Graph Decomposition*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

X033533-Algorithm: Analysis and Theory

*Special Thanks is given to Prof. Yijia Chen for sharing his teaching materials.

Outline

Exploring Graphs

Exploring Graphs

Algorithm 2: EXPLORE(G, v)

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

Output: VISITED(u) is set to *true* for all nodes u **reachable** from v

VISITED(v) $=$ *true*;
PREVIST(v);
for *each edge* $(v, u) \in E$ **do**
    if *not* VISITED(u) **then**
        EXPLORE(G, u);
POSTVISIT(v);

Exploring Graphs

Algorithm 3: EXPLORE(G, v)

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

Output: VISITED(u) is set to *true* for all nodes u **reachable** from v

```
VISITED(v) = true;
PREVIST(v);
for each edge (v, u) ∈ E do
    if not VISITED(u) then
        EXPLORE(G, u);
POSTVISIT(v);
```

Note: PREVISIT and POSTVISIT procedures are optional. They work on a vertex when it is first discovered and left for the last time.

Correctness Proof

Theorem: $\textsc{Explore}(G, v)$ is correct, i.e., it visits exactly all nodes that are reachable from v.

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:
EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:
EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Every node which is reachable from v must be visited eventually:

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:
EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Every node which is reachable from v must be visited eventually:
If there is some u that EXPLORE misses, choose any path from v to u, and look at the last vertex v on that path that the procedure actually visited.

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:
EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Every node which is reachable from v must be visited eventually:
If there is some u that EXPLORE misses, choose any path from v to u, and look at the last vertex v on that path that the procedure actually visited. Let w be the node immediately after it on the same path.

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:
EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Every node which is reachable from v must be visited eventually:
If there is some u that EXPLORE misses, choose any path from v to u, and look at the last vertex v on that path that the procedure actually visited. Let w be the node immediately after it on the same path.

So z was visited but w was not. This is a contradiction:

Correctness Proof

Theorem: EXPLORE(G, v) is correct, i.e., it visits exactly all nodes that are reachable from v.

Proof: Every node which it visits must be reachable from v:
EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Every node which is reachable from v must be visited eventually:
If there is some u that EXPLORE misses, choose any path from v to u, and look at the last vertex v on that path that the procedure actually visited. Let w be the node immediately after it on the same path.

So z was visited but w was not. This is a contradiction: while EXPLORE was at node z, it would have noticed w and moved on to it.

Depth-First Search

Depth-First Search

Algorithm 5: DFS(G, v)

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

Output: $\text{VISITED}(v)$ is set to *true* for all nodes $v \in V$

```
VISITED(v) = true;
foreach v ∈ V do
    VISITED(v) = false;
foreach v ∈ V do
    if not VISITED(v) then
        EXPLORE(G, v);
```

Running time of DFS

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.
2. A loop in which adjacent edges are scanned, to see if they lead somewhere new. This loop takes a different amount of time for each vertex.

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.
2. A loop in which adjacent edges are scanned, to see if they lead somewhere new. This loop takes a different amount of time for each vertex.

The total work done in step 1 is then $O(|V|)$.

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.
2. A loop in which adjacent edges are scanned, to see if they lead somewhere new. This loop takes a different amount of time for each vertex.

The total work done in step 1 is then $O(|V|)$.

In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly *twice*, once during EXPLORE(G, x) and once during EXPLORE(G, y). The overall time for step 2 is therefore $O(|E|)$.

Running time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.
2. A loop in which adjacent edges are scanned, to see if they lead somewhere new. This loop takes a different amount of time for each vertex.

The total work done in step 1 is then $O(|V|)$.

In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly *twice*, once during EXPLORE(G, x) and once during EXPLORE(G, y). The overall time for step 2 is therefore $O(|E|)$.

Thus the depth-first search has a running time of $O(|V| + |E|)$.

Outline

Connectivity in undirected graphs

Connectivity in undirected graphs

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

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.

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.

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 (cont'd)

Connectivity in undirected graphs (cont'd)

Thus depth-first search is trivially adapted to check if a graph is connected.

Connectivity in undirected graphs (cont'd)

Thus depth-first search is trivially adapted to check if a graph is connected.

More generally, to assign each node v an integer $\text{CCNUM}[v]$ identifying the connected component to which it belongs.

Connectivity in undirected graphs (cont'd)

Thus depth-first search is trivially adapted to check if a graph is connected.

More generally, to assign each node v an integer $\text{CCNUM}[v]$ identifying the connected component to which it belongs.

All it takes is

$\underline{\text{PREVISIT}}(v)$

$\text{CCNUM}[v] = cc$

Connectivity in undirected graphs (cont'd)

Thus depth-first search is trivially adapted to check if a graph is connected.

More generally, to assign each node v an integer $\text{CCNUM}[v]$ identifying the connected component to which it belongs.

All it takes is

$\underline{\text{PREVISIT}}(v)$

$\text{CCNUM}[v] = cc$

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

Outline

Previsit and postvisit orderings

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

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);

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 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)

$\text{PRE}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

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).

$\underline{\text{PREVISIT}}(v)$

$\text{PRE}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

$\underline{\text{POSTVISIT}}(v)$

$\text{POST}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

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).

$\underline{\text{PREVISIT}}(v)$

$\text{PRE}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

$\underline{\text{POSTVISIT}}(v)$

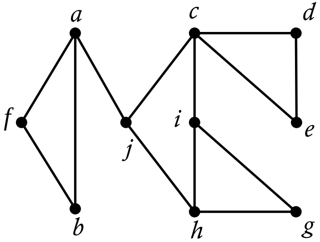
$\text{POST}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

Lemma: For any nodes u and v, the two intervals $[\text{PRE}(u), \text{POST}(u)]$ and $[\text{PRE}(u), \text{POST}(u)]$ are either disjoint or one is contained within the other.

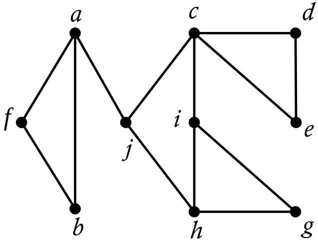
An executing example

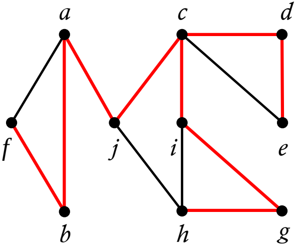
Assume we use alphabetical order to explore G:



An executing example

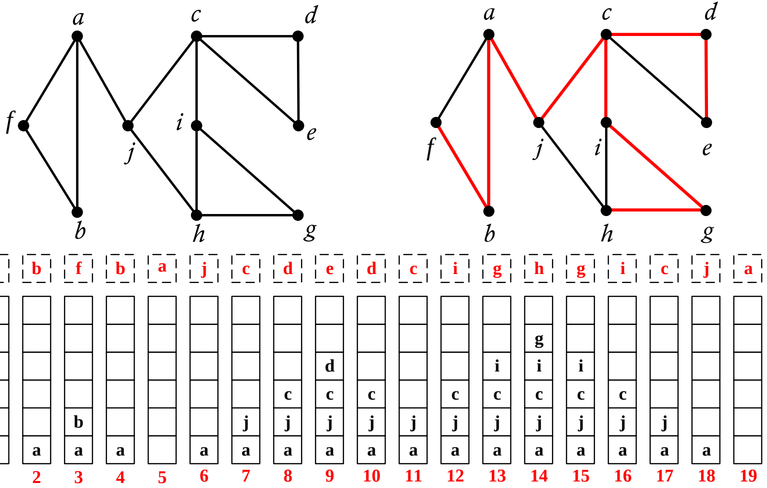
Assume we use alphabetical order to explore G:





An executing example

Assume we use alphabetical order to explore G:



Outline

Types of edges

DFS yields a **search tree/forests**.

Types of edges

DFS yields a **search tree/forests**.

- root.
- descendant and ancestor.
- parent and child.

Types of edges

DFS yields a **search tree/forests**.

- root.
- descendant and ancestor.
- parent and child.

- **Tree edges** are actually part of the DFS forest.

Types of edges

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.

Types of edges

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.
- **Backedges** lead to an ancestor in the DFS tree.

Types of edges

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.
- **Backedges** lead to an ancestor in the DFS tree.
- **Cross edges** lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

Types of edges (cont'd)

Types of edges (cont'd)

PRE/POST ordering for (u, v)	Edge type
$[_u \quad [_v \quad\quad]_v \quad]_u$	Tree/forward
$[_v \quad [_u \quad\quad]_u \quad]_v$	Back
$[_v \quad]_v \quad\quad [_u \quad]_u$	Cross

Outline

1. Depth-First Search in Undirected Graphs
   - Exploring Graphs
   - Connectivity in Undirected Graphs
   - Previsit and Postvisit Orderings
2. Depth-First Search in Directed Graphs
   - Types of Edges
   - Directed Acyclic Graphs
   - Strongly Connected Components
3. Breadth-First Search
   - Correctness and Efficiency

Directed acyclic graphs (DAG)

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.

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

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.

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

Proof: "$\Leftarrow$" One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.

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

Proof: "$\Leftarrow$" One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

"$\Rightarrow$" Conversely, if the graph has a cycle

$v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$,

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.
Lemma: A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Proof: "$\Leftarrow$" One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

"$\Rightarrow$" Conversely, if the graph has a cycle $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$, look at the first node v_i on this cycle to be discovered (the node with the lowest PRE number).

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.
Lemma: A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Proof: "$\Leftarrow$" One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

"$\Rightarrow$" Conversely, if the graph has a cycle $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$, look at the first node v_i on this cycle to be discovered (the node with the lowest PRE number).

All the other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree.

Directed acyclic graphs (DAG)

Definition: A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$.
Lemma: A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Proof: "$\Leftarrow$" One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

"$\Rightarrow$" Conversely, if the graph has a cycle $v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0$, look at the first node v_i on this cycle to be discovered (the node with the lowest PRE number).

All the other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree.

In particular, the edge $v_{i-1} \to v_i$ (or $v_k \to v_0$ if $i = 0$) is a back edge.

Directed acyclic graphs (cont'd)

Directed acyclic graphs (cont'd)

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

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.
Hence there is a linear-time algorithm for ordering the nodes of a dag.

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.
Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing POST numbers, the vertex with the smallest POST number comes last in this linearization, and it must be a **sink** – no outgoing edges.

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.
Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing POST numbers, 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.

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.
Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing POST numbers, 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.

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.
Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing POST numbers, 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:

Directed acyclic graphs (cont'd)

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

Lemma: In a dag, every edge leads to a vertex with a lower POST number.
Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing POST numbers, 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.

Outline

1. Depth-First Search in Undirected Graphs
   - Exploring Graphs
   - Connectivity in Undirected Graphs
   - Previsit and Postvisit Orderings
2. Depth-First Search in Directed Graphs
   - Types of Edges
   - Directed Acyclic Graphs
   - Strongly Connected Components
3. Breadth-First Search
   - Correctness and Efficiency

Defining connectivity for directed graphs

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.

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**.

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**.

Lemma: Every directed graph is a dag of its strongly connected components.

An efficient algorithm

An efficient algorithm

Lemma: If the EXPLORE subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

An efficient algorithm

Lemma: If the EXPLORE subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

Therefore, if we call explore on a node that lies somewhere in a sink strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component.

An efficient algorithm

Lemma: If the EXPLORE subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

Therefore, if we call explore on a node that lies somewhere in a sink strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component.

We have two problems:

(A) How do we find a node that we know for sure lies in a sink strongly connected component?

An efficient algorithm

Lemma: If the EXPLORE subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

Therefore, if we call explore on a node that lies somewhere in a sink strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component.

We have two problems:

(A) How do we find a node that we know for sure lies in a sink strongly connected component?

(B) How do we continue once this first component has been discovered?

An efficient algorithm (cont'd)

An efficient algorithm (cont'd)

Lemma: The node that receives the highest POST number in a depth-first search must lie in a *source strongly connected component*.

An efficient algorithm (cont'd)

Lemma: The node that receives the highest POST number in a depth-first search must lie in a *source strongly connected component*.

Lemma: If C and C' are strongly connected components, 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'.

An efficient algorithm (cont'd)

Lemma: The node that receives the highest POST number in a depth-first search must lie in a *source strongly connected component*.

Lemma: If C and C' are strongly connected components, 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 strongly connected components 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.

Solving problem A

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

G^R has exactly the same strongly connected components as G.

Solving problem A

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

G^R has exactly the same strongly connected components as G.

So, if we do a depth-first search of G^R, the node with the highest POST number will come from a source strongly connected component in G^R, which is to say a sink strongly connected component in G.

Solving problem B

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

Solving problem B

Once we have found the first strongly connected component and deleted it from the graph, the node with the highest post number among those remaining will belong to a sink strongly connected component 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 strongly connected component, and so on.

The linear-time algorithm

The linear-time algorithm

1. Run depth-first search on G^R.

The linear-time algorithm

1. Run depth-first search on G^R.
2. Run the undirected connected components algorithm on G, and during the depth-first search, process the vertices in decreasing order of their POST numbers from step 1.

Outline

The algorithm

The algorithm

Algorithm 7: $\textsc{BFS}(G, s)$

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

Output: For all vertices u reachable from s, $\textsc{dist}(u)$ is set to the distance from s to u

foreach $u \in V$ **do**
  $\textsc{dist}(u) = \infty$;
$\textsc{dist}(s) = 0$; $Q = [s]$ (queue containing just s);
while *Q is not empty* **do**
  $u = \textsc{eject}(Q)$;
  foreach *edge* $(u, v) \in E$ **do**
    if $\textsc{dist}(v) = \infty$ **then**
      $\textsc{inject}(Q, v)$; $\textsc{dist}(v) = \textsc{dist}(u) + 1$;

Correctness and efficiency

Correctness and efficiency

Lemma: For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

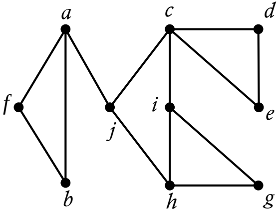
Correctness and efficiency

Lemma: For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

Lemma: BFS has a running time of $O(|V| + |E|)$.

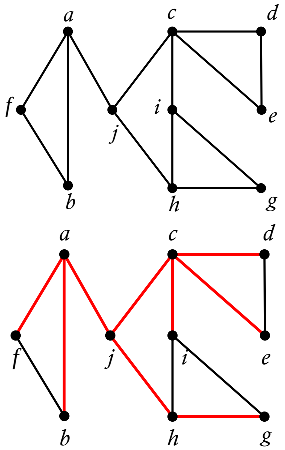
An executing example

Assume we use alphabetical order to explore G:



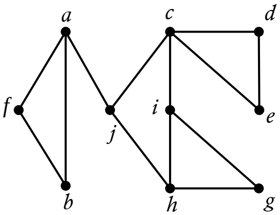
An executing example

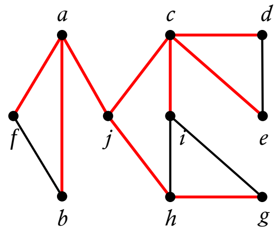
Assume we use alphabetical order to explore G:



An executing example

Assume we use alphabetical order to explore G:





1		a					
2	a	b	f	j			
3	b	f	j				
4	f	j					
5	j	c	h				
6	c	h	d	e	i		
7	h	d	e	i	g		
8	d	e	i	g			
9	e	i	g				
10	i	g					
11	g						