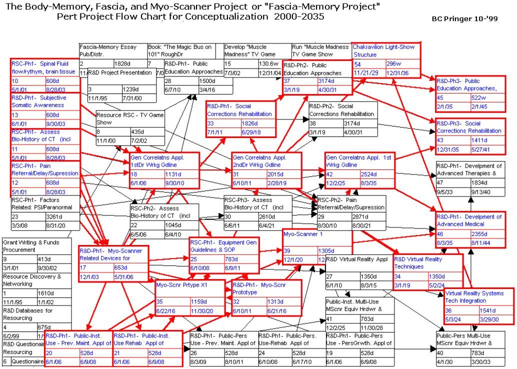


Possible orderings are linearisations of the dependency graph:
Possible order 1: Shirt, Socks, Undershorts, Watch, Pants, Tie, Belts, Jacket, Shoes
Possible order 2: Watch, Undershorts, Socks, Pants, Shoes, Shirt, Belt, Tie, Jacket

DFS and Linearisations

Application of Linearisation

- Job/Task/Instruction scheduling
- Project Evaluation and Review Technique (PERT)
- `makefiles` in Unix / `APT` in Ubuntu Linux
- Class/Package dependency in a software project



DFS and Linearisations

Question

Is there a digraph that can not be linearized?

DFS and Linearisations

Question

Is there a digraph that can not be linearized?

Answer: Yes! Digraphs with cycles.

DFS and Linearisations

Question

Is there a digraph that can not be linearized?

Answer: Yes! Digraphs with cycles.

Question

What dags can be linearized?

DFS and Linearisations

Question

Is there a digraph that can not be linearized?

Answer: Yes! Digraphs with cycles.

Question

What dags can be linearized?

Answer: All of them!

The Zero In-degree algorithm finds a linearization for a dag:

Algorithm: ZeroInDegree(G)

INPUT: a DAG G
OUTPUT: a linearisation of G
list $\leftarrow$ an empty list

```
while G is not empty do
    for each u in V
        if inDegree(u) = 0 then
            Add u to the end of list
            Delete u from G
return list
```

The Zero In-degree algorithm finds a linearization for a dag:

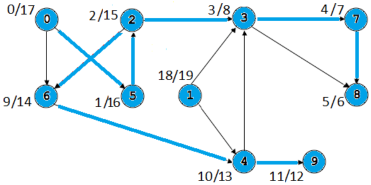
Algorithm: ZeroInDegree(G)

INPUT: a DAG G
OUTPUT: a linearisation of G
list $\leftarrow$ an empty list

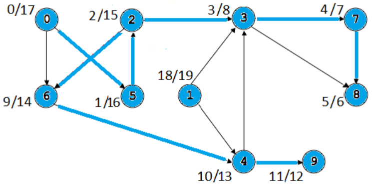
```
while G is not empty do
    for each u in V
        if inDegree(u) = 0 then
            Add u to the end of list
            Delete u from G
return list
```

Shortcoming: The algorithm runs in time $O((n + m)n)$.

DFS and Linearisations



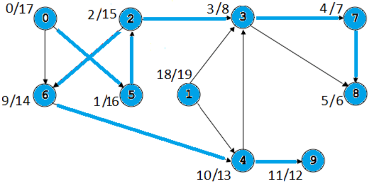
DFS and Linearisations



Fact

Let G be a dag. If (u, v) is an edge in G, then $post(v) < post(u)$.

DFS and Linearisations



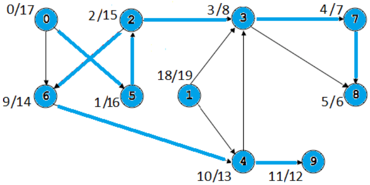
Fact

Let G be a dag. If (u, v) is an edge in G, then $post(v) < post(u)$.

Proof

There are two cases:

DFS and Linearisations



Fact

Let G be a dag. If (u, v) is an edge in G, then $post(v) < post(u)$.

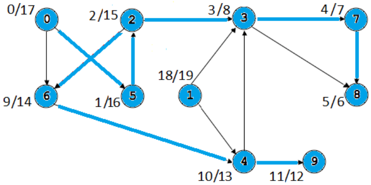
Proof

There are two cases:

Case 1. u is discovered earlier than v is.

Then v must be finished before u is finished.

DFS and Linearisations



Fact

Let G be a dag. If (u, v) is an edge in G, then $post(v) < post(u)$.

Proof

There are two cases:

Case 1. u is discovered earlier than v is.

Then v must be finished before u is finished.

Case 2. v is discovered earlier than u is.

Since G is acyclic, there is no path that goes from v to u.

Hence v is again finished earlier than u is finished. □

DFS and Linearisations

We obtain an easy algorithm for graph linearisation in time $O(m + n)$:
Output the list of nodes in decreasing finishing order

DFS and Linearisations

We obtain an easy algorithm for graph linearisation in time $O(m + n)$: Output the list of nodes in decreasing finishing order

Algorithm: DFS-Linearize(G)

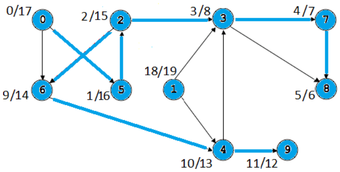
INPUT: a dag G
OUTPUT: a linearisation of G
stack $\leftarrow$ an empty stack
Run DFS, in addition:
    When a node is finished, push it to *stack*.
`return elements in` *stack* `in the same order as they are popped out`

DFS and Linearisations

We obtain an easy algorithm for graph linearisation in time $O(m+n)$: Output the list of nodes in decreasing finishing order

Algorithm: DFS-Linearize(G)

INPUT: a dag G
OUTPUT: a linearisation of G
stack $\leftarrow$ an empty stack
Run DFS, in addition:
    When a node is finished, push it to *stack*.
`return elements in` *stack* `in the same order as they are popped out`



DFS-Linearize(G) :

1, 0, 5, 2, 6, 4, 9, 3, 7, 8

Further Comments

Acyclicity and Linearizability

- We established two characterizations of linearizability of a digraph:

  A digraph is linearizable if and only if

  - it is acyclic
  - the DFS forest has no back edge

- In other words

$$\text{Linearizable} \equiv \text{Acyclicity} \equiv \text{No-Back-edgeness}$$

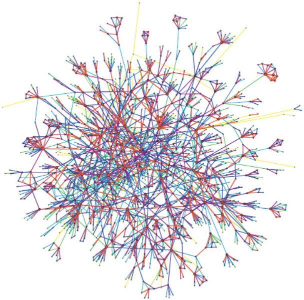
- With this understanding we are able to design algorithms for deciding these properties.

Part V: Connectivity and Components

Decomposing Graphs

Why decompose graphs?

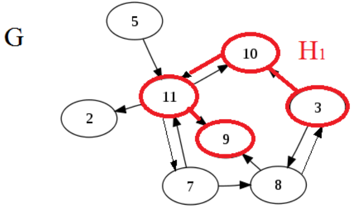
[Divide-and-Conquer] Often, when we solve a problem on graph, it is much more efficient to decompose the graph into components, solve the problem on individual components, then combine the solutions.



Subgraphs

Definition [Subgraphs]

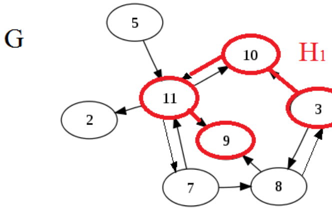
Let $E \subseteq V^2$, and $V' \subseteq V$.
- we use $E \upharpoonright V'$ to denote the set $\{(u, v) \in E \mid u, v \in V'\}$.
- A subgraph of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E \upharpoonright V'$.
- If $E' = E \upharpoonright V'$, then G' is an induced subgraph of G.



Subgraphs

Definition [Subgraphs]

Let $E \subseteq V^2$, and $V' \subseteq V$.
- we use $E \restriction V'$ to denote the set $\{(u, v) \in E \mid u, v \in V'\}$.
- A subgraph of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E \restriction V'$.
- If $E' = E \restriction V'$, then G' is an induced subgraph of G.



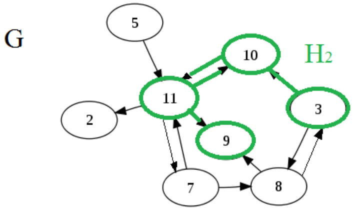
Let $V' = \{3, 9, 10, 11\}$. $E \restriction V' = \{(10, 11), (11, 10), (3, 10), (11, 9)\}$

A subgraph is $(\{3, 9, 10, 11\}, \{(3, 10), (10, 11), (11, 9)\})$

Subgraphs

Definition [Subgraphs]

Let $E \subseteq V^2$, and $V' \subseteq V$.
- we use $E \upharpoonright V'$ to denote the set $\{(u, v) \in E \mid u, v \in V'\}$.
- A subgraph of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E \upharpoonright V'$.
- If $E' = E \upharpoonright V'$, then G' is an induced subgraph of G.



Let $V' = \{3, 9, 10, 11\}$. $E \upharpoonright V' = \{(10, 11), (11, 10), (3, 10), (11, 9)\}$

An induced subgraph is $(\{3, 9, 10, 11\}, \{(3, 10), (10, 11), (11, 9), (11, 10)\})$

Decomposing Undirected Graphs

Connectivity in Undirected Graphs

- Recall a node is reachable from another if there is a path linking these two nodes

Decomposing Undirected Graphs

Connectivity in Undirected Graphs

- Recall a node is reachable from another if there is a path linking these two nodes
- Here reachability is an equivalence relation:
  - (reflexivity) Any node u is reachable from itself.
  - (symmetry) If v is reachable from u then u is reachable from v.
  - (transitivity) If v is reachable from u, u is reachable from w, then v is reachable from w.

Decomposing Undirected Graphs

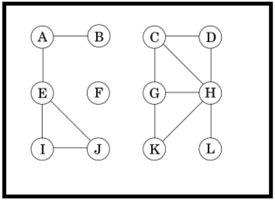
Connectivity in Undirected Graphs

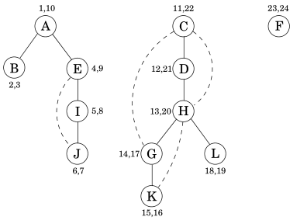
- Recall a node is reachable from another if there is a path linking these two nodes
- Here reachability is an equivalence relation:
  - (reflexivity) Any node u is reachable from itself.
  - (symmetry) If v is reachable from u then u is reachable from v.
  - (transitivity) If v is reachable from u, u is reachable from w, then v is reachable from w.
- We may decompose the graph into equivalence classes: Two nodes are in the same class if they are reachable from each other
- Each equivalence class is a connected component

Decomposing Undirected Graphs

Definition [Undirected Connectivity]

A connected components is the induced subgraph of a maximal set of nodes that are pairwise reachable.
An undirected graph is connected if it contains only one connected component.

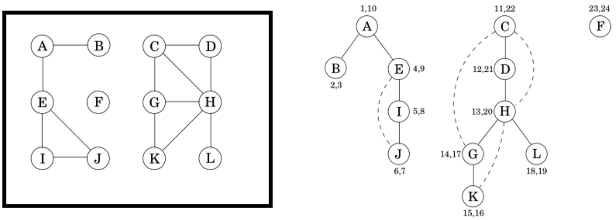




Decomposing Undirected Graphs

Definition [Undirected Connectivity]

A connected components is the induced subgraph of a maximal set of nodes that are pairwise reachable.
An undirected graph is connected if it contains only one connected component.



DFS and CC

- We may use DFS to decide if two nodes are in the same CC.
- Running time $O(m + n)$.

Decomposing Directed Graph

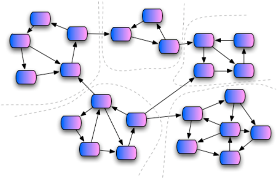
Definition [Directed Connectivity]

In a digraph G, we say that two nodes u, v are in the same strongly connected component (SCC) if there is a path from u to v and a path from v to u. A digraph is strongly connected if it contains only one SCC.

Decomposing Directed Graph

Definition [Directed Connectivity]

In a digraph G, we say that two nodes u, v are in the same strongly connected component (SCC) if there is a path from u to v and a path from v to u. A digraph is strongly connected if it contains only one SCC.



Strongly Connected Components

Extreme special cases:

Strongly Connected Components

Extreme special cases:

- If G is acyclic, then every node is itself a SCC. Therefore there are n SCCs in G.

Strongly Connected Components

Extreme special cases:

- If G is acyclic, then every node is itself a SCC.
  Therefore there are n SCCs in G.
- If G is a cycle, then G is a SCC.
  Therefore there is only 1 SCCs in G.

Strongly Connected Components

Extreme special cases:

- If G is acyclic, then every node is itself a SCC.
  Therefore there are n SCCs in G.
- If G is a cycle, then G is a SCC.
  Therefore there is only 1 SCCs in G.
- If G is undirected, then u, v are in the same SCC whenever u can reach v.
  Therefore checking SCC is same as reachability.

Strongly Connected Components

Extreme special cases:

- If G is acyclic, then every node is itself a SCC.
  Therefore there are n SCCs in G.
- If G is a cycle, then G is a SCC.
  Therefore there is only 1 SCCs in G.
- If G is undirected, then u, v are in the same SCC whenever u can reach v.
  Therefore checking SCC is same as reachability.

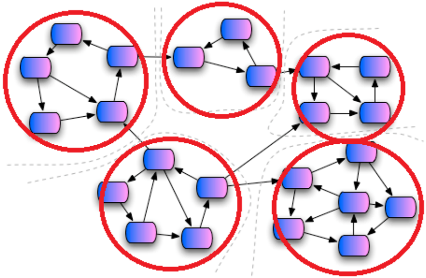
Question

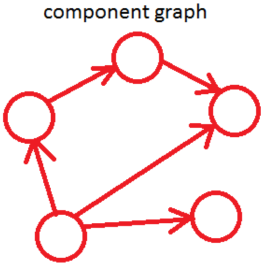
Given a digraph, find all the strongly connected components.

Meta-Graph

Definition [Meta-Graph]

Given a graph G. If we collapse all nodes in the same SCCs together, only keeping the edges between different components, then we get the meta-graph, G^{SCC}.





Meta-Graph

Let G be a digraph, and G^{SCC} be its meta-graph after collapsing every SCC into one node.

- The meta-graph G^{SCC} must be acyclic.

  Why?

Source and Sink

Meta-Graph

Let G be a digraph, and G^{SCC} be its meta-graph after collapsing every SCC into one node.

- The meta-graph G^{SCC} must be acyclic.

  Why?
- We can linearize G^{SCC}.

Source and Sink

Meta-Graph

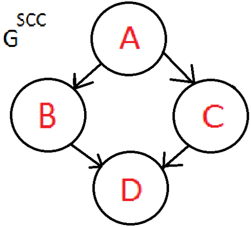
Let G be a digraph, and G^{SCC} be its meta-graph after collapsing every SCC into one node.

- The meta-graph G^{SCC} must be acyclic.

  Why?
- We can linearize G^{SCC}.
- Source: A meta-node in G^{SCC} with no incoming edge.
- Sink: A meta-node in G^{SCC} with no outgoing edge.

Source and Sink

Consider the following meta-graph:



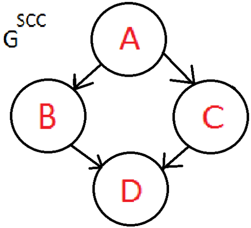
A is a source, *D* is a sink.

Observation

If we run DFS on a node in a sink, then we will find all nodes in this sink.

Finding SCC

Consider the following meta-graph:



A Plan for Finding SCC

Given G. Repeat the following:

1. Find a node u in a sink
2. Run dfs_explore(G, u)
3. Declare all visited nodes an SCC. Take those nodes out.

Finding SCC

Problem

How do we find a node in a sink?

Finding SCC

Problem

How do we find a node in a sink?

Observe

Finding SCC

Problem

How do we find a node in a sink?

Observe

- We are given the graph G, but no information about G^{SCC}.

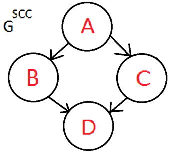
Finding SCC

Problem

How do we find a node in a sink?

Observe

- We are given the graph G, but no information about G^{SCC}.
- We can find a node in a source:
  Run DFS. Take the node that is finished last.



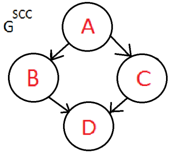
Finding SCC

Problem

How do we find a node in a sink?

Observe

- We are given the graph G, but no information about G^{SCC}.
- We can find a node in a source:
  Run DFS. Take the node that is finished last.



- But running DFS on a source does not work.

Finding SCC

Source and Sink

Does finding a source node help in finding a sink node?

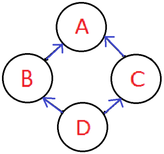
Finding SCC

Source and Sink

Does finding a source node help in finding a sink node?

Fact

Let G be a digraph. Let G^T be the transpose of G: the digraph obtained from G by reversing the direction of every edge.



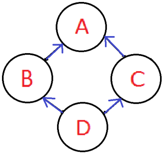
Finding SCC

Source and Sink

Does finding a source node help in finding a sink node?

Fact

Let G be a digraph. Let G^T be the transpose of G: the digraph obtained from G by reversing the direction of every edge.



- G and G^T have the same SCCs.

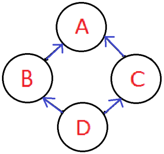
Finding SCC

Source and Sink

Does finding a source node help in finding a sink node?

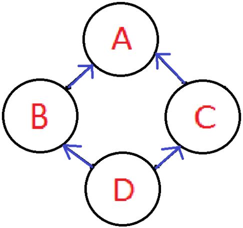
Fact

Let G be a digraph. Let G^T be the transpose of G: the digraph obtained from G by reversing the direction of every edge.

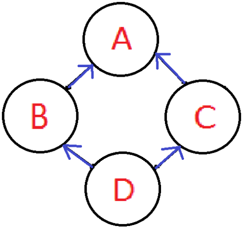


- G and G^T have the same SCCs.
- A source in G becomes a sink in G^T.

DFS and Strongly Connected Components

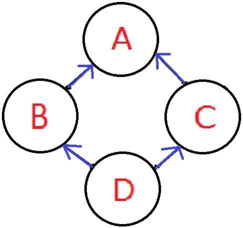


DFS and Strongly Connected Components



Example. When the edges are reversed, A, B, C, D are still SCCs.

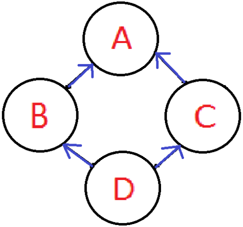
DFS and Strongly Connected Components



Example. When the edges are reversed, A, B, C, D are still SCCs.
Let x be the last finished node in DFS.

Running dfs_explore(G^T, x) will compute A.

DFS and Strongly Connected Components



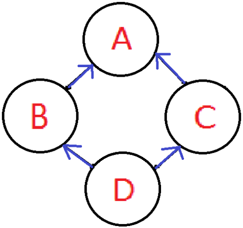
Example. When the edges are reversed, A, B, C, D are still SCCs.
Let x be the last finished node in DFS.

Running dfs_explore(G^T, x) will compute A.

Let y be the last node that is finished in the remaining graph.

Running dfs_explore(G^T, y) will compute C.

DFS and Strongly Connected Components



Example. When the edges are reversed, A, B, C, D are still SCCs.
Let x be the last finished node in DFS.

Running dfs_explore(G^T, x) will compute A.

Let y be the last node that is finished in the remaining graph.

Running dfs_explore(G^T, y) will compute C.

Continue for B, D (decreasing order of *post*)

DFS and Strongly Connected Components

The following algorithm takes as input any digraph G, outputs all the SCCs of G. The algorithm runs in time $O(m+n)$.

DFS and Strongly Connected Components

The following algorithm takes as input any digraph G, outputs all the SCCs of G. The algorithm runs in time $O(m+n)$.

Algorithm SCC(G)

INPUT: a digraph G
OUTPUT: SCCs of G
stack $\leftarrow$ empty stack
Run $dfs(G)$, at the same time do:
    When a node is finished, push it onto a *stack*
$G^T \leftarrow G$ with all edges reversed
`for each` u `in` *stack* `(in popped order)`
    `Run dfs_explore(`G^T`,`u`)`
    `The nodes visited by` **`explore`** `is the SCC of` u`.`

DFS and Strongly Connected Components

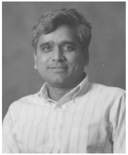
Discussion

- Essentially, the algorithm runs DFS twice: first time on G, then on G^T.
  In the second time, when no where to go, select the next node in decreasing order of finishing time of the first DFS.

DFS and Strongly Connected Components

Discussion

- Essentially, the algorithm runs DFS twice: first time on G, then on G^T.
  In the second time, when no where to go, select the next node in decreasing order of finishing time of the first DFS.
- The algorithm is called the Kosaraju-Sharir algorithm



"At some point, the learning stops and the pain begins."

------- S. Rao Kosaraju

Summary

- DFS is a linear time ($O(m + n)$) graph traversal algorithms.
- DFS implementation: Recursive or Stack-based
- Nodea are processed in two stages: discovered, finished.
- The DFS algorithm computes a DFS forest in the graph; edges in the graph are classified into tree edges, forward edges, back edges and cross edges

Using DFS we can also answer the following questions about a graph:

- Reachability: Given to nodes u, v, is u reachable from v?
- Cyclicity: Is the graph acyclic? If it contains a cycle, find a cycle.
- Linearisation: If the graph is acyclic, find a linearisation.
- Connectivity: Is the graph connected?
- Connected component: If the graph is undirected, what are the connected components of it?
  If the graph is directed, what are the strongly connected components of it?