Introduction to graph theory and algorithms

Jean-Yves L'Excellent and Bora Uçar

GRAAL, LIP, ENS Lyon, France

CR-07: Sparse Matrix Computations, September 2010

http://graal.ens-lyon.fr/~bucar/CR07/

Outline

1. Definitions and some problems
2. Basic algorithms
   - Breadth-first search
   - Depth-first search
   - Topological sort
   - Strongly connected components
3. Questions

Graph notations and definitions

A graph $G = (V, E)$ consists of a finite set V, called the vertex set and a finite, binary relation E on V, called the edge set.

Three standard graph models

Undirected graph: The edges are unordered pair of vertices, i.e., $\{u, v\} \in E$ for some $u, v \in V$.

Directed graph: The edges are ordered pair of vertices, that is, (u, v) and (v, u) are two different edges.

Bipartite graph: $G = (U \cup V, E)$ consists of two disjoint vertex sets U and V such that for each edge $(u, v) \in E$, $u \in U$ and $v \in V$.

An ordering or labelling of $G = (V, E)$ having n vertices, i.e., $|V| = n$, is a mapping of V onto $1, 2, \ldots, n$.

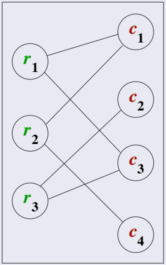
Matrices and graphs: Rectangular matrices

The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

Rectangular matrices

$$A = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{cccc} \times & & \times & \\ \times & & & \times \\ & \times & \times & \end{array} \right) \end{array}$$

Bipartite graph



The set of rows corresponds to one of the vertex set R, the set of columns corresponds to the other vertex set C such that for each $a_{ij} \neq 0$, (r_i, c_j) is an edge.

Matrices and graphs: Square unsymmetric pattern

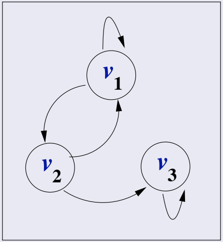
The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

Square unsymmetric pattern matrices

$$A = \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} \times & \times & \\ \times & & \times \\ & & \times \end{pmatrix} \end{array}$$

Graph models

- Bipartite graph as before.
- Directed graph



The set of rows/cols corresponds the vertex set V such that for each $a_{ij} \neq 0$, (v_i, v_j) is an edge. Transposed view possible too, i.e., the edge (v_i, v_j) directed from column i to row j. Usually self-loops are omitted.

Matrices and graphs: Square unsymmetric pattern

A special subclass

Directed acyclic graphs (DAG):
A directed graphs with no loops (maybe except for self-loops).

DAGs

We can sort the vertices such that if (u, v) is an edge, then u appears before v in the ordering.

Question: What kind of matrices have a DAG?

Matrices and graphs: Symmetric pattern

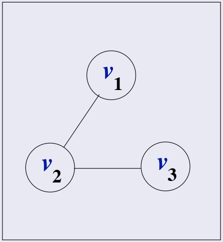
The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

Square symmetric pattern matrices

$$A = \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} & \times & \\ \times & \times & \times \\ & \times & \times \end{pmatrix} \end{array}$$

Graph models

- Bipartite and directed graphs as before.
- Undirected graph



The set of rows/cols corresponds the vertex set V such that for each $a_{ij}, a_{ji} \neq 0$, $\{v_i, v_j\}$ is an edge. No self-loops; usually the main diagonal is assumed to be zero-free.

Definitions: Edges, degrees, and paths

Many definitions for directed and undirected graphs are the same. We will use (u, v) to refer to an edge of an undirected or directed graph to avoid repeated definitions.

- An edge (u, v) is said to incident on the vertices u and v.
- For any vertex u, the set of vertices in $\mathrm{adj}(u) = \{v : (u, v) \in E\}$ are called the neighbors of u. The vertices in $\mathrm{adj}(u)$ are said to be adjacent to u.
- The degree of a vertex is the number of edges incident on it.
- A path p of length k is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ where $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, k$. The two end points v_0 and v_k are said to be connected by the path p, and the vertex v_k is said to be reachable from v_0.

Definitions: Components

- An undirected graph is said to be connected if every pair of vertices is connected by a path.
- The connected components of an undirected graph are the equivalence classes of vertices under the "is reachable" from relation.
- A directed graph is said to be strongly connected if every pair of vertices are reachable from each other.
- The strongly connected components of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation.

Definitions: Trees and spanning trees

A tree is a connected, acyclic, undirected graph. If an undirected graph is acyclic but disconnected, then it is a forest.

Properties of trees

- Any two vertices are connected by a unique path.
- $|E| = |V| - 1$

A rooted tree is a tree with a distinguished vertex r, called the root.

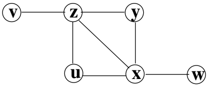
There is a unique path from the root r to every other vertex v. Any vertex y in that path is called an ancestor of v. If y is an ancestor of v, then v is a descendant of y.

The subtree rooted at v is the tree induced by the descendants of v, rooted at v.

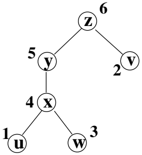
A spanning tree of a connected graph $G = (V, E)$ is a tree $T = (V, F)$, such that $F \subseteq E$.

Ordering of the vertices of a rooted tree

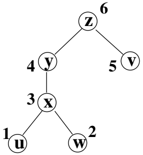
- A topological ordering of a rooted tree is an ordering that numbers children vertices before their parent.
- A postorder is a topological ordering which numbers the vertices in any subtree consecutively.



Connected graph G



Rooted spanning tree
with topological ordering



Rooted spanning tree
with postordering

Postordering the vertices of a rooted tree – I

The following recursive algorithm will do the job:

$[porder] = \textsc{PostOrder}(T, r)$

  for each child c of r **do**

    $porder \leftarrow [porder, \textsc{PostOrder}(T, c)]$

  $porder \leftarrow [porder, r]$

We need to run the algorithm for each root r when T is a forest.

Usually recursive algorithms are avoided, as for a tree with large number of vertices can cause stack overflow.

Postordering the vertices of a rooted tree – II

$[porder]$=$\textsc{PostOrder}(T, r)$
  $porder \leftarrow [\cdot]$
  $seen(v) \leftarrow$ **False** for all $v \in T$
  $seen(r) \leftarrow$ **True**
  $\textsc{Push}(S, r)$
  while $\textsc{NotEmpty}(S)$ **do**
    $v \leftarrow \textsc{Pop}(S)$
    if $\exists$ a child c of v with $seen(c) =$ **False then**
      $seen(c) \leftarrow$ **True**
      $\textsc{Push}(S, c)$
    else
      $porder \leftarrow [porder, v]$

Again, have to run for each root, if T is a forest.

Both algorithms run in $\mathcal{O}(n)$ time for a tree with n nodes.

Permutation matrices

A permutation matrix is a square $(0,1)$-matrix where each row and column has a single 1.

If P is a permutation matrix, $PP^T = I$, i.e., it is an orthogonal matrix. Let,

$$A = \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} \times & \times & \\ \times & & \times \\ & & \times \end{pmatrix} \end{array}$$

and suppose we want to permute columns as $[2,1,3]$. Define $p_{2,1} = 1$, $p_{1,2} = 1$, $p_{3,3} = 1$, and $B = AP$ (if column j to be at position i, set $p_{ji} = 1$)

$$B = \begin{array}{c c} & \begin{array}{ccc} 2 & 1 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} \times & \times & \\ & \times & \times \\ & & \times \end{pmatrix} \end{array} = \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} \times & \times & \\ \times & & \times \\ & & \times \end{pmatrix} \end{array} \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} & 1 & \\ 1 & & \\ & & 1 \end{pmatrix} \end{array}$$

Matching in bipartite graphs and permutations

A matching in a graph is a set of edges no two of which share a common vertex. We will be mostly dealing with matchings in bipartite graphs.

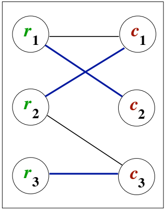
In matrix terms, a matching in the bipartite graph of a matrix corresponds to a set of nonzero entries no two of which are in the same row or column.

A vertex is said to be matched if there is an edge in the matching incident on the vertex, and to be unmatched otherwise. In a perfect matching, all vertices are matched.

The cardinality of a matching is the number of edges in it. A maximum cardinality matching or a maximum matching is a matching of maximum cardinality. Solvable in polynomial time.

Matching in bipartite graphs and permutations

Given a square matrix whose bipartite graph has a perfect matching, such a matching can be used to permute the matrix such that the matching entries are along the main diagonal.



$$\begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{ccc} \times & \times & \\ \times & & \times \\ & & \times \end{array}\right) \end{array}$$

$$\begin{array}{c c} & \begin{array}{ccc} 2 & 1 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{ccc} \times & \times & \\ & \times & \times \\ & & \times \end{array}\right) \end{array} = \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{ccc} \times & \times & \\ \times & & \times \\ & & \times \end{array}\right) \end{array} \begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{ccc} & 1 & \\ 1 & & \\ & & 1 \end{array}\right) \end{array}$$

Definitions: Reducibility

Reducible matrix: An $n \times n$ square matrix is reducible if there exists an $n \times n$ permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},$$

where A_{11} is an $r \times r$ submatrix, A_{22} is an $(n-r) \times (n-r)$ submatrix, where $1 \leq r < n$.

Irreducible matrix: There is no such a permutation matrix.

Theorem: An $n \times n$ square matrix is irreducible iff its directed graph is strongly connected.

Proof: Follows by definition.

Definitions: Fully indecomposability

Fully indecomposable matrix: There is no permutation matrices P and Q such that

$$PAQ = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},$$

with the same condition on the blocks and their sizes as above.

Theorem: An $n \times n$ square matrix A is fully indecomposable iff for some permutation matrix P, the matrix PA is irreducible and has a zero-free main diagonal.

Proof: We will come later in the semester to the "if" part.

Only if part (by contradiction): Let $B = PA$ be an irreducible matrix with zero-free main diagonal. B is fully indecomposable iff A is (why?). Therefore we may assume that A is irreducible and has a zero-free diagonal. Suppose, for the sake of contradiction, A is not fully indecomposable.

Fully indecomposable matrices

Fully indecomposable matrix

There is no permutation matrices P and Q such that

$$PAQ = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},$$

with the same condition on the blocks and their sizes as above.

Proof cont.: Let P_1AQ_1 be of the form above with A_{11} of size $r \times r$. We may write $P_1AQ_1 = A'Q'$, where $A' = P_1AP_1{}^T$ with zero-free diagonal (why?), and $Q' = P_1Q_1$ is a permutation matrix which has to permute (why?) the first r columns among themselves, and similarly the last $n-r$ columns among themselves. Hence, A' is in the above form, and A is reducible: contradiction. $\square$

Definitions: Cliques and independent sets

Clique

In an undirected graph $G = (V, E)$, a set of vertices $S \subseteq V$ is a clique if for all $s, t \in S$, we have $(s, t) \in E$.

Maximum clique: A clique of maximum cardinality (finding a maximum clique in an undirected graph is NP-complete).

Maximal clique: A clique is a maximal clique, if it is not contained in another clique.

In a symmetric matrix A, a clique corresponds to a subset of rows R and the corresponding columns such that the matrix $A(R, R)$ is full.

Independent set

A set of vertices is an independent set if none of the vertices are adjacent to each other. Can we find the largest one in polynomial time?

In a symmetric matrix A, an independent set corresponds to a subset of rows R and the corresponding columns such that the matrix $A(R, R)$ is either zero, or diagonal.

Definitions: More on cliques

Clique: In an undirected graph $G = (V, E)$, a set of vertices $S \subseteq V$ is a clique if for all $s, t \in S$, we have $(s, t) \in E$.

In a symmetric matrix A corresponds to a subset of rows R and the corresponding columns such that the matrix $A(R, R)$ is full.

Cliques in bipartite graphs: Bi-cliques

In a bipartite graph $G = (U \cup V, E)$, a pair of sets $\langle R, C \rangle$ where $R \subseteq U$ and $C \subseteq V$ is a bi-clique if for all $a \in R$ and $b \in C$, we have $(a, b) \in E$.

In a matrix A, corresponds to a subset of rows R and a subset of columns C such that the matrix $A(R, C)$ is full.

The maximum node bi-clique problem asks for a bi-clique of maximum size (e.g., $|R| + |C|$), and it is polynomial time solvable, whereas maximum edge bi-clique problem (e.g., asks for a maximum $|R| \times |C|$) is NP-complete.

Definitions: Hypergraphs

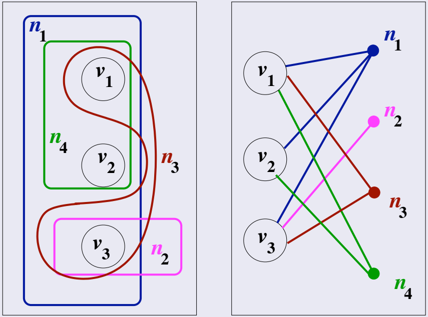
Hypergraph: A hypergraph $H = (V, N)$ consists of a finite set V called the vertex set and a set of non-empty subsets of vertices N called the hyperedge set or the net set. A generalization of graphs.

For a matrix A, define a hypergraph whose vertices correspond to the rows and whose nets correspond to the columns such that vertex v_i is in net n_j iff $a_{ij} \neq 0$ (the column-net model).

A sample matrix

$$\begin{array}{c|c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} \times & & \times & \times \\ \times & & & \times \\ \times & \times & \times & \end{pmatrix} \end{array}$$

The column-net hypergraph model



Basic graph algorithms

Searching a graph: Systematically following the edges of the graph so as to visit all the vertices.

- Breadth-first search,
- Depth-first search.

Topological sort (of a directed acyclic graph): It is a linear ordering of all the vertices such that if (u, v) directed is an edge, then u appears before v in the ordering.

Strongly connected components (of a directed graph; why?): Recall that a strongly connected component is a maximal set of vertices for which every pair its vertices are reachable. We want to find them all.

We will use some of the course notes by Cevdet Aykanat (`http://www.cs.bilkent.edu.tr/~aykanat/teaching.html`)

Breadth-first search: Idea

Graph $G = (V, E)$, directed or undirected with adjacency list repres.

GOAL: Systematically explores edges of G to

- discover every vertex reachable from the source vertex s
- compute the shortest path distance of every vertex from the source vertex s
- produce a breadth-first tree (BFT) G_Π with root s
  - BFT contains all vertices reachable from s
  - the unique path from any vertex v to s in G_Π constitutes a shortest path from s to v in G

IDEA: Expanding frontier across the breadth -greedy-

- propagate a wave 1 edge-distance at a time
- using a FIFO queue: O(1) time to update pointers to both ends

Breadth-first search: Key components

Maintains the following fields for each $u \in V$

- color[u]: color of u
  - WHITE : not discovered yet
  - GRAY : discovered and to be or being processed
  - BLACK: discovered and processed
- $\Pi[u]$: parent of u (NIL of $u = s$ or u is not discovered yet)
- $d[u]$: distance of u from s

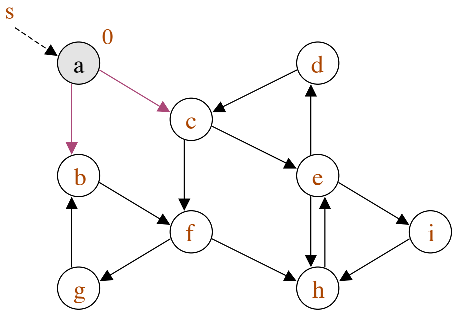
Processing a vertex = scanning its adjacency list

Breadth-first search: Algorithm

BFS(G, s)
  for each $u \in V - \{s\}$ do
    color[u] $\leftarrow$ WHITE
    $\Pi[u] \leftarrow$ NIL; $d[u] \leftarrow \infty$
  color[s] $\leftarrow$ GRAY
  $\Pi[s] \leftarrow$ NIL; $d[s] \leftarrow 0$
  $Q \leftarrow \{s\}$
  while $Q \neq \emptyset$ do
    $u \leftarrow$ head[Q]
    for each v in Adj[u] do
      if color[v] = WHITE then
        color[v] $\leftarrow$ GRAY
        $\Pi[v] \leftarrow u$
        $d[v] \leftarrow d[u] + 1$
        ENQUEUE(Q, v)
    DEQUEUE(Q)
    color[u] $\leftarrow$ BLACK

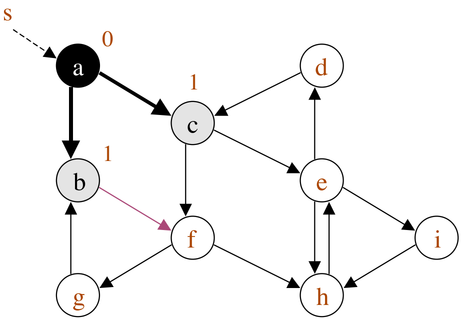
Breadth-first search: Example

Sample Graph:



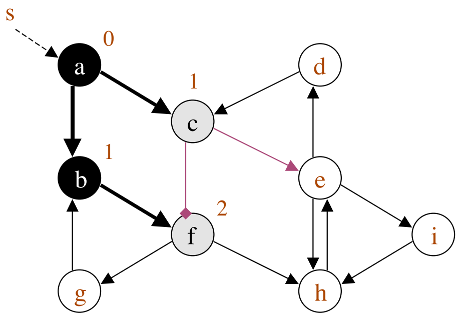
FIFO queue Q	just after processing vertex
⟨a⟩	-

Breadth-first search: Example



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a

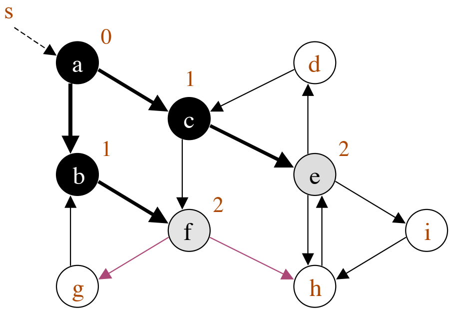
Breadth-first search: Example



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b

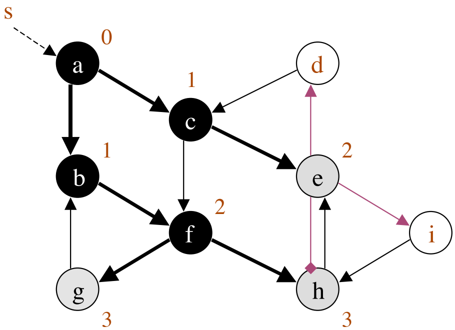
↑

Breadth-first search: Example



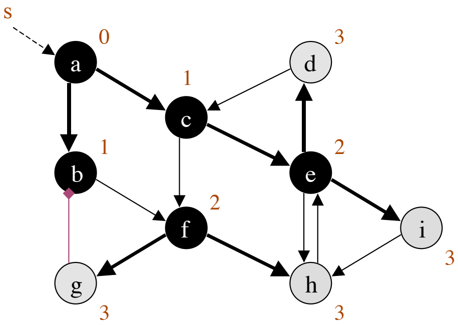
FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c

Breadth-first search: Example



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f

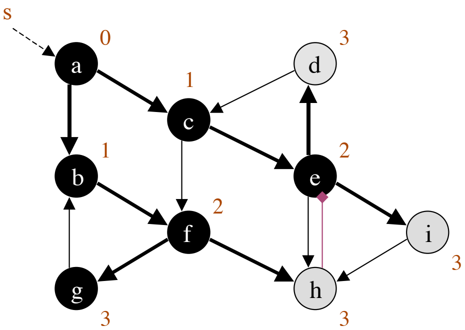
Breadth-first search: Example



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	e

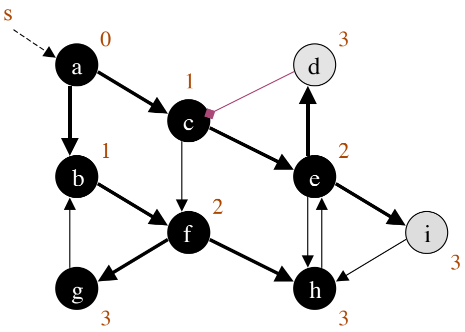
all distances are filled in after processing e

Breadth-first search: Example



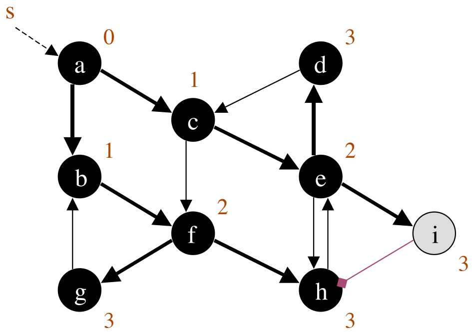
FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	g

Breadth-first search: Example



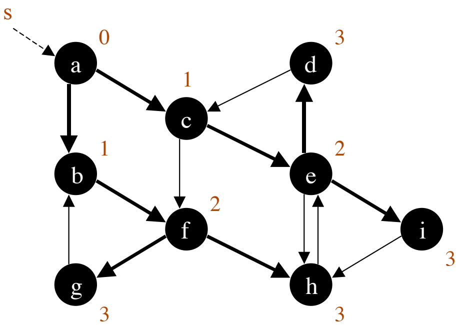
FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	h

Breadth-first search: Example



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	d

Breadth-first search: Example



FIFO queue Q	just after processing vertex
⟨a⟩	-
⟨a,b,c⟩	a
⟨a,b,c,f⟩	b
⟨a,b,c,f,e⟩	c
⟨a,b,c,f,e,g,h⟩	f
⟨a,b,c,f,e,g,h,d,i⟩	i

algorithm terminates: all vertices are processed

Breadth-first search: Analysis

Running time: $O(V+E)$ = considered linear time in graphs

- initialization: $\Theta(V)$
- queue operations: $O(V)$
  - each vertex enqueued and dequeued at most once
  - both enqueue and dequeue operations take $O(1)$ time
- processing gray vertices: $O(E)$
  - each vertex is processed at most once and

$$\sum_{u \in V} |Adj[u]| = \Theta(E)$$

Breadth-first search: The paths to the root

BFS(G, s) , where $V_\Pi = \{v \in V: \Pi[v] \neq \text{NIL}\} \cup \{s\}$ and
$E_\Pi = \{(\Pi[v], v) \in E: v \in V_\Pi - \{s\}\}$
is a breadth-first tree such that

- V_Π consists of all vertices in V that are reachable from s
- $\forall v \in V_\Pi$, unique path p(v, s) in G_Π constitutes a sp(s, v) in G

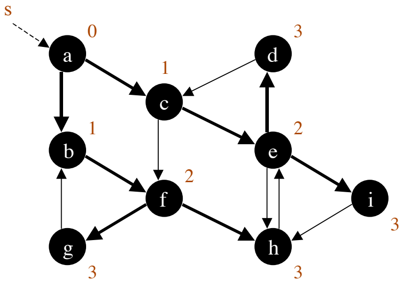
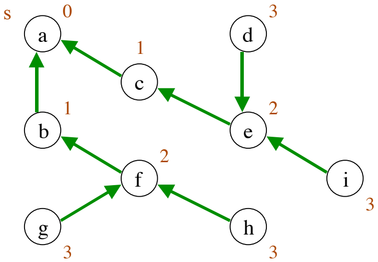
PRINT-PATH(G, s, v)

> Prints out vertices on a $s \rightarrow v$ shortest path

if v = s **then** print s
else if $\Pi[v]$ = NIL **then**
print no "$s \rightarrow v$ path"
else
PRINT-PATH(G, s, $\Pi[v]$)
print v

Breadth-first search: The BFS tree

Breadth-First Tree Generated by BFS

BFS(G,a) terminated	BFT generated by BFS(G,a)
	

Depth-first search: Idea

- Graph G=(V,E) directed or undirected
- Adjacency list representation
- Goal: Systematically explore every vertex and every edge
- Idea: search deeper whenever possible
  - Using a LIFO queue (Stack; FIFO queue used in BFS)

Depth-first search: Key components

- Maintains several fields for each $v \in V$
- Like BFS, colors the vertices to indicate their states. Each vertex is
  - Initially white,
  - grayed when discovered,
  - blackened when finished
- Like BFS, records discovery of a white v during scanning Adj[u] by $\pi[v] \leftarrow u$

Depth-first search: Key components

- Unlike BFS, predecessor graph G_π produced by DFS forms spanning forest
- $G_\pi=(V,E_\pi)$ where

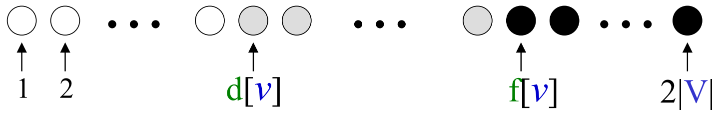
$$E_\pi=\{(\pi[v],v): v\in V \text{ and } \pi[v]\neq \text{NIL}\}$$

- G_π= depth-first forest (DFF) is composed of disjoint depth-first trees (DFTs)

Depth-first search: Key components

- DFS also timestamps each vertex with two timestamps
- $d[v]$: records when v is first discovered and grayed
- $f[v]$: records when v is finished and blackened
- Since there is only one discovery event and finishing event for each vertex we have $1 \leq d[v] < f[v] \leq 2|V|$

Time axis for the color of a vertex



Depth-first search: Algorithm

DFS(G)

  for each $u \in V$ **do**
    color[u] ← *white*
    $\pi[u] \leftarrow$ NIL
  time ← 0
  for each $u \in V$ **do**
    if color[u] = *white* **then**
      DFS-VISIT(G, u)

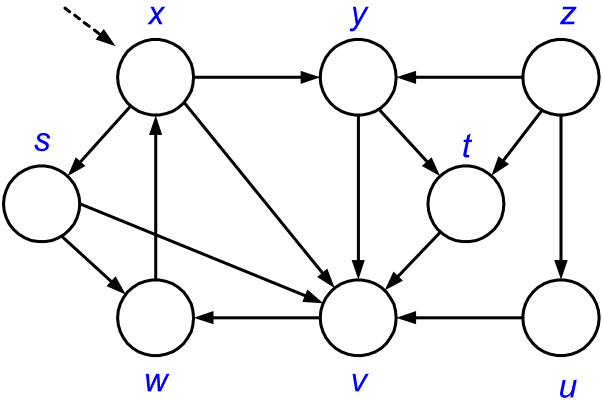
DFS-VISIT(G, u)

  color[u] ← *gray*
  d[u] ← *time* ← *time* +1
  for each $v \in$ Adj[u] **do**
    if color[v] = *white* **then**
      $\pi[v] \leftarrow u$
      DFS-VISIT(G, v)
  color[u] ← *black*
  f[u] ← *time* ← *time* +1

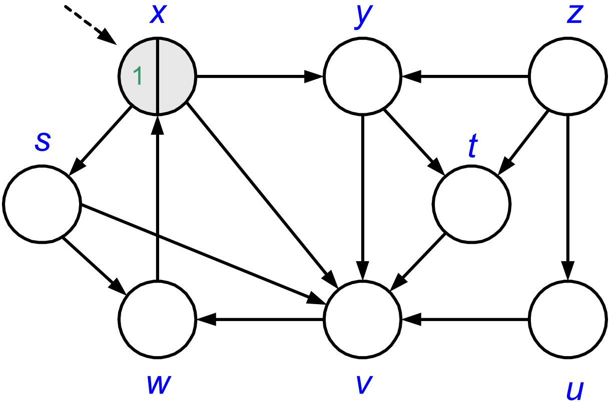
Depth-first search: Analysis

- Running time: $\Theta(V+E)$
- Initialization loop in **DFS** : $\Theta(V)$
- Main loop in **DFS**: $\Theta(V)$ exclusive of time to execute calls to **DFS-VISIT**
- **DFS-VISIT** is called exactly once for each $v \in V$ since
  - **DFS-VISIT** is invoked only on white vertices and
  - **DFS-VISIT(G, u)** immediately colors u as gray
- For loop of **DFS-VISIT(G, u)** is executed $|\mathrm{Adj}[u]|$ time
- Since $\Sigma\ |\mathrm{Adj}[u]| = E$, total cost of executing loop of **DFS-VISIT** is $\Theta(E)$

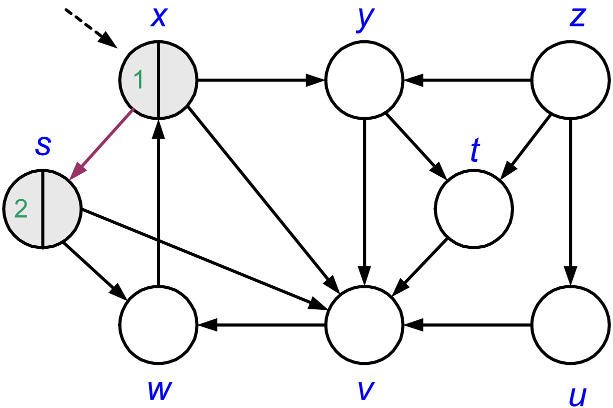
Depth-first search: Example



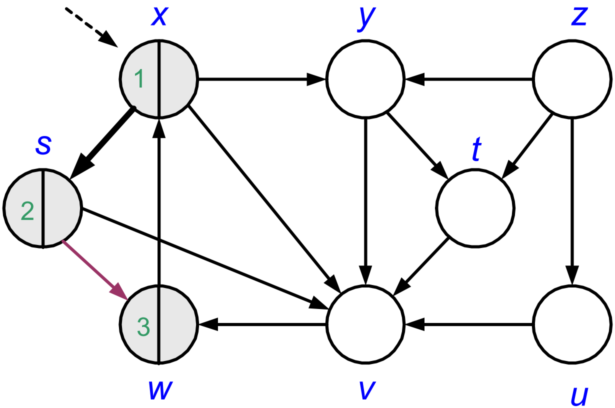
Depth-first search: Example



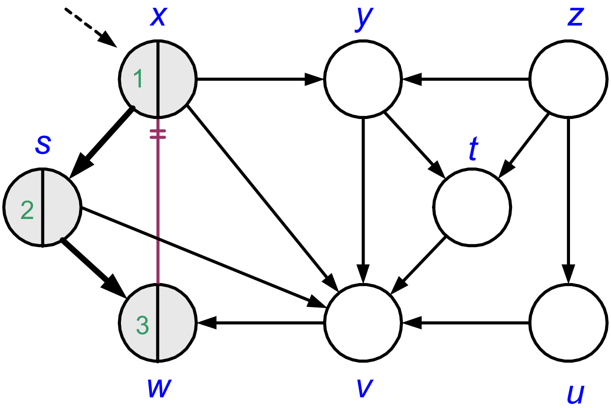
Depth-first search: Example



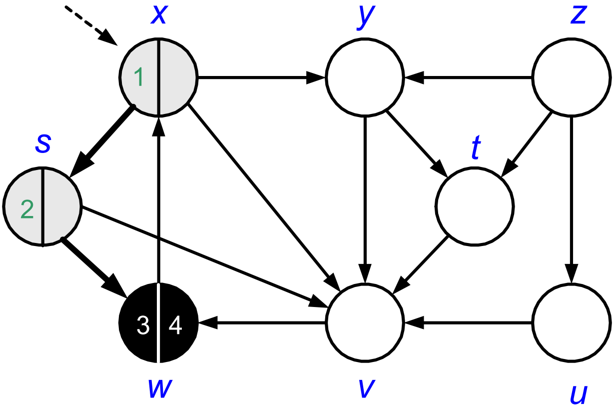
Depth-first search: Example



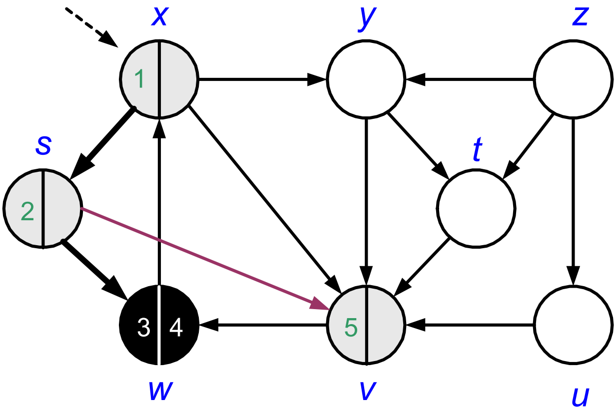
Depth-first search: Example



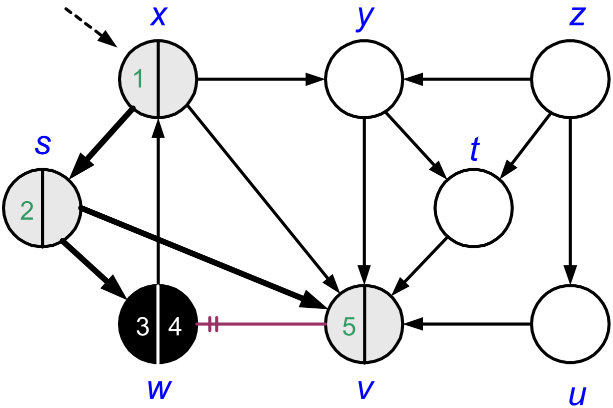
Depth-first search: Example



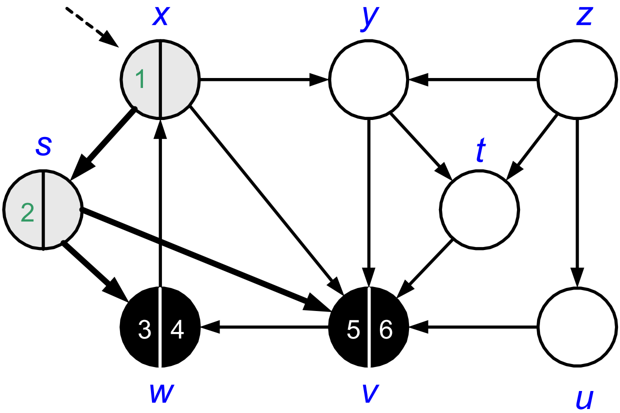
Depth-first search: Example



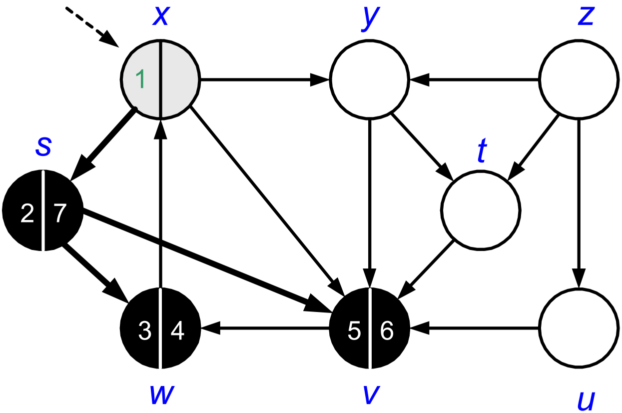
Depth-first search: Example



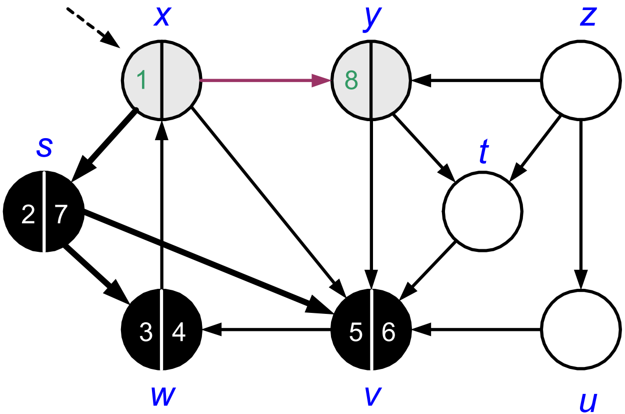
Depth-first search: Example



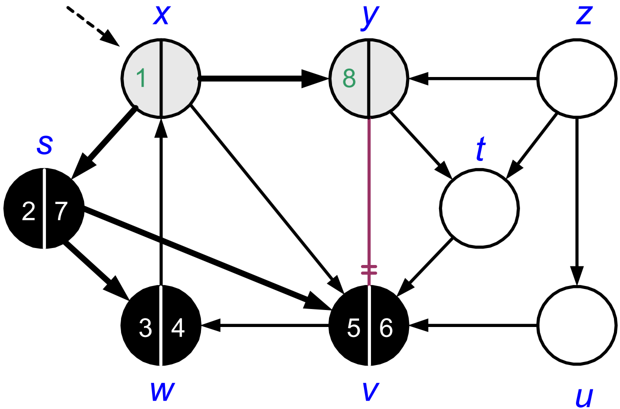
Depth-first search: Example



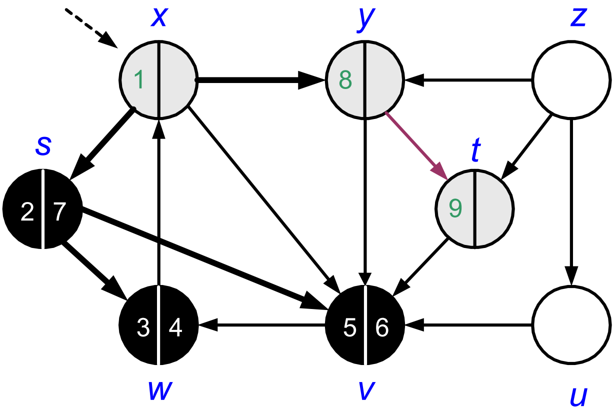
Depth-first search: Example



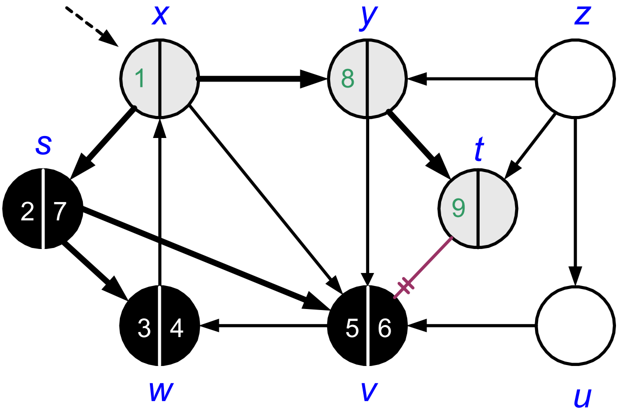
Depth-first search: Example



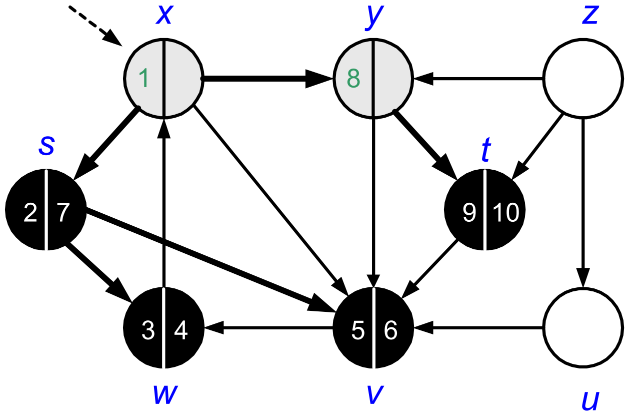
Depth-first search: Example



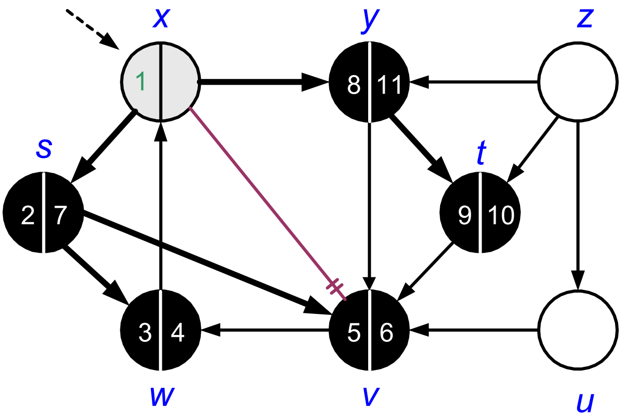
Depth-first search: Example



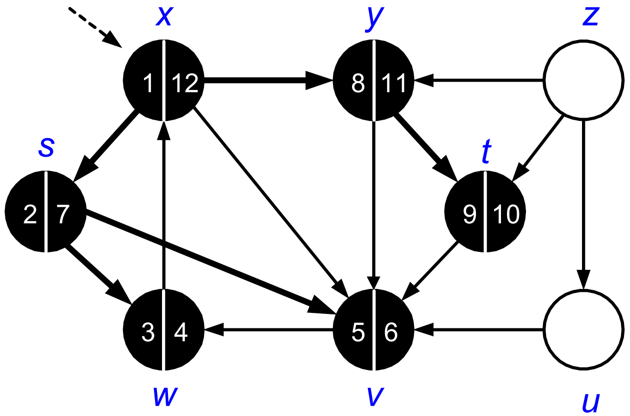
Depth-first search: Example



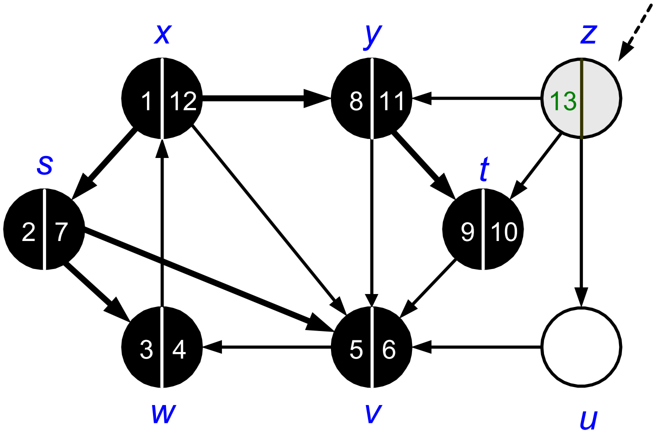
Depth-first search: Example



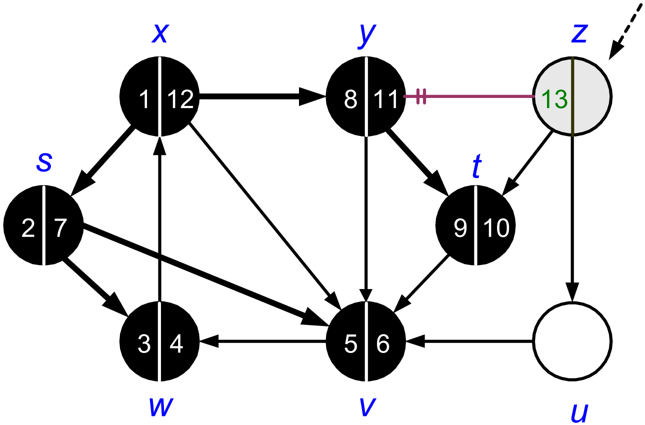
Depth-first search: Example



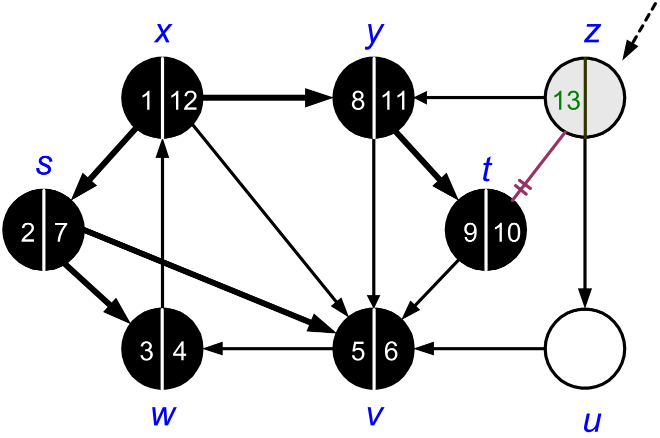
Depth-first search: Example



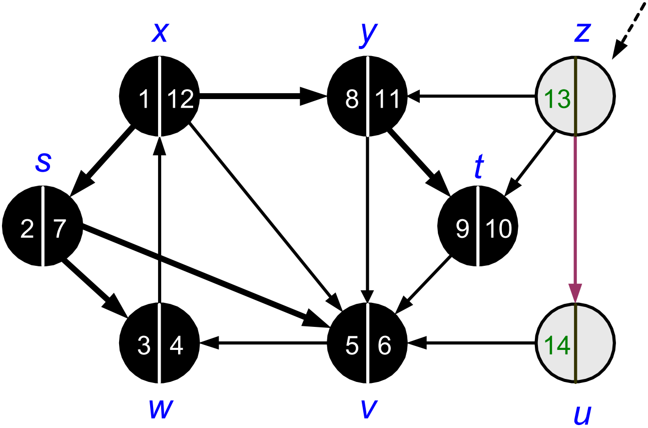
Depth-first search: Example



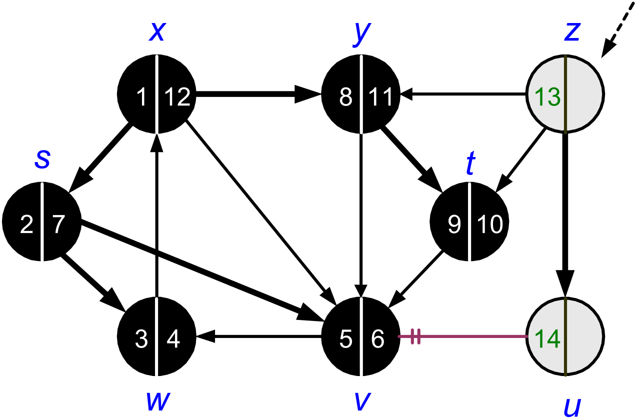
Depth-first search: Example



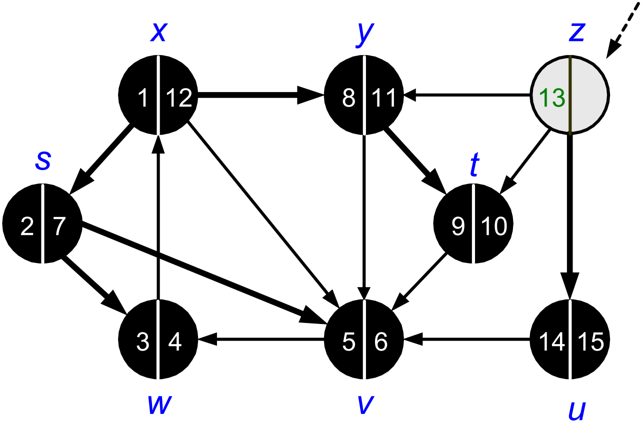
Depth-first search: Example



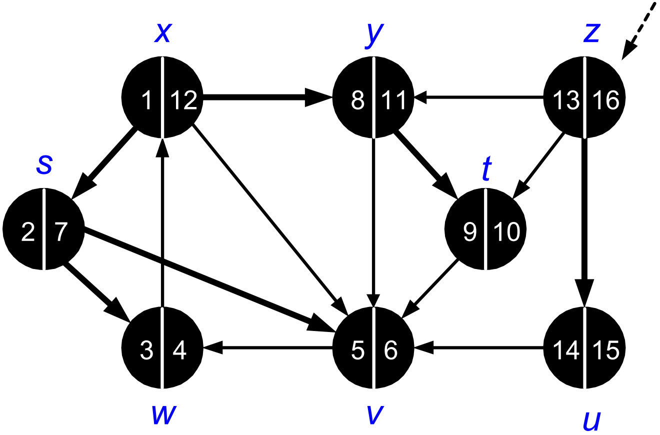
Depth-first search: Example



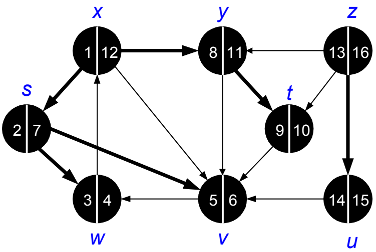
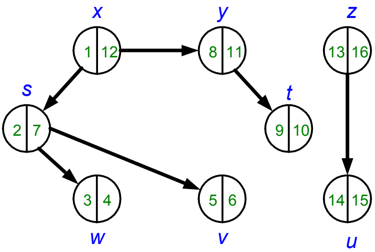
Depth-first search: Example



Depth-first search: Example



Depth-first search: DFT and DFF

DFS(G) terminated	Depth-first forest (DFF)
	

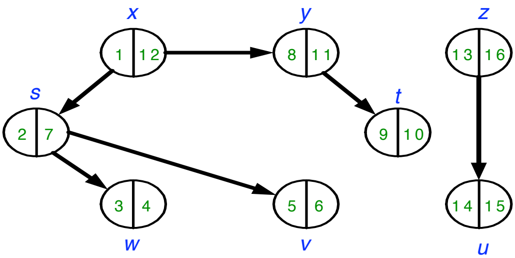
Depth-first search: Parenthesis theorem

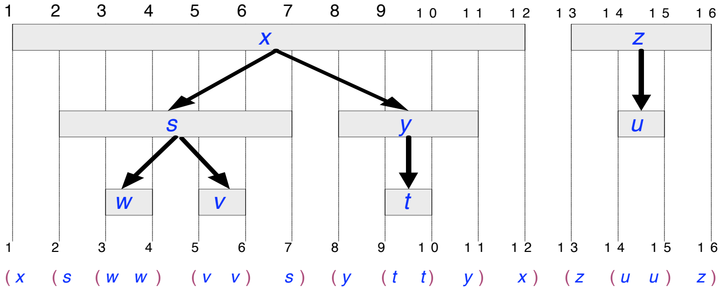
Thm: In any DFS of $G=(V,E)$, let $\text{int}[v] = [d[v], f[v]]$ then exactly one of the following holds for any u and $v \in V$

- $\text{int}[u]$ and $\text{int}[v]$ are entirely disjoint
- $\text{int}[v]$ is entirely contained in $\text{int}[u]$ and v is a descendant of u in a DFT
- $\text{int}[u]$ is entirely contained in $\text{int}[v]$ and u is a descendant of v in a DFT

Depth-first search: Parenthesis theorem

Parenthesis Theorem





Depth-first search: Edge classification

Tree Edge: discover a new (WHITE) vertex
▷GRAY to WHITE◁

Back Edge: from a descendent to an ancestor in DFT
▷GRAY to GRAY◁

Forward Edge: from ancestor to descendent in DFT
▷GRAY to BLACK◁

Cross Edge: remaining edges (btwn trees and subtrees)
▷GRAY to BLACK◁

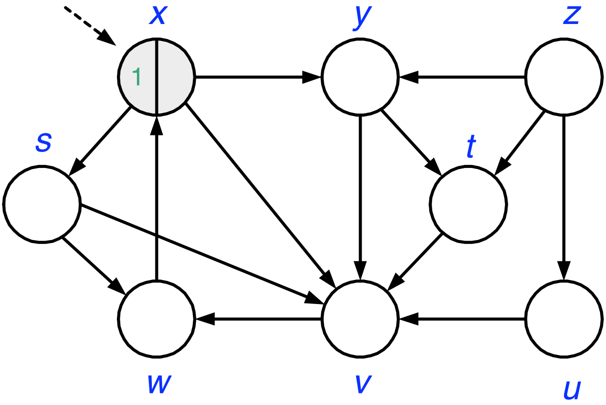
Note: ancestor/descendent is wrt Tree Edges

Depth-first search: Edge classification

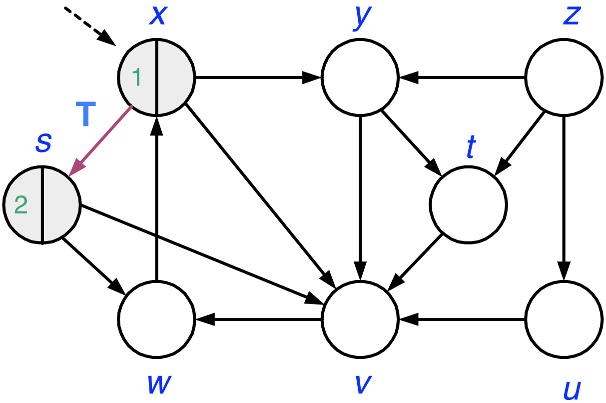
- How to decide which GRAY to BLACK edges are forward, which are cross

  Let BLACK vertex $v \in \mathrm{Adj}[u]$ is encountered while processing GRAY vertex u
  - (u,v) is a forward edge if $d[u] < d[v]$
  - (u,v) is a cross edge if $d[u] > d[v]$

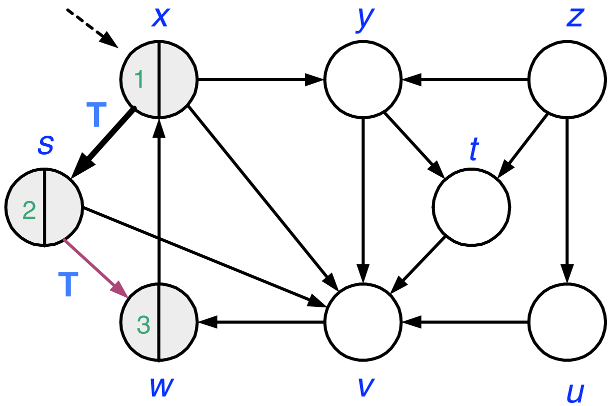
Depth-first search: Edge classification example



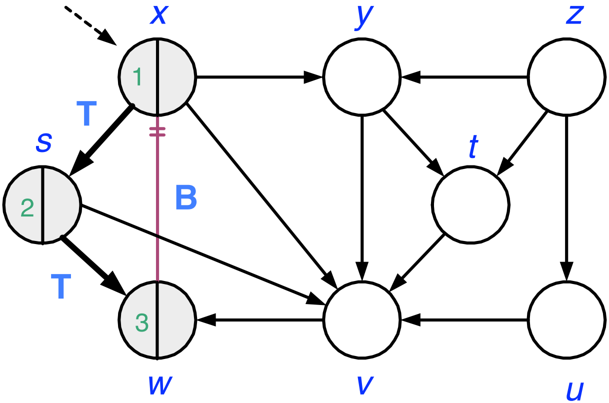
Depth-first search: Edge classification example



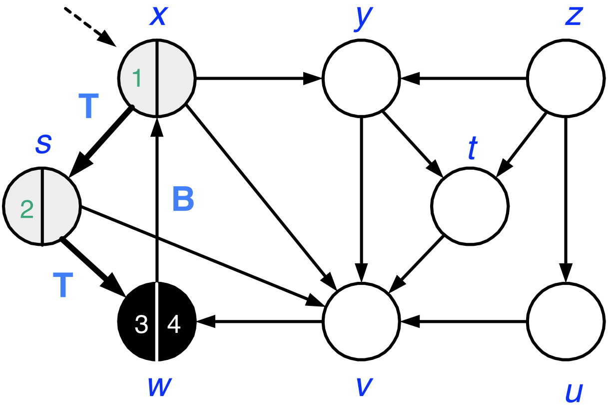
Depth-first search: Edge classification example



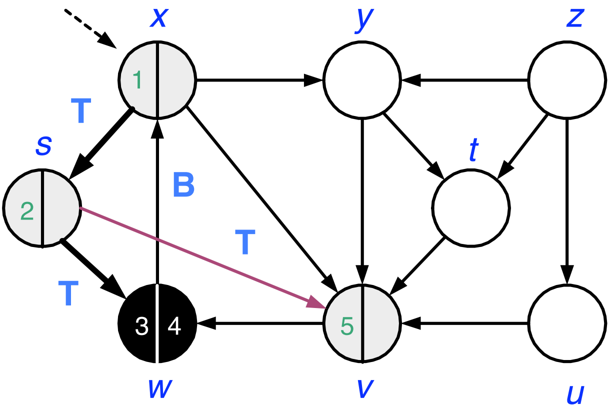
Depth-first search: Edge classification example



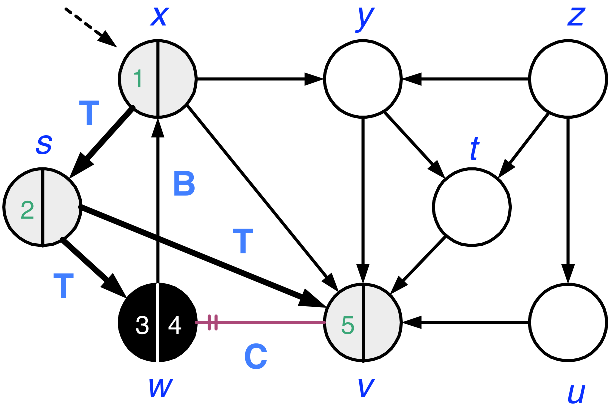
Depth-first search: Edge classification example



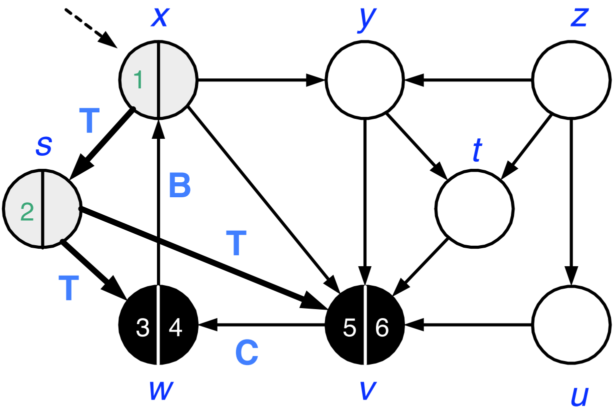
Depth-first search: Edge classification example



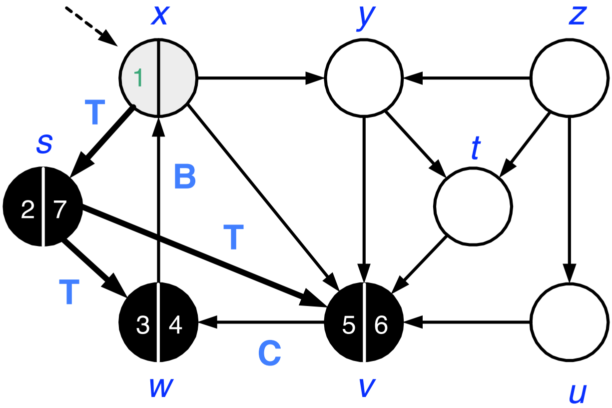
Depth-first search: Edge classification example



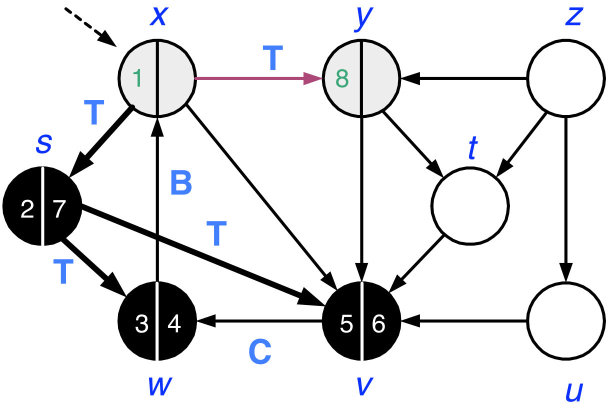
Depth-first search: Edge classification example



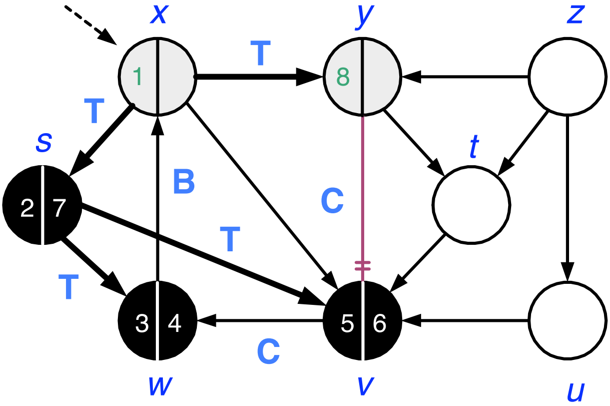
Depth-first search: Edge classification example



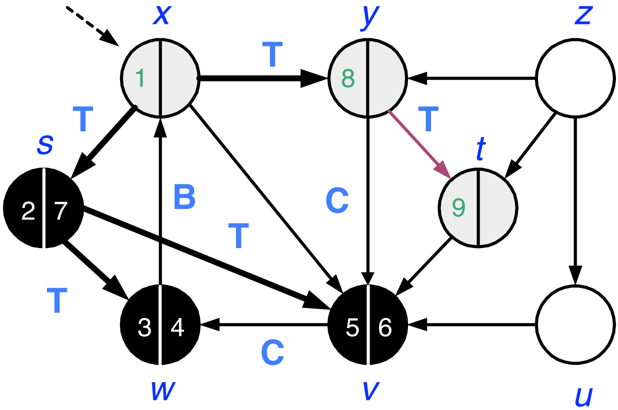
Depth-first search: Edge classification example



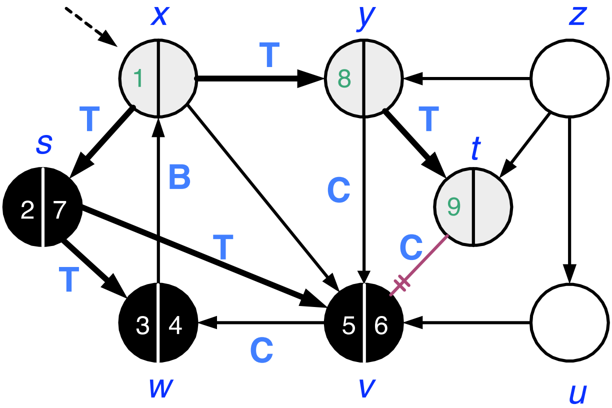
Depth-first search: Edge classification example



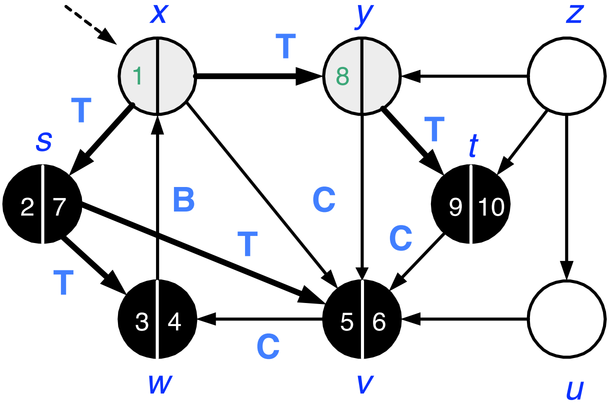
Depth-first search: Edge classification example



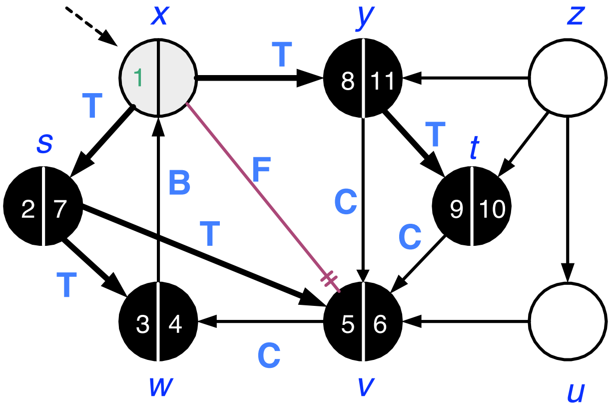
Depth-first search: Edge classification example



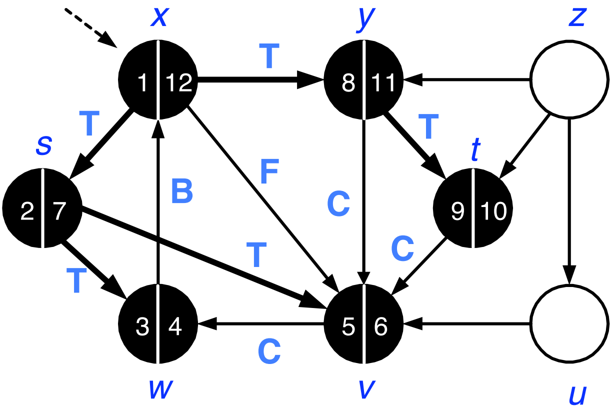
Depth-first search: Edge classification example



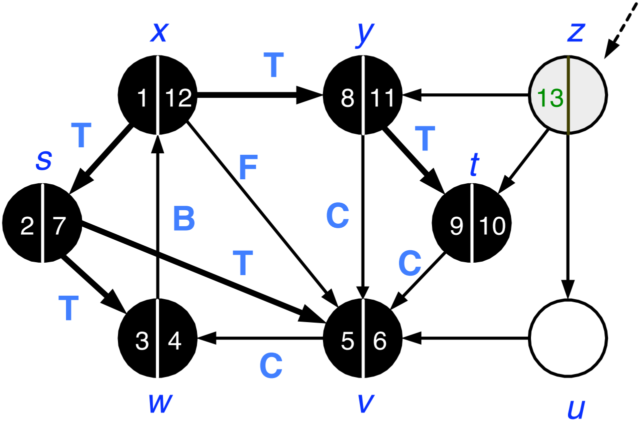
Depth-first search: Edge classification example



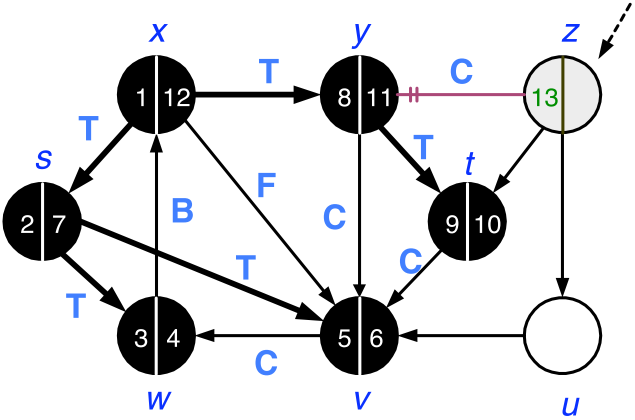
Depth-first search: Edge classification example



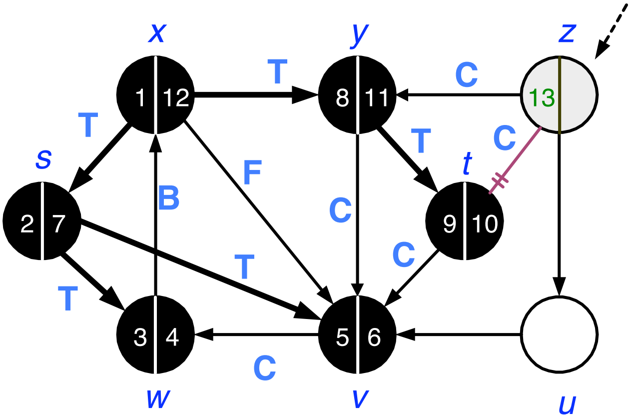
Depth-first search: Edge classification example



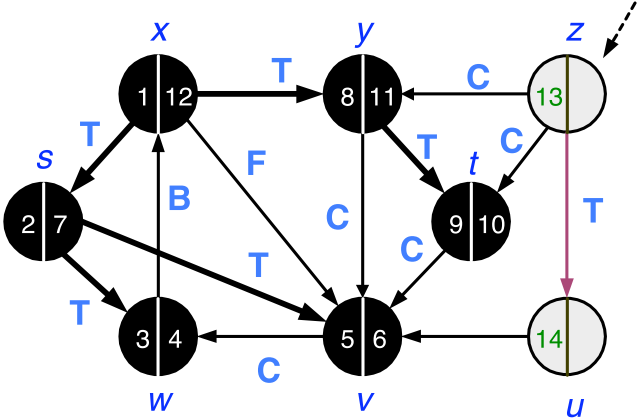
Depth-first search: Edge classification example



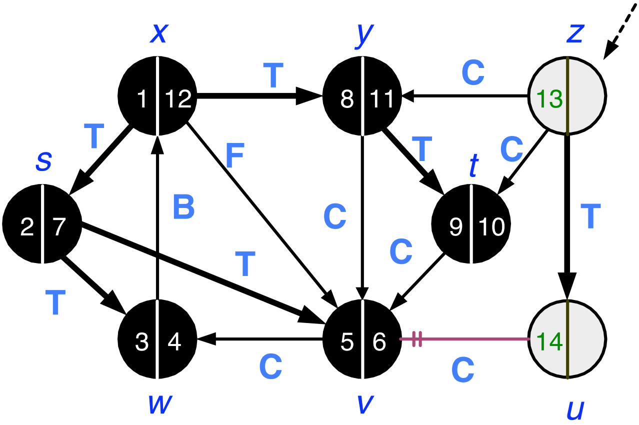
Depth-first search: Edge classification example



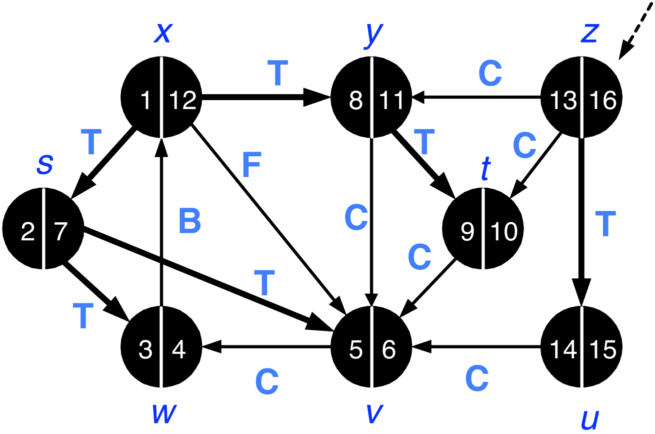
Depth-first search: Edge classification example



Depth-first search: Edge classification example



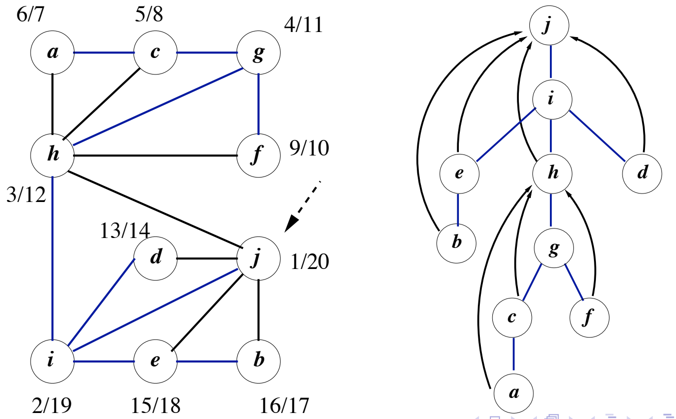
Depth-first search: Edge classification example



Depth-first search: Undirected graphs

Edge classification

Any DFS on an undirected graph produces only Tree and Back edges.



Depth-first search: Non-recursive algorithm

$[\pi, d, f]$=DFS(G, v)

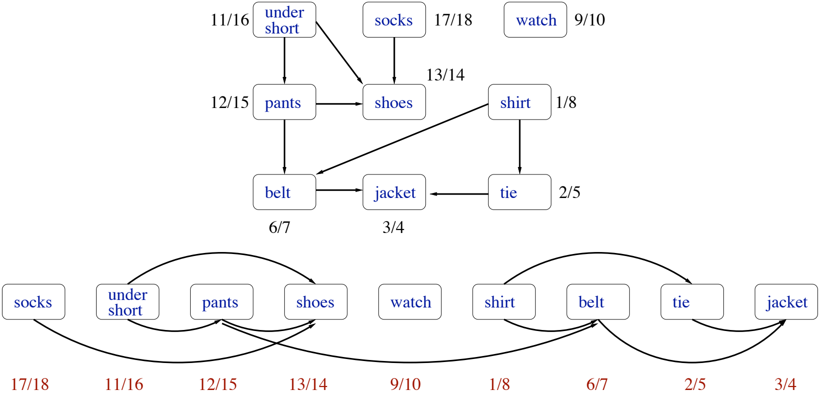
- $top \leftarrow 1$
- $stack(top) \leftarrow v$
- $d(v) \leftarrow ctime \leftarrow 1$
- **while** $top > 0$ **do**
  - $u \leftarrow stack(top)$
  - **if** there is a vertex $w \in Adj(u)$ where $\pi(w)$ is not set **then**
    - $top \leftarrow top + 1$
    - $stack(top) \leftarrow w$
    - $\pi(w) \leftarrow u$
    - $d(w) \leftarrow ctime \leftarrow ctime + 1$
  - **else**
    - $f(u) \leftarrow ctime \leftarrow ctime + 1$
    - $top \leftarrow top - 1$

Topological sort

Topological sort (of a directed acyclic graph): It is a linear ordering of all the vertices such that if (u, v) is a directed edge, then u appears before v in the ordering.

Ordering is not necessarily unique.

Topological sort: Example



Topological sort: Algorithm

The algorithm

- run DFS(G)
- when a vertex is finished, output it
- vertices are output in the reverse topologically sorted order

Runs in $O(V + E)$ time — a linear time algorithm.

The algorithm: Correctness

if $(u, v) \in E$, then $f[u] > f[v]$

Proof: Consider the color of v during exploring the edge (u, v), where u is GRAY. $\square$

v cannot be GRAY (otherwise a Back edge in an acyclic graph !!!).

If v is WHITE, then u is an ancestor of v, hence $f[u] > f[v]$.

If v is BLACK, $f[v]$ is computed already, $f[u]$ is going to be computed, hence $f[u] > f[v]$.

Strongly connected components (SCC)

The strongly connected components of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation.

For a graph $G = (V, E)$, the transpose is defined as $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

Constructing G^T from G takes $O(V + E)$ time with adjacency list (like the CSR or CSC storage format for sparse matrices) representation.

Notice that G and G^T have the same SCCs.

Strongly connected components: Algorithm

(1) Run **DFS**(G) to compute finishing times for all $u \in V$

(2) Compute G^T

(3) Call **DFS**(G^T) processing vertices in main loop in decreasing f[u] computed in Step (1)

(4) Output vertices of each DFT in DFF of Step (3) as a separate SCC

Strongly connected components: Analysis

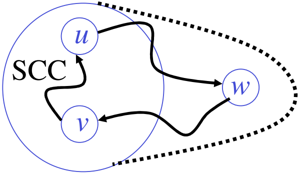
Lemma 1: no path between a pair of vertices in the same SCC, ever leaves the SCC

Proof: let u and v be in the same SCC $\Rightarrow u \leftrightarrows v$

let w be on some path $u \mapsto w \mapsto v \Rightarrow$ $u \mapsto w$

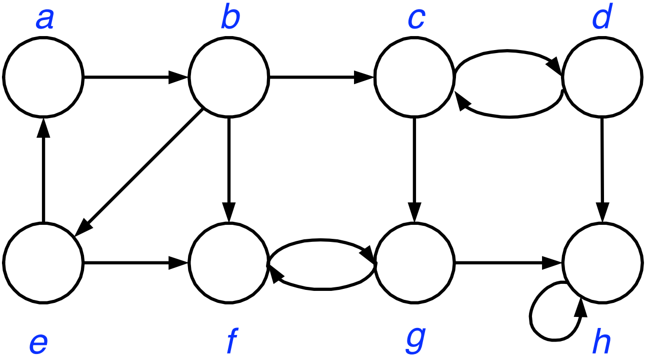
but $v \mapsto u \Rightarrow \exists$ a path $w \mapsto v \mapsto u \Rightarrow$ $w \mapsto u$

therefore u and w are in the same SCC



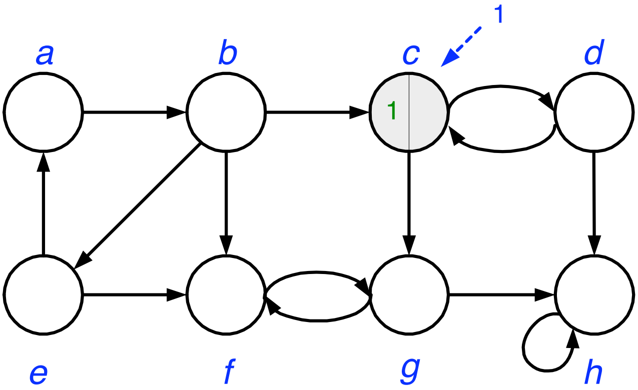
QED

Strongly connected components: Example



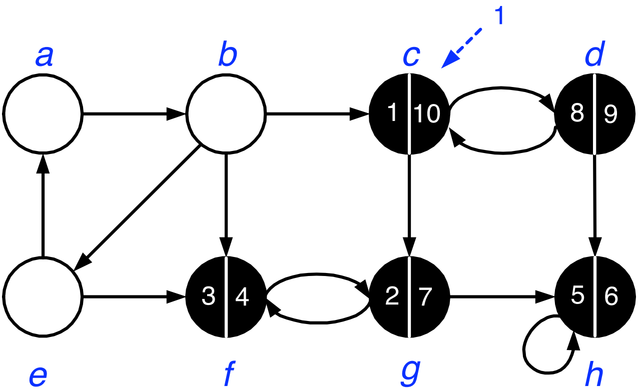
Strongly connected components: Example

(1) Run **DFS**(G) to compute finishing times for all $u \in V$



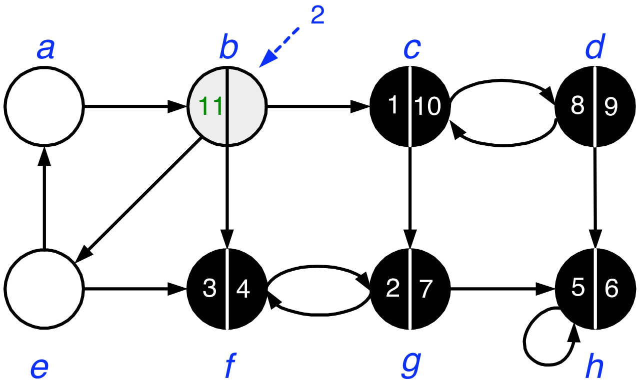
Strongly connected components: Example

(1) Run **DFS**(G) to compute finishing times for all $u \in V$

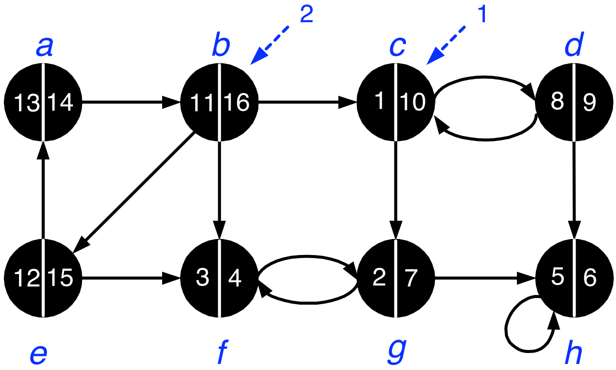


Strongly connected components: Example

(1) Run **DFS**(G) to compute finishing times for all $u \in V$



Strongly connected components: Example

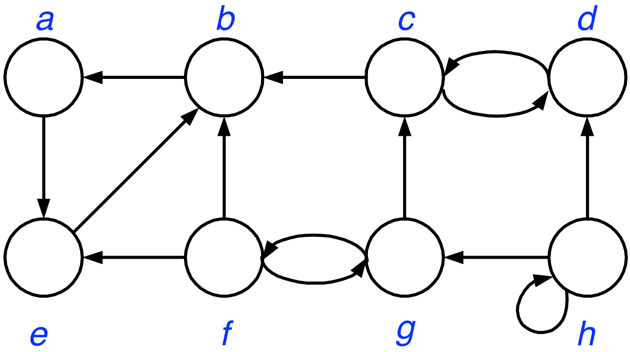


Vertices sorted according to the finishing times:

$$\langle b, e, a, c, d, g, h, f \rangle$$

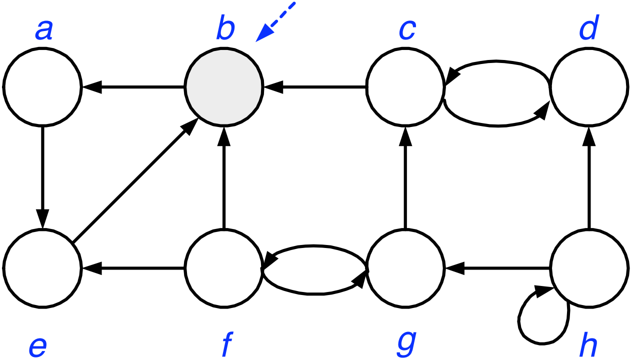
Strongly connected components: Example

(2) Compute G^T



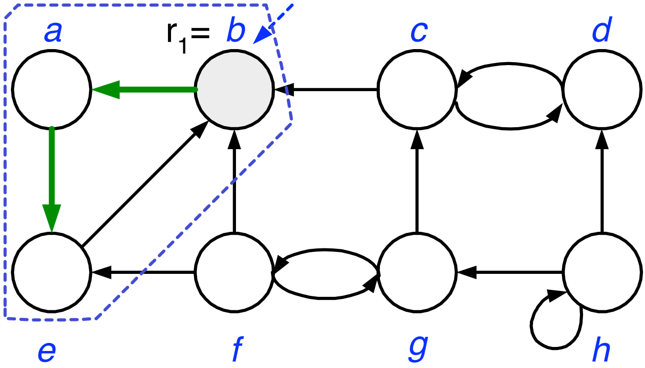
Strongly connected components: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



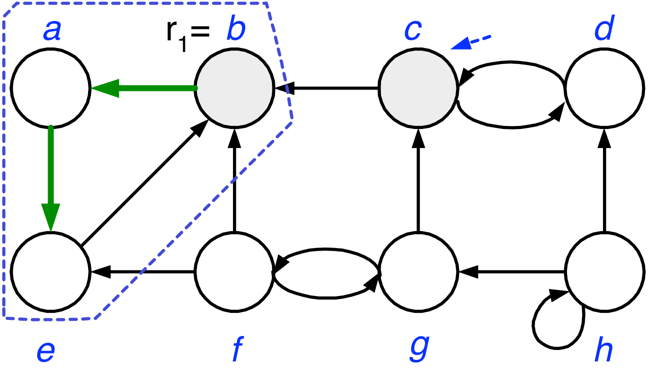
Strongly connected components: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



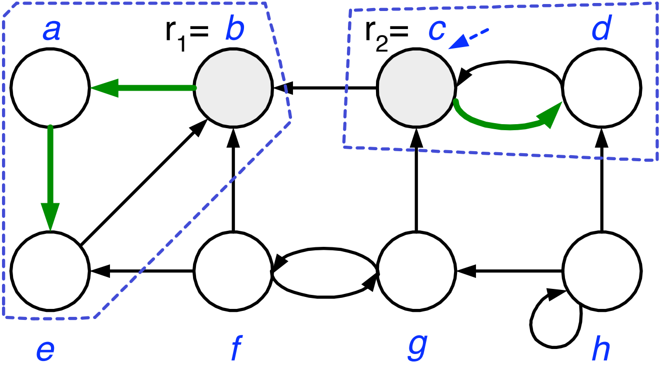
Strongly connected components: Example

(3) Call **DFS**(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



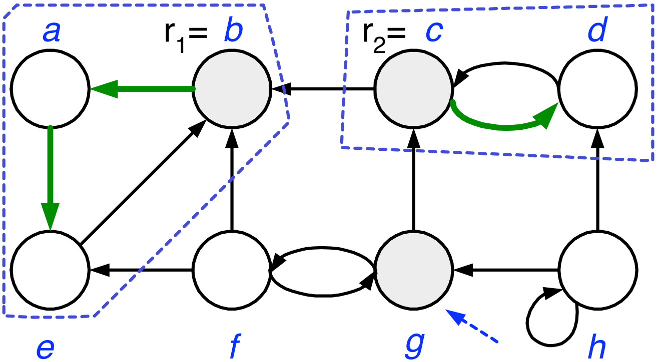
Strongly connected components: Example

(3) Call **DFS**(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



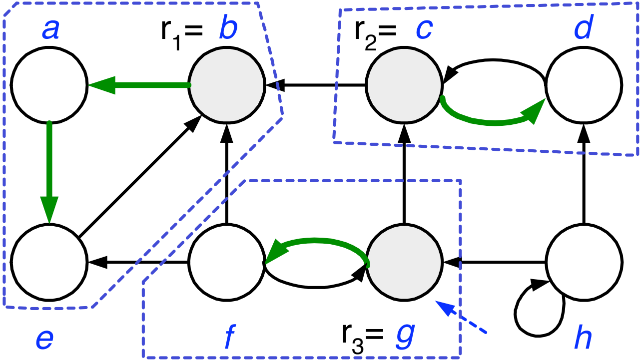
Strongly connected components: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



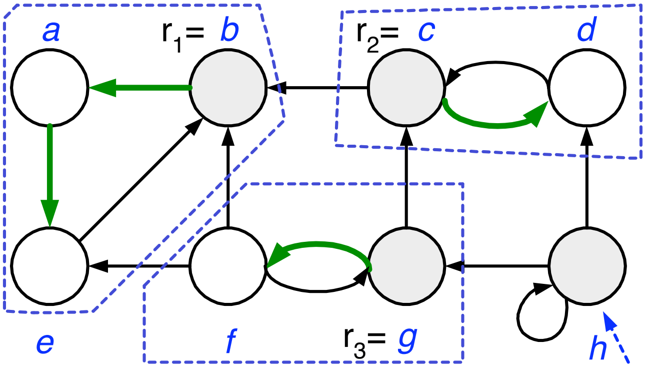
Strongly connected components: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



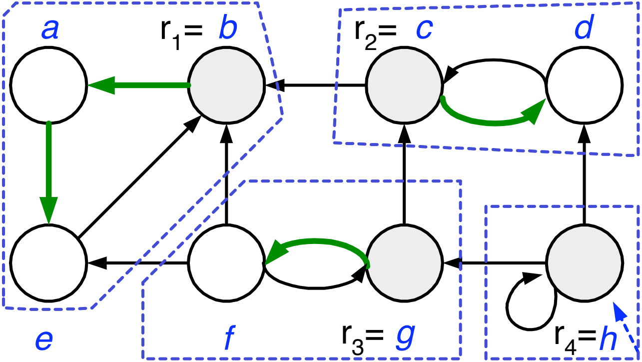
Strongly connected components: Example

(3) Call **DFS**(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$



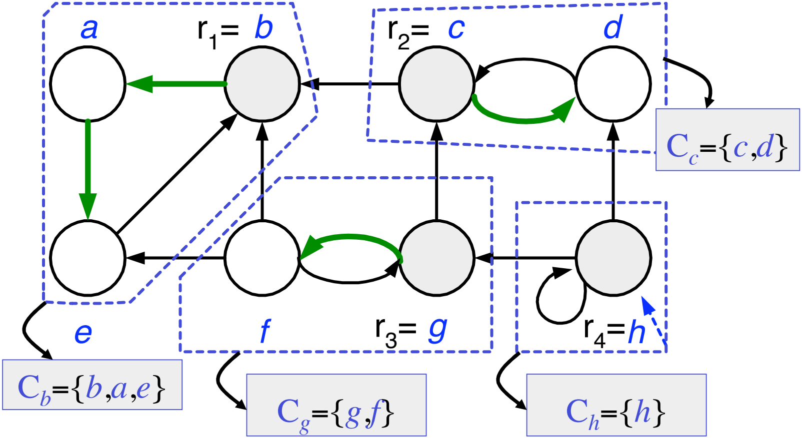
Strongly connected components: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: $\langle b, e, a, c, d, g, h, f \rangle$

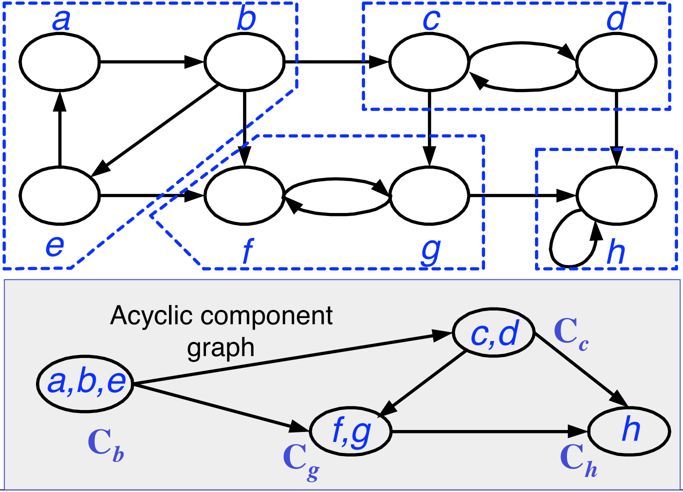


Strongly connected components: Example

(4) Output vertices of each DFT in DFF as a separate SCC



Strongly connected components: Example



Strongly connected components: Observations

- In any DFS(G), all vertices in the same SCC are placed in the same DFT.
- In the DFS(G) step of the algorithm, the last vertex finished in an SCC is the first vertex discovered in the SCC.
- Consider the vertex r with the largest finishing time. It is a root of a DFT. Any vertex that is reachable from r in G^T should be in the SCC of r (why?)

SCC and reducibility

To detect if there exists a permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix} ,$$

where A_{11} is an $r \times r$ submatrix, A_{22} is an $(n-r) \times (n-r)$ submatrix, where $1 \leq r < n$:

run SCC on the directed graph of A to identify each strongly connected component as an irreducible block (more than one SCC?). Hence A_{11}, too, can be in that form (how many SCCs?).

Could not get enough of it: Questions

How would you describe the following in the language of graphs

- the structure of PAP^T for a given square sparse matrix A and a permutation matrix P,
- the structure of PAQ for a given square sparse matrix A and two permutation matrices P and Q,
- the structure of A^k, for $k > 1$,
- the structure of AA^T,
- the structure of the vector b, where $b = Ax$ for a given sparse matrix A, and a sparse vector x.

Could not get enough of it: Questions

Can you define:

- the row-net hypergraph model of a matrix.
- a matching in a hypergraph (is it a hard problem?).

Can you relate:

- the DFS or BFS on a tree to a topological ordering? postordering?

Find an algorithm

- how do you transpose a matrix in CSR or CSC format?