DEPTH-FIRST TRAVERSAL

AND

ITS APPLICATIONS

TO

PROCESSING

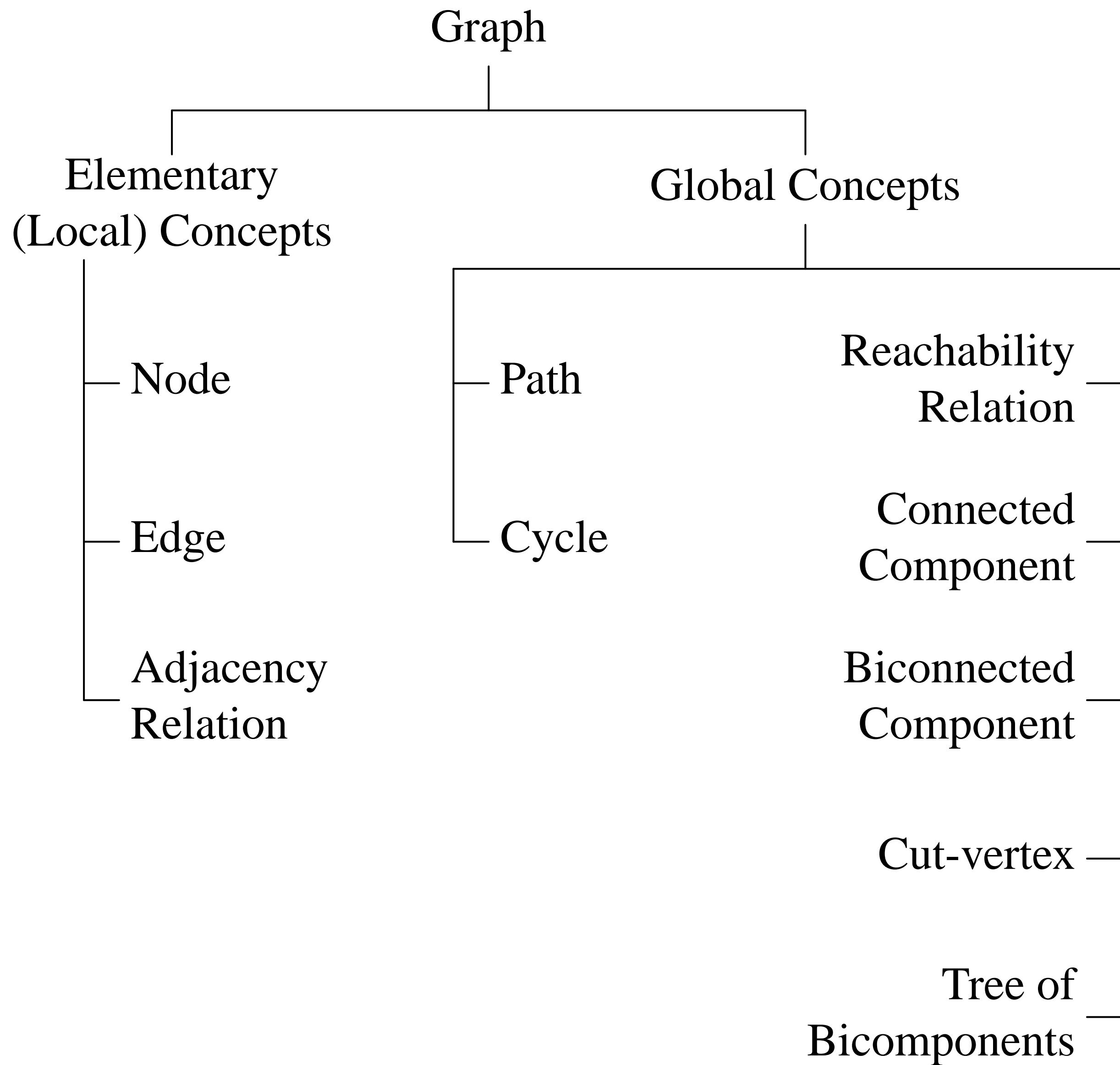
GRAPHS AND DIGRAPHS

CHAPTER GOAL

- Introduce the depth-first traversal of *undirected* graphs.
- Show three important applications:
  - A linear $O(|V| + |E|)$ algorithm for determining the connectedness of a graph G, where $|V|$ = #(vertices in G) and $|E|$ = #(edges in G).
  - Detection of regions in a binary-image.
  - A linear $O(|E|)$ algorithm for determining the bicomponents of a connected graph G.

- Introduce the depth-first traversal of *directed* graphs.
- Show two important applications:
  - A linear $O(|V| + |E|)$ algorithm to determine all nodes reachable from a given node.
  - A linear $O(|V| + |E|)$ algorithm for determining the strong components of a digraph.

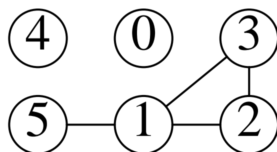
GRAPH CONCEPTS



GRAPHS AND THEIR REPRESENTATION

Graph $G = (V, E)$:

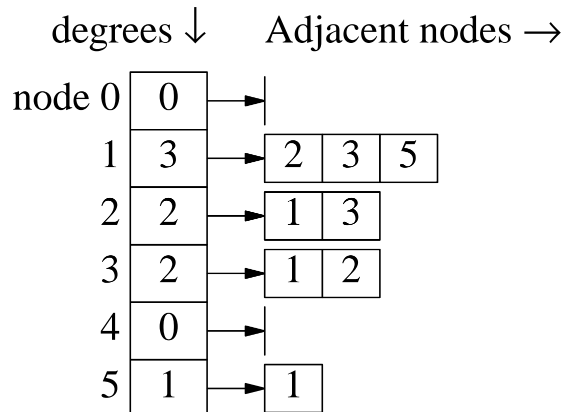
- V = the set of *nodes* (vertices), and
- $E = \{(x, y): x, y \in V \text{ and } x \neq y\}$ is a set of *edges* (links).
- The edges (x, y) and (y, x) are considered the same *undirected* edge, joining the nodes x and y. We say the nodes x and y are *adjacent* if $(x, y) \in E$.
- degree(x) = #(edges adjacent to x); $\sum$degree(x) = 2.$|E|$.



A graph with 6 nodes and 4 edges. Node 0 has no edges adjacent to it.

Adjacency-list Representation:

- $3! \times 2! \times 2! = 24$ possible combination of adj-lists orderings.



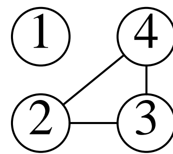

```
typedef struct { int degree, *adjNodes; //arraySize = degree
} GraphNode;  //this suffices when graph does not change

int numNodes;
GraphNode *nodes; //array-size = numNodes
```

A PROGRAM TO READ AN INPUT GRAPH

```
//cc class/algo/c/readGraph.c
//Contact kundu@csc.lsu.edu for comments and questions
//Below is a sample input file: graph.dat
//  6  //numNodes; nodes are numbered as 0, 1, 2, ...
//  0 (0): //nodeNum, degree, adjNodes
//  1 (3): 2 3 5
//  2 (2): 1 3
//  3 (2): 1 2
//  4 (0):
//  5 (1): 1

#include <stdio.h>
#define INPUT_FILE "graph.dat"

typedef struct {
        int degree, *adjNodes;
} GraphNode;

int numNodes, PRINT_SPACE;
GraphNode *nodes;

void PrintIntVector(int *vector, int size)
{ int i;
  for (i=0; i<size; i++)
      printf(" %*d", PRINT_SPACE, vector[i]);
  printf("\n");
}

void SkipToEndOfLine(FILE *fpInp)
{ while (getc(fpInp) != '\n');
}

void ReadInput() //create graph
{ int i, j, nodeNum;
  FILE *fp;
  if (NULL == (fp = fopen(INPUT_FILE, "r")))
     { fprintf(stderr, "Cannot open %s\n", INPUT_FILE); exit(1); }
  fscanf(fp, "%d", &numNodes); SkipToEndOfLine(fp);
  nodes = (GraphNode *)malloc(numNodes * sizeof(GraphNode));
  for (i=0; i<numNodes; i++) {
      fscanf(fp, "%d", &nodeNum);
      fscanf(fp, " ( %d ): ", &nodes[nodeNum].degree);
      nodes[nodeNum].adjNodes = (int *)malloc(nodes[nodeNum].degree*sizeof(int));
      for (j=0; j<nodes[nodeNum].degree; j++)
          fscanf(fp, "%d", &nodes[nodeNum].adjNodes[j]);
      SkipToEndOfLine(fp);
  }
  fclose(fp);
  printf("numNodes = %d\n", numNodes);
  printf("nodeNum (degree): adjNodes\n");
  PRINT_SPACE = (numNodes < 10) ? 1 : 2;
  for (i=0; i<numNodes; i++) {
      printf("%*d (%*d):", PRINT_SPACE, i, PRINT_SPACE, nodes[i].degree);
      PrintIntVector(nodes[i].adjNodes, nodes[i].degree);
  }
}

int main() { ReadInput(); }
```

SOME GRAPH-COMPUTATION PROBLEMS

1? State an algorithm for generating all graphs with n nodes $V = \{1, 2, \cdots, n\}$ and m edges. Explain the steps for the example graph ($n = 4$ and $m = 3$) below. How many such graphs are there?



How is this problem related to the problem of generating binary strings of a given length and a given number of ones?

2? Show all such graphs with $n = 4$ nodes and $m = 3$ edges; here node labels are important and thus the graphs below are considered to be different (although they are isomorphic, i.e., they differ only in node labels).

Two different but isomorphic graphs.

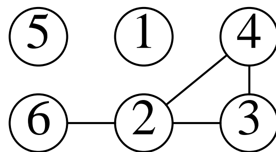
Show all non-isomorphic graphs with $n = 4$ nodes and $m = 3$ edges.

3? State an algorithm for generating a random graph with n nodes and m edges. (We can use such an algorithm to test the average computation time of an algorithm to determine the connectedness of a graph.)

PATHS AND CYCLES IN A GRAPH

Path $\pi(x, y) = \langle x, x_1, x_2, \cdots, x_n, y \rangle$ from x to y:

- Each $(x_i, x_{i+1}) \in E$, where $x_0 = x$ and $x_{n+1} = y$; x = start-node and y = end-node. Usually, all x_i will be distinct.



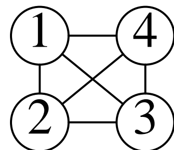
Two acyclic-paths from 3 to 6:
- $\pi_1 = \langle 3, 2, 6 \rangle$
- $\pi_2 = \langle 3, 4, 2, 6 \rangle$

Cycle: A path starting and ending at the same node.

Connected Graph: There is an xy-path for each x and y, $x \neq y$.

The Maximally and The Minimally Connected Graphs:

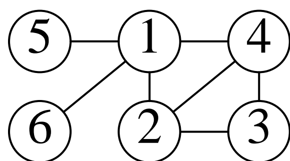
- *Complete Graph K_N:* A graph with $N = |V|$ nodes and $N(N-1)/2 = |E|$ edges connecting every pair of nodes.



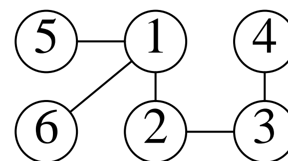
K_4 with $N = 4$ vertices and 6 edges.

- *Tree:* An acyclic connected graph.

Spanning Tree T of G: It has $V(T) = V(G)$ and $E(T) \subseteq E(G)$.



(i) A connected graph G with cycles.



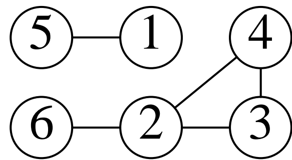
(ii) A spanning tree of G; it has a unique path $\pi(x, y)$ for any $x \neq y$.

Question: How many spanning trees does the above G have?

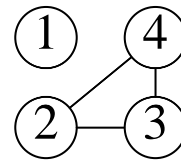
SUBGRAPH AND CONNECTED COMPONENT

Subgraph $G_S = (S, E_S)$ of G on vertices $S \subseteq V$:

- $E_S = \{(x, y) \in E$: both x and $y \in S\}$.
- If $S = \varnothing$, G_S is the empty graph with no nodes and edges.



(i) A graph G.



(ii) The subgraph on $S = \{1, 2, 3, 4\}$.

Question: How many subgraphs does the above G have?

Connected component (in short, *component*):

- A maximal connected subgraph G_S of G.
- "Maximal" means that for $S' \supset S$, $G_{S'}$ is not connected.
  - $S = \{1, 5\}$ and $\{2, 3, 4, 6\}$ give two components of G above.
  - $S = \{2, 3, 4\}$ is not a component though G_S is connected.
  - Only 15 of the 63 non-empty subgraphs of G above are connected.

Question: Which of the following is true?

•? The component of G containing x is given by the subgraph on $S(x) = \{y$: $y = x$ or there is an xy-path in $G\}$.

•? Two components of a graph G have no node in common.

•? G is connected if and only if it has one component.

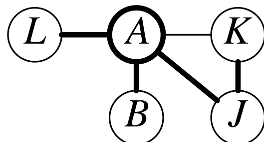
DEPTH-FIRST TRAVERSAL OF A GRAPH

Three Basic Operations:

- *Move forward.* Given a start-node s, follow an edge (y, z) from the current node y to reach a *new* node z not visited before and make z = new current node. The chain of these links (y, z) form a unique path from s to each node visited.
- *Backtrack.* When there is no such (y, z), we backtrack from the current node y to the previous node x on the sy-path, and look for an edge (x, y') to a new node y' not visited yet.
- *Terminate.* Backtracking from s terminates the traversal.

Depth-First Tree: A rooted ordered tree with root = start-node.

Example. dfLabel(y) = i (≥ 1) means y is the ith node visited.



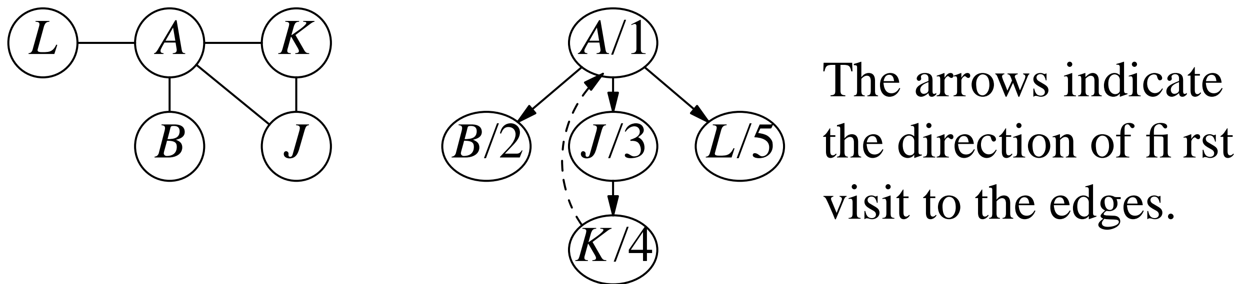
- adj(A) = (B, J, K, L), adj(B) = (A), adj(J) = (A, K), adj(K) = (A, J), and adj(L) = (A).
- The bold edges show the depth-fi rst tree; root = A.

Current node y	Edges (y, z) at y are visited in the order of $z \in$ adj(y). at A	at B	at J	at K	at L
A, dfLabel(A) = 1	(A, B)				
B, dfLabel(B) = 2		(B, A)			
backtrack $B \rightarrow A$	(A, J)				
J, dfLabel(J) = 3			(J, A) (J, K)		
K, dfLabel(K) = 4				(K, A) (K, J)	
backtrack $K \rightarrow J$					
backtrack $J \rightarrow A$	(A, K) (A, L)				
L, dfLabel(L) = 5					(L, A)
backtrack $L \rightarrow A$					
backtrack from A					

DEPTH-FIRST TREE

df-Tree: A rooted ordered tree, with root = start-node and the edges $\{(x, y)$: y is reached first time via edge $(x, y)\}$. Children of a node are ordered by their dfLabels.

Example. A connected graph G and a df-tree for it.



Properties of df-Traversal of A Connected Graph:

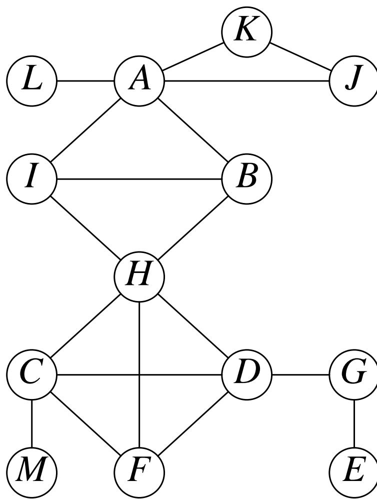
- Each edge of the graph is visited once in each direction.
  - A tree-edge is first visited from parent to child.
  - a non-tree edge (x, y) is first visited from a node x to an ancestor y of x (hence called a *back-edge*). Test for back-edge: $y \neq \text{parent}(x)$ and $\text{dfLabel}(y) < \text{dfLabel}(x)$.
  - No cross-edge (x, y), where y is not a descendant or an ancestor of x.
- The df-Tree depends on both the start-node s and the node-order in each adjacency-list adj(x).
- The df-tree is a spanning tree of G only if G is connected.
- #(backtrackings to x) = #(children of x) $\leq$ degree(x).

Question: List all possible df-trees for the above G and $s = A$. Is there a spanning tree of G which is not a df-tree?

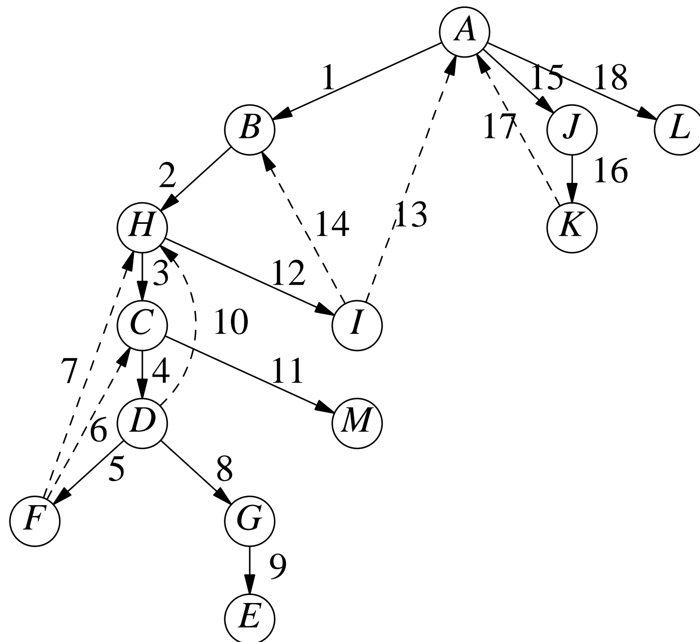
A LARGER EXAMPLE OF DF-TRAVERSAL

- The number of times we backtrack to a node x = #(children of x in the df-tree). Note: backtracking is not the same as visiting a tree-edge the second time (from child to parent).

Example. Each adj(x) is assumed ordered alphabetically as in adj(A) = $\langle B, I, J, K, L\rangle$. The number and the arrow next to an edge in the figure on the right show the order and the direction of first visit of that edge.



(i) A connected graph.



(ii) The df-tree for start-node $s = A$.

Question: Let (y_1, x) and (y_2, x) be two backedges to x and dflabel(y_1) < dfLabel(y_2). Consider the four traversal-events of these edges in different directions: $e_1 = y_1 \rightarrow x$, $e_2 = x \rightarrow y_1$, $e_3 = y_2 \rightarrow x$, and $e_4 = x \rightarrow y_2$. What are the possible orders in which these events can occur (only a few out of 4! = 24 are possible)?

PSEUDOCODE FOR DF-SEARCH

Notations:

nextToVisit(x) = The node in adj(x) to visit next from x.
parent(x) = Parent of x in df-tree; parent(x) = x if x = root.
dfLabel(x) = k (≥ 1) if x is the kth node visited.
currNode = The node at which we explore a new edge.
lastDfLabel = Most recent df-label assigned (to some node).

Algorithm DF-SEARCH: //non-recursive form

Input: The adjacency-lists adj(x) and a start-node s.
Output: The df-tree via parent(x) of each node x visited.

1. [Initialize.] For (each node x), let dfLabel(x) = 0 and nextToVisit(x) = the first item of adj(x). Also, let currNode = parent(s) = s and dfLabel(s) = lastDfLabel = 1.
2. [Move forward.] While (nextToVisit(currNode) $\neq$ null) do the following:
   (a) Let temp = nextToVisit(currNode) and update nextToVisit(currNode) to the next node in adj(currNode).
   (b) If (dfLabel(temp) = 0), then let parent(temp) = currNode, currNode = temp, and dfLabel(currNode) = lastDfLabel = lastDfLabel + 1. (Otherwise, (currNode, temp) is an edge from a node to its parent, or a back-edge, or a forward non-tree edge, i.e., a back-edge in the reverse direction.)
3. [Backtrack or terminate.] If (currNode $\neq$ parent(currNode)), then currNode = parent(currNode) and goto step 2, else stop.

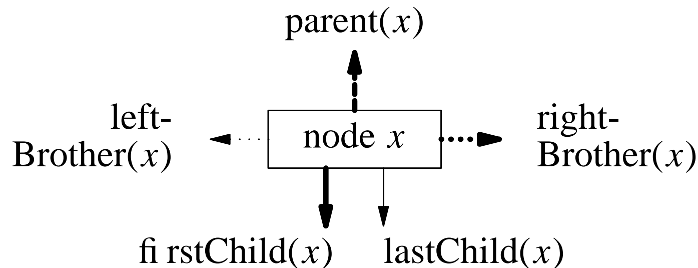
DATA-STRUCTURES FOR BUILDING DF-TREE

Problem: The parent-information does not allow us to access the nodes of the df-tree starting at s = root(df-tree).

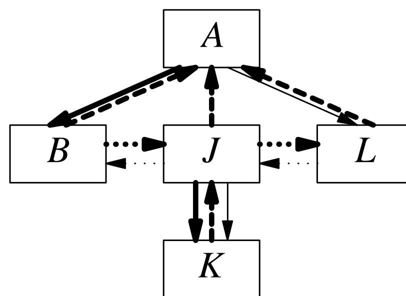
Solution: Set child-pointers and brother-pointers along with parent-information for each node visited. (We may create separate nodes for the df-tree or build the pointers within the graph-nodes. This also holds for the recursive version of DF-SEARCH given later.)

Node-structure for a Rooted Ordered Tree:

```
typedef struct TreeNode {
        int node;
        struct TreeNode *parent, *leftBroth, *rightBroth,
                        *firstChild, *lastChild;
} TreeNode;
```



Example. Here, fi rstChild(J) = lastChild(J).



Shown are only the non-null pointers.

A RECURSIVE VERSION OF DF-SEARCH

Notes:

- Backtracking corresponds to return from the recursion.
- The lastDfLabel and the array dfLabels[1..N] can be *static* in DF-SEARCH; nextToVisit[1..N] is not used.
- The recursive-call at x creates the subtree of df-tree at x as we build the df-tree via child-pointers, etc.

Algorithm DF-SEARCH(currNode): //first call: currNode = s

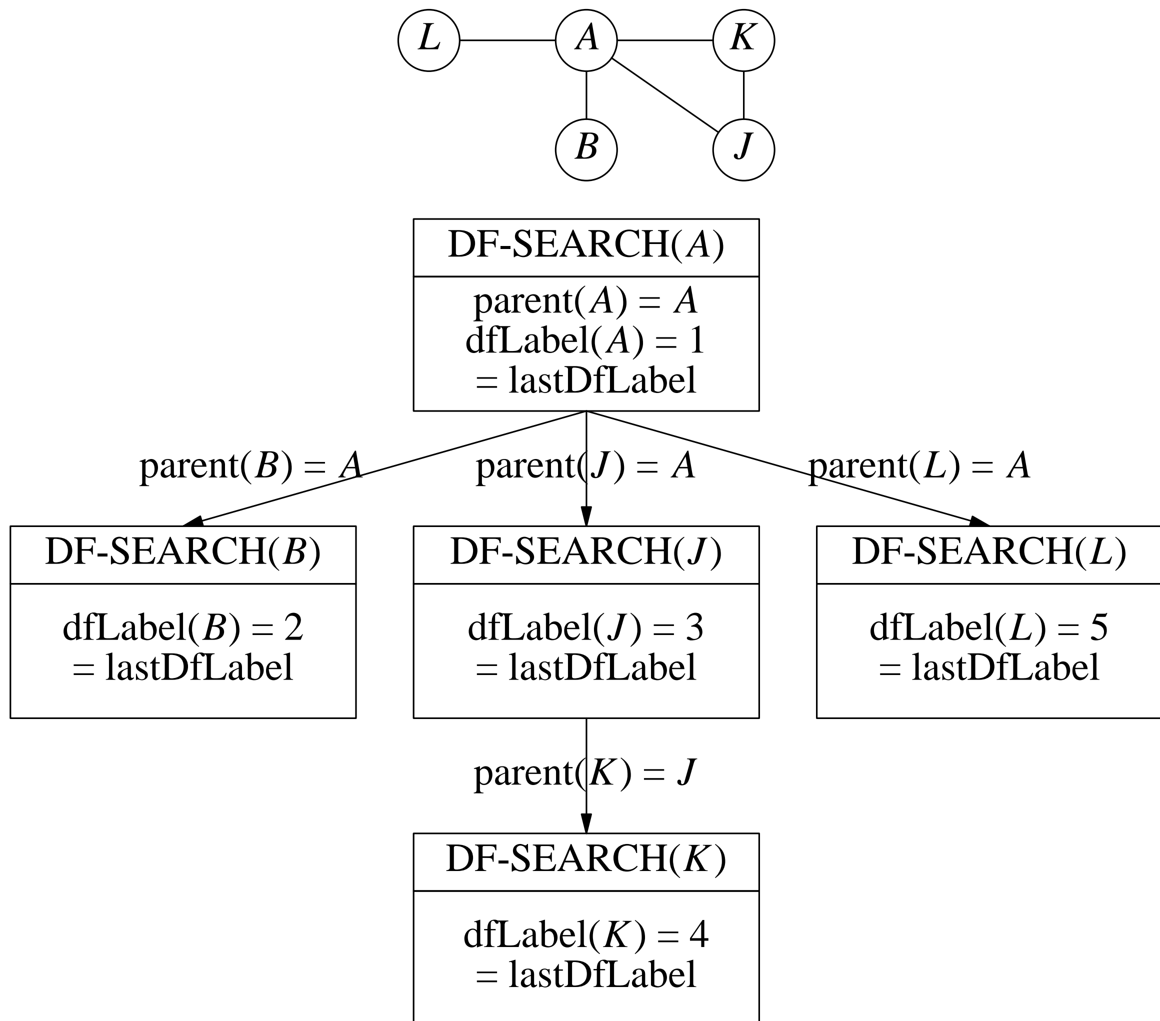
> **Input:** The adjacency-lists adj(x) and the start-node s.
> **Output:** The df-tree, with root = s.

1. [Create df-tree node.] Create currTreeNode in df-tree for currNode. If (lastDfLabel = 0), then initialize dfLabel(x) = 0 for each x and let parent(currTreeNode) = currTreeNode. Now, let dfLabel(currNode) = lastDfLabel = lastDfLabel + 1.
2. For (each node x in adj(currNode)) do the following:
   If (dfLabel(x) = 0) then add the root-node (which corresponds to x) of the subtree returned by DF-SEARCH(x) as the next child of currTreeNode.
3. Return(currTreeNode).

Complexity: $\Theta(|V| + |E|) = \Theta(|E|)$ for a *connected* G; otherwise, $O(|E|)$.

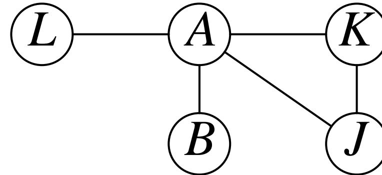
- An edge (x, y) is examined at most once in each direction.
- A constant amount of work per node x (assigning dfLabel(x), adding x to the df-tree, and backtracking from x).

ILLUSTRATION OF THE RECURSIVE DF-SEARCH



- #(calls to DF-SEARCH) = #(nodes in df-tree) = #(nodes in G), if G is connected.
- #(direct calls from DF-SEARCH(x)) = #(children x).

A SAMPLE INPUT GRAPH AND OUTPUT OF DF-SEARCH



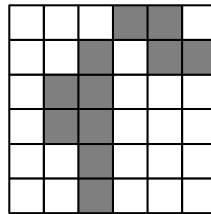
Output for Start-node = 0 (A):

```
numNodes = 5
nodeNum (degree): adjNodes
0 (4): 1 2 3 4
1 (1): 0
2 (2): 0 3
3 (2): 0 2
4 (1): 0
processing (0,1): tree-edge
processing (1,0): tree-edge 2nd-visit
backtracking 1 -> 0
processing (0,2): tree-edge
processing (2,0): tree-edge 2nd-visit
processing (2,3): tree-edge
processing (3,0): back-edge
processing (3,2): tree-edge 2nd-visit
backtracking 3 -> 2
backtracking 2 -> 0
processing (0,3): back-edge 2nd-visit
processing (0,4): tree-edge
processing (4,0): tree-edge 2nd-visit
backtracking 4 -> 0
------------------------------
df-tree in preorder: nodes and the list of children
Node=0 (parent=0, numChildren=3):  1 2 4
Node=1 (parent=0, numChildren=0):
Node=2 (parent=0, numChildren=1):  3
Node=3 (parent=2, numChildren=0):
Node=4 (parent=0, numChildren=0):
------------------------------
```

OBJECT-REGION IN A BINARY-IMAGE

Binary-image:

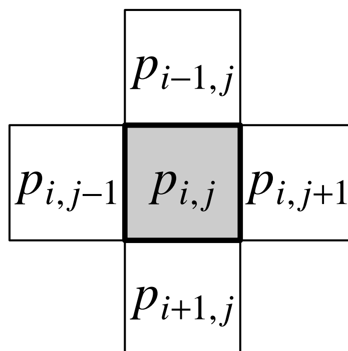
- An $m \times n$ array, where a pixel p_{ij} = 1 (shown below as a shaded square) represents a part of an *object* in the image; a pixel $p_{ij} = 0$ represents a part of the *background*.



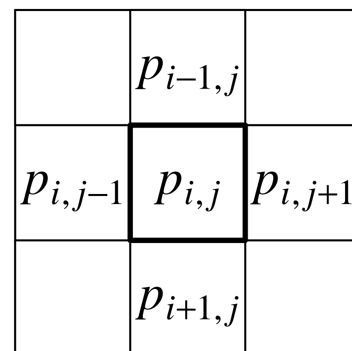
A 6×6 binary image.

Adjacent Pixels:

- Two 1-pixels are adjacent if they share an edge.
- Two 0-pixels are adjacent if they share a corner or an edge.



At most 4 neighbors of an 1-pixel p_{ij}.



At most 8 neighbors of a 0-pixel p_{ij}.

Image-object: A *maximal* set of connected 1-pixels p_{ij}.

Example. The above image has 2 objects of sizes 7 and 4 and two background regions of sizes 24 and 1.

Question: Give a 5×5 image with maximum number of objects.

OBJECT DETERMINATION BY DF-TRAVERSAL

Order of Visit of Neighbors of 1-pixel p_{ij} (if present):

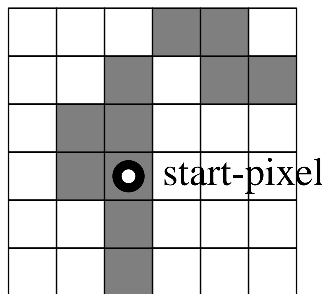
(1) $p_{i,j+1}$ (E-neighbor) (3) $p_{i,j-1}$ (W-neighbor)
(2) $p_{i-1,j}$ (N-neighbor) (4) $p_{i+1,j}$ (S-neighbor)

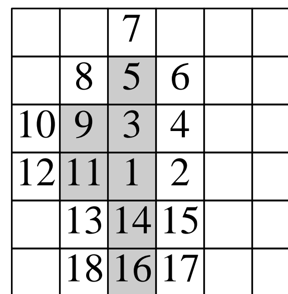
Additional Backtracking Rule:

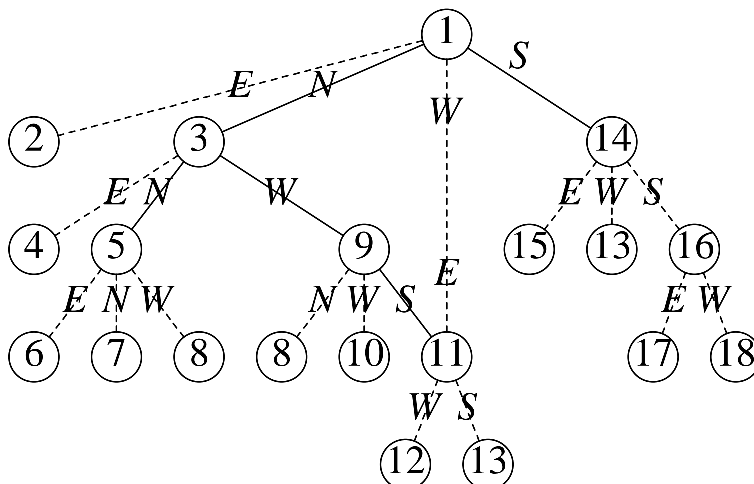
- If we reach a 0-pixel, we immediately backtrack.

Example. Shown below is the depth-fi rst labeling of the pixels visited starting at the position labeled 1.

All neighboring 0-pixels of the region R containing the start-pixel are also visited. The total number of visits to those 0-pixels is at most $2(|R| + 1)$ - why?

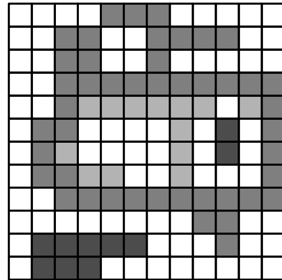






The dashed lines indicate neighbor that are visited but are not in the current region. These nodes appear as many times as they are visited, each time causing an immediate backtracking; both nodes 8 and 13 appear here twice.

OBJECT SIMPLIFICATION BY FILLING UP HOLES IN THE OBJECT

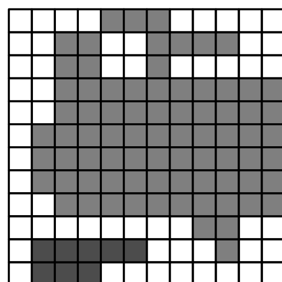


- An image with three objects.
- The largest object contains two holes and one of the objects is placed in one of those holes.
- The non-boundary pixels of this object are shown in light shade.

External Regions:

- The 0-regions which contain at least one 0-pixel from any of top/bottom rows and leftmost/rightmost columns.

Holes: These are 0-regions other than the external regions.



- After fi lling-in the two holes in the largest object.
- This swallows the third object in one of those holes.

Questions:

1? Show df-labels of pixels in the 0-region in the above image starting at $p_{1,1}$; do not label 1-pixels that may be visited.

2? Give a psuedocode to detect the pixels in the holes of an object. What is the computational complexity?

3? Mark the 1-pixel that are visited by starting at the southmost pixel $p_{11,10}$ of the largest object above and using the rule "keep to the right". Assume that we arrived at $p_{11,10}$ by going north, which means $p_{10,10}$ is the next pixel visited.

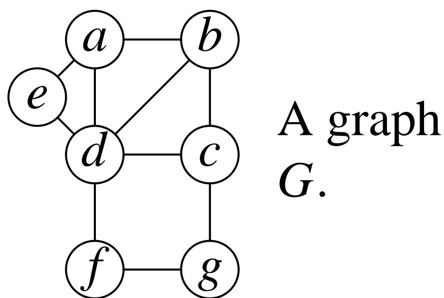
FINDING CHORD-FREE PATHS IN A GRAPH FROM A START-NODE

Chord-free Path:

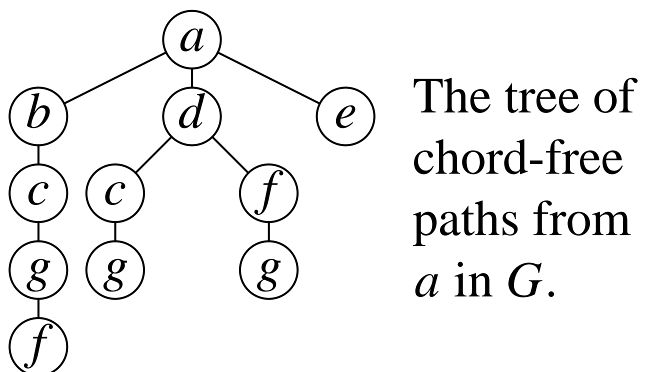
A path $\pi = \langle x_1, x_2, \cdots, x_k \rangle$ in a graph is chord-free if (x_i, x_j) is not an edge for $1 \leq i < j - 1 < k$.

- A subpath of a chord-free path π is also chord-free.

Example. The path $\pi = \langle a, b, d \rangle$ is not chord-free because of the edge (a, d). We can arrange the chord-free paths from a start-node s as a tree as shown below; the four maximal chord-free paths from a are: $\langle a, b, c, g, f \rangle$, $\langle a, d, c, g \rangle$, $\langle a, d, f, g \rangle$, $\langle a, e \rangle$.



A graph G.



The tree of chord-free paths from a in G.

Question:

•? Show the tree of chord-free paths from b in G above.

•? How many chord-free paths are there from node 1 in the complete graph K_n?

•? Show a graph with N (≥ 1) nodes which has an exponential number of chord-free paths for each start-node s.

ALGORITHM FOR CHORD-FREE PATHS

Notes:

- The algorithm below maintains only a small part of the tree of chord-free paths from the start-node at any moment, and does not build the tree of all chord-free paths. In particular, it uses only $O(N)$ memory-space.
- The arrays mark[1..N], and parent[1..N] are static in CHORD-FREE-PATHS.

Algorithm CHORD-FREE-PATHS(currNode): //= s in 1st call

Input: A graph G and a start-node s.
Output: The chord-free paths from s.

1. If (fi rstCall), initalize mark(x) = parent(x) = 0 for each node x, mark(currNode) = 1, and parent(currNode) = currNode.
2. Let $S = \{x \in$ adj(currNode): mark(x) = 0$\}$. Let mark(x) = 1 and parent(x) = currNode for each $x \in S$.
3. If ($S \neq \emptyset$) then for (each $x \in S$) call CHORD-FREE-PATHS(x); else print the path from currNode to the start-node formed by the parent-links.
4. Reset mark(x) = parent(x) = 0 for each $x \in S$ and return.

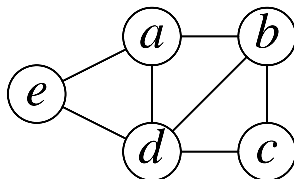
Complexity: Total length of all maximal chord-free paths.

Question:

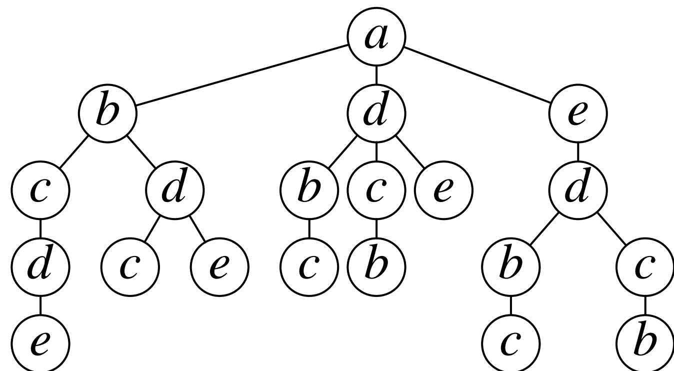
•? Show the recusion call-tree, S, marked-nodes, and their parents before and after each call for the example G shown in the previous page.

ALL ACYCLIC PATHS IN A GRAPH FROM A START-NODE

Example. Shown below is a graph and the tree of all acyclic paths from a.



(i) A graph G.



(ii) The tree of acyclic paths from a in G.

Some Properties of this Tree:

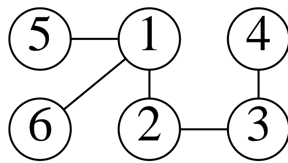
- A node x appears in the tree as many times as the number of acyclic sx-paths.
- We can find all cycles (without repetition of nodes) at a node s from these sx-paths:
  - Combine an sx-path with the edge (s, x), if exists.
  - Each cycle is obtained twice in this process.

Question:

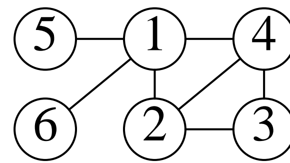
•? State a suitable method of node-marking in the depth-first search to output all (maximal) acyclic-paths from a start-node in a graph. State the complete algorithm in the form of a pseudocode using recursion.

CUT-VERTEX AND BICONNECTED GRAPH

Cut-vertex: A node x is a cut-vertex of G if its deletion breaks up the connected component of G containing x into 2 or more components.

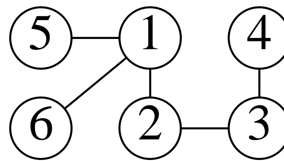


(i) Every non-terminal node in a tree is a cut-vertex.

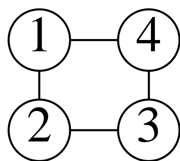


(ii) Node 1 is the only cut-vertex.

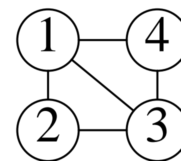
Biconnected: A connected graph G which has ≥ 2 nodes and has no cut-vertex (i.e., we need to delete 2 or more nodes to disconnect the graph, if possible).



(i) A tree is biconnected if and only if it has 1 or 2 nodes.



(ii) A cycle is the simplest biconnected graph with $\geq$3 nodes.



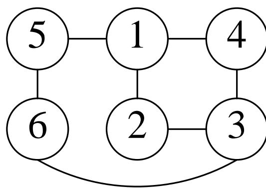
(iii) Adding an edge to a biconnected graph keeps it biconnected.

STRUCTURE OF A BICONNECTED GRAPH

Theorem. A minimal biconnected graph $G = (V, E)$, $|V| \geq 3$ is made of cycles $C_1, C_2, \cdots, C_m$ ($m \geq 1$), where

- each C_i, $i > 1$, contains at least one node not in $C_1 \cup C_2 \cup \cdots \cup C_{i-1}$, and
- each C_i, $i > 1$, has at least one edge in common with $C_1 \cup C_2 \cup \cdots \cup C_{i-1}$.

Example. A biconnected graph and its cycles C_i as in the Theorem above. (What is another choice for C_3?).



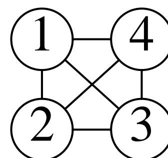
C_1: $\langle 1, 5, 6, 3, 4, 1 \rangle$
C_2: $\langle 1, 2, 3, 4, 1 \rangle$, $2 \notin C_1$, $(1, 4) \in C_1 \cap C_2$

Question:

•? Is there another choice of such cycles?

•? Why do we assume that $|V| \geq 3$ above?

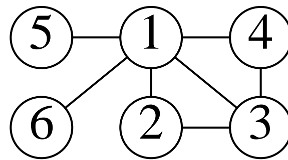
•? What are some choice of cycles for K_4?



BICOMPONENT

Bicomponent: A maximal biconnected subgraph G_S of G.

Example. A graph G with three bicomponents; $S = \{1, 3, 4\}$ gives a biconnected subgraph but it is not maximal.

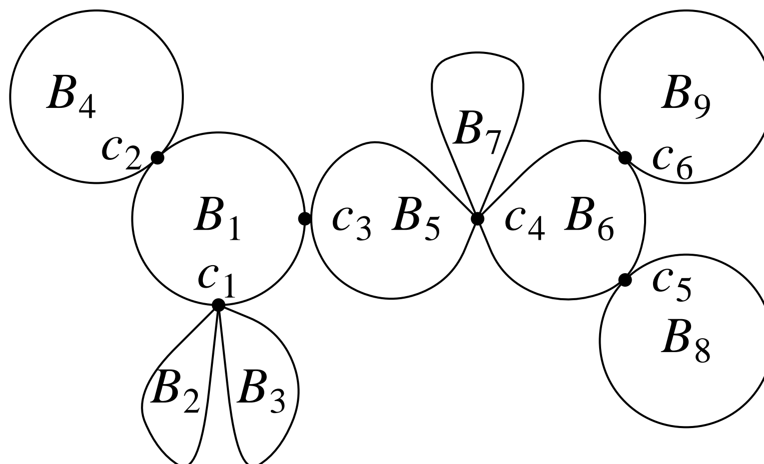


Three bicomponents:
$S = \{1, 5\}$, $\{1, 6\}$, and $\{1, 2, 3, 4\}$

Question: How many biconnected subgraphs does G have?

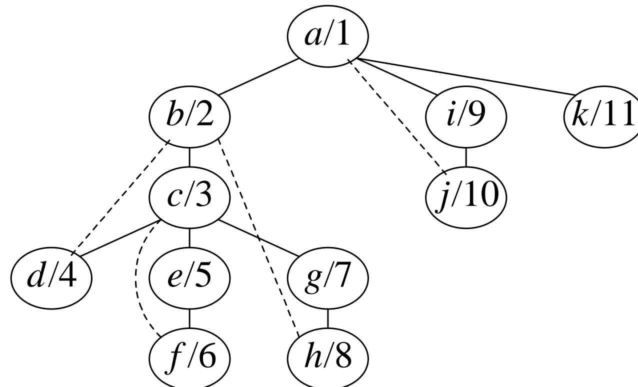
Theorem.

- Two bicomponents of a graph G can have at most one node in common, and such a vertex is a cut-vertex.
- The bicomponents of a connected graph form a tree structure. (There is no edge between two bicomponents, and hence a cycle is fully contained within a bicomponent.)



A graph with bicomponents B_1 to B_9 joined in the form of a tree via cut-vertices c_1 to c_6.

AN EXAMPLE OF A DF-TREE AND SOME OBSERVATIONS ON BICOMPONENTS



- There are 5 bicomponents and 3 cut-vertices:

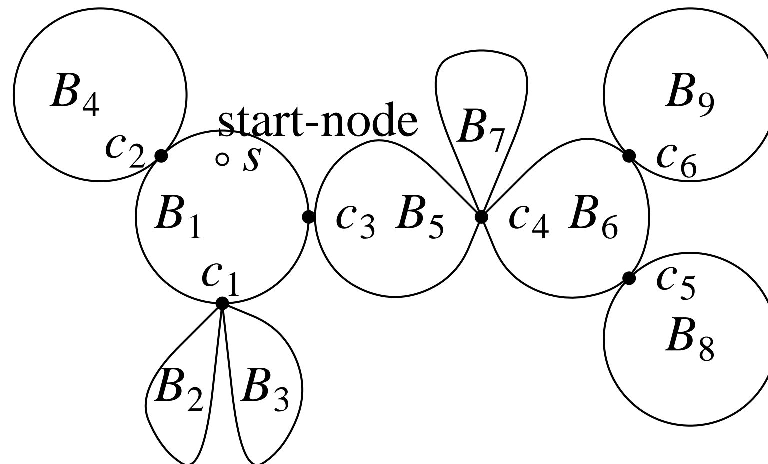
$\{c, e, f\}$	entered via cut-vertex c
$\{b, c, d, g, h\}$	entered via cut-vertex b
$\{a, b\}$	(entered via cut-vertex a)
$\{a, i, j\}$	(entered via cut-vertex a)
$\{a, k\}$	(entered via cut-vertex a)

- The root node has more than one child if and only if it is a cut-vertex.
- The subtree of df-tree at a node x consists of all nodes reachable from x in the graph without using the ancestors of x.
- Once we enter a bicomponent, we traverse it fully (all nodes and edges) before existing it by backtracking out of the cut-vertex through which it was entered.

Question:

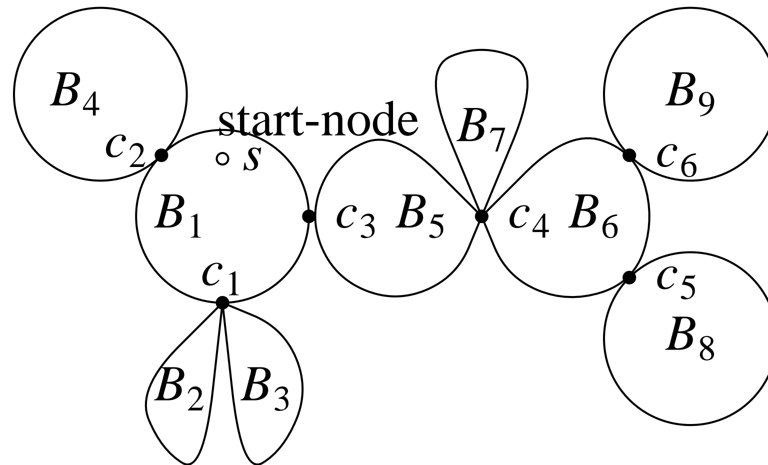
•? If we were to add the edge (g, i) to the graph, how will the df-tree and the bicomponents change?

OBSERVATIONS ON DF-TRAVERSAL OF BICOMPONENTS



- Each bicomponent B_i, other than the one containing the start-point s, is entered the first time via a cut-vertex $c(B_i) \in B_i$.
  - B_2 (B_3) is entered the first time via $c_1 = c(B_2)$ $(= c(B_3))$.
  - B_6 is entered the first time via $c_4 = c(B_6)$.
- As we backtrack to the cut-vertex $c(B_i)$ from some node in B_i, all nodes and edges of B_i have been visited, including those of other bicomponents entered since entering B_i.
  - As we backtrack to c_5 from a node in B_8, we complete visiting all nodes and edges of B_8.
  - When we later backtrack to c_4 from a node in B_6, we complete visiting all nodes and edges of B_6, B_8, and B_9.
  - Then, when we even later backtrack to c_3 from a node in B_5, we complete visiting all nodes and edges of B_5 to B_9.

(CONTINUED)

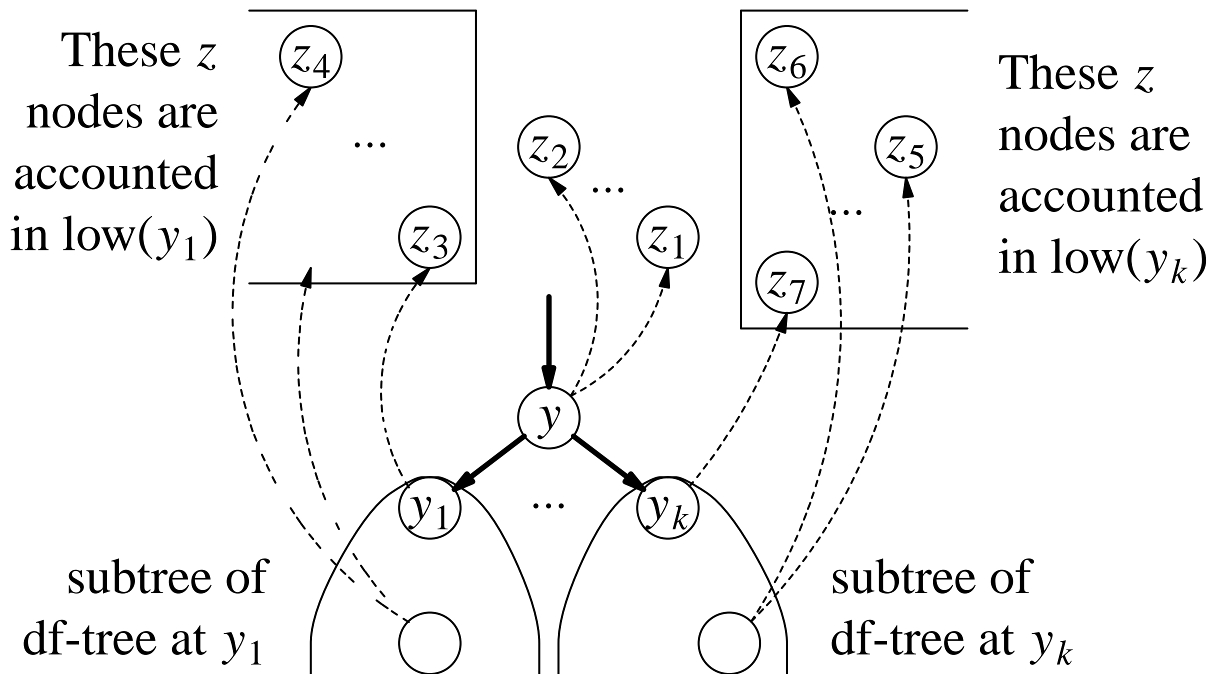


- Since each cycle is contained within a bicomponent, there is no back-edge from a descendant of $c(B_i)$ to its ancestors.
  - This gives us a method to detect the cut-vertex $c(B_i)$.
- The df-tree has exactly one child of $c(B_i)$ in B_i because the deletion of $c(B_i)$ will otherwise break up B_i into two or more connected pieces corresponding to subtrees at $c(B_i)$.

Detection of Cut-Vertex:

- Let low(y) = min {df-label(z): $z = y$ or there is a back-edge to z from y or a descendant of y}.
  - For $y \in B_i$ and $y \neq c(B_i)$, low(y) $\geq$ df-label($c(B_i)$).
- low(y) is known when we are ready to backtrack from y.
- When we backtrack to x from a child y of x, x is a cut-vertex if low(y) $\geq$ dfLabel(x).

ALTERNATE METHOD FOR COMPUTING low(y)



New Equation for low(y):

= min { df-label(z): $z = y$ or
there is a backedge to z from y or
from a *descendant* of y
}

= min { df-label(y),
min {df-label(z): (y, z) is a back-edge},
min {low(y_i): y_i is a *child* of y in df-tree}
}

Question:

•? Since low(y_i) involves dfLabel(y_i) in the minimum and low(y) does not involve dfLabel(y_i), why do we still have the second equality above?

MODIFICATION TO DF-SEARCH FOR COMPUTING low(x)

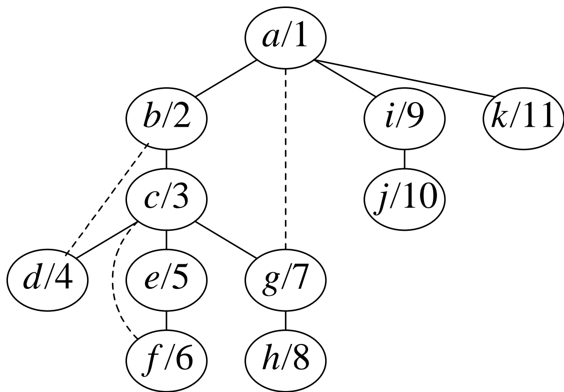
Computing low(x):

1. [Initialize low(x).] As you compute dfLabel(x), let low(x) = dfLabel(x).
2. [Update low(x) for a back-edge.] When a back-edge (x, y) is detected, let low(x) = min {low(x), dfLabel(y)}.
3. [Update low(x) as you backtrack to x from a child y of x.] Let low(x) = min {low(x), low(y)}.

Computing Bicomponents: Uses a separate stack S.

1. Every time you add a node x to the dfTree, add x to S.
2. Before backtracking from y to x = parent(y) do the following:
   (i) If (($x \neq$ root) and (low(y) $\geq$ dfLabel(x)), then x is a cut-vertex and output x together with the nodes in S upto y as a bicomponent. Also, remove the items in S upto y.
   (ii) If (x = start-node) then do the the same as (i) except that x is a cut-vertex if and only if there are unvisited nodes in adj(start-node). (S will now have only start-node after the output of the bicomponent.)

EXAMPLE OF BICOMPONENT COMPUTATION



low(d) = min {4, 2} = 2.
low(e) = min {5, 3} = 3.
low(g) = min {7, 1} = 1.

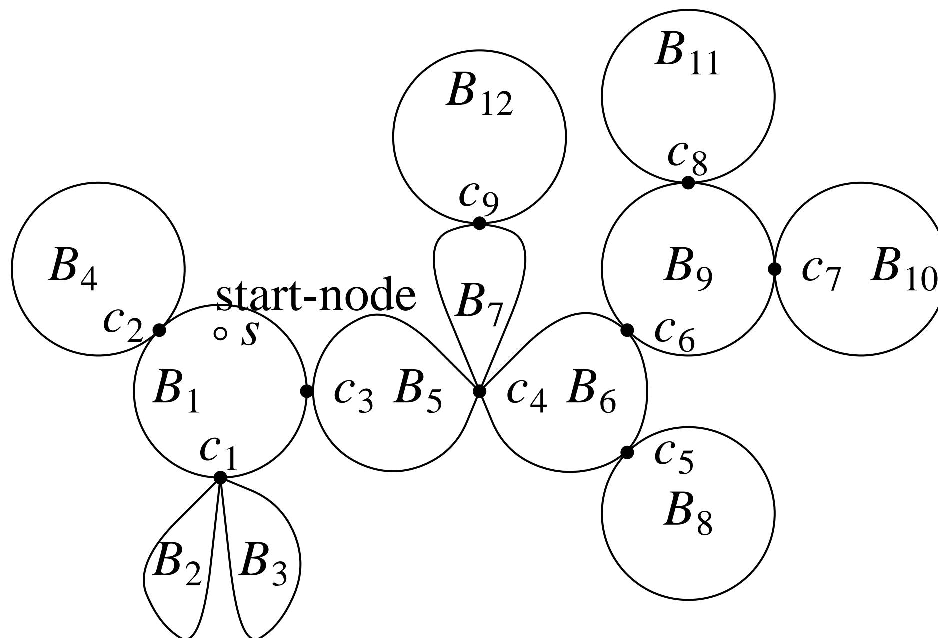
low(c)
= min {3, low(d), low(e), low(g)}
= {3, 2, 3, 1} = 1.

(Shown below are only the first visit to an edge and backtracking.)

New link processed	Dflabel	Low	Cut-vertex	Stack (top on left)	Bicomponent
start at a	dfLabel(a) = 1	low(a) = 1		a	
(a, b)	dfLabel(b) = 2	low(b) = 2		b, a	
(b, c)	dfLabel(c) = 3	low(c) = 3		c, b, a	
(c, d)	dfLabel(d) = 4	low(d) = 4		d, c, b, a	
(d, b)		low(d) = 2			
		= min{4, 2}			
backtracking $d \to c$		low(c) = 2	(low(d) < dfLabel(c))		
		= min{3, 2}			
(c, e)	dfLabel(e) = 5	low(e) = 5		e, d, c, b, a	
(e, f)	dfLabel(f) = 6	low(f) = 6		f, e, d, c, b, a	
(f, c)		low(f) = 3			
		= min{6, 3}			
backtracking $f \to e$		low(e) = 3	(low(f) < dfLabel(e))		
		= min{5, 3}			
backtracking $e \to c$		low(c) = 2	c	d, c, b, a	{c, f, e}
		= min{2, 3}	(low(e) ≥ dfLabel(c))		
(c, g)	dfLabel(g) = 7	low(g) = 7		g, d, c, b, a	
(g, a)		low(g) = 1			
		= min{7, 1}			
(g, h)	dfLabel(h) = 8	low(h) = 8		h, g, d, c, b, a	
backtracking $h \to g$		low(g) = 1	g	g, d, c, b, a	{g, h}
		= min{1, 8}	(low(h) ≥ dfLabel(g))		
backtracking $g \to c$		low(c) = 1	(low(g) < dfLabel(c))		
		= min{2, 1}			
backtracking $c \to b$		low(b) = 1	(low(c) < dfLabel(b))		
		= min{2, 1}			
backtracking $b \to a$				a	{g, d, c, b, a}
(a, i)	dfLabel(i) = 9	low(i) = 9	a	i, a	
		a			
(i, j)	dfLabel(j) = 10	low(j) = 10		j, i, a	
backtracking $j \to i$		low(i) = 9	i	i, a	{i, j}
		= min{9, 10}	(low(j) ≥ dfLabel(i))		
backtracking $i \to a$				a	{a, i}
(a, k)	dfLabel(k) = 11	low(k) = 11		k, a	
backtracking $k \to a$				a	{a, k}

EXERCISE

1. Consider a graph with bicomponents B_1 to B_{12} connected to each other as shown below via the cut-vertices c_1 to c_9. Let s = start-node in B_1 and $s \neq c_1$, c_3 and d be a given node in B_9 and d not a cut-vertes, i.e., $d \neq c_6$, c_7, and c_8. If we do a df-traversal from s to determine B_i's, then which B_i's might be detected before we discover B_9; state the case of largest number of B_i's and of the smallest number of B_i's. in each case, give all possible order in which those B_i's might be detected.

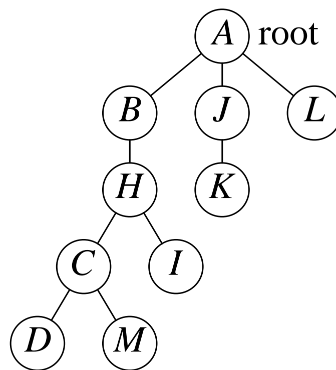


Clearly describe the logic for distinguishing the cut-vertices c_j and the bicomponents B_j along the path of bicomponents from B_1 to B_9 during the df-traversal. Your logic should work for the general case. First, consider the case $d \neq$ a cut-vertex and then the case d = a cut-vertex.

MINIMUM EDGE-ADDITION TO MAKE A CONNECTED GRAPH BICONNECTED

Problem: Given a connected graph G with $n \geq 3$ nodes, find a set of edges E_0 that can be added to G to make it biconnected and $|E_0|$ is minimum.

Example: If G is a tree, consider it as a rooted tree with an arbitrary node x of degree(x) ≥ 2 slected as the root.



Two of the many solutions:
$E_0 = \{(D, M), (M, I), (I, K), (K, L)\}$
$E_0 = \{(D, M), (D, I), (D, K), (D, L)\}$.

Idea: Any connected G is a "tree" of bicomponents.

Algorithm: //Complexity $O(|E|)$

1. Use df-traversal of G to detect bicomponents and let $E_0 = \emptyset$.
2. For (each terminal bicomponent C_j of G), do the following:
   (a) Choose $x_j \in C_j$ and x_j not a cut-vertex.
   (b) Add (x_{j-1}, x_j) to E_0 if $j > 1$.

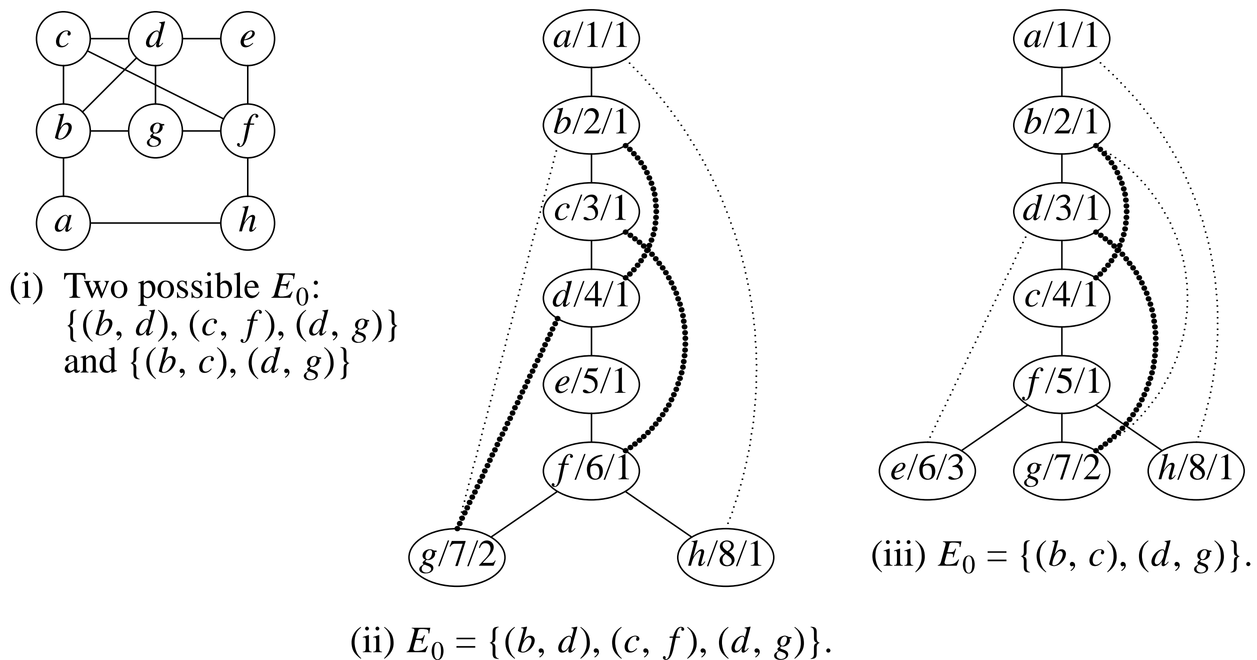
Question: State clearly how to determine the "terminal" bicomponents during df-traversal and select $x_j \in C_j$ such that x_j not a cut-vertex.

MAXIMAL EDGE-ELIMINATION KEEPING BICONNECTEDNESS

Problem: Given a biconnected graph G, find a maximal set of edges E_0 in G whose removal keeps G biconnected.

Idea: In the bicomponent algorithm, we keep one edge from x or a descendant of x corresponding to low(x). Let E_0 be the set of all other backedges.

Example. Shown below in (i) is a graph and two E_0's. They result from different df-trees as shown in (ii)-(iii), where each node shows its dfLabel(x) and low(x); the bold dotted lines are the back-edges in E_0.



(i) Two possible E_0: $\{(b, d), (c, f), (d, g)\}$ and $\{(b, c), (d, g)\}$

(ii) $E_0 = \{(b, d), (c, f), (d, g)\}$.

(iii) $E_0 = \{(b, c), (d, g)\}$.

Question: Give an example G for which the above idea does not work. Then, refine the idea to make it work (by removal of other back-edges, if necessary).

Application of bicomponent decomposition to shortest-path computation:

(a) Consider a source-node $s \in B_1$ and a destination-node $d \in B_6$ in the graph shown above. Also assume $s \neq c_3$ and $d \neq c_4$. Then, each sd-path consists of an sc_3-path, a c_3c_4-path, and an c_4d-path.

(b) Therefore, each shortest sd-path consists of shortest paths from s to c_3, c_3 to c_4, and c_4 to d.

(c) To compute a shortest sd-path, we can then use Dijkstra's algorithm to the smaller graphs B_1, B_5, and B_6 separately. This means a linear algorithm for computing the bicomponents can give us a signifi cant reduction in the computation because of the $O(N^2)$ complexity of Dijkstra's algorithm.

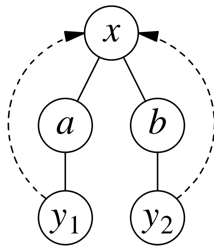
POSSIBLE ORDER OF VISITS TO BACK-EDGES AND THE SECOND VISITS TO THEM

Assumptions:

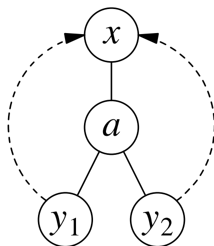
- (y_1, x) and (y_2, x) are two back-edges to x.
- dflabel(y_1) < ddfLable(y_2).

Notations: $e_1 = (y_1, x)$, $e_2 = (y_2, x)$, $e_3 = (x, y_1)$, $e_4 = (x, y_2)$.

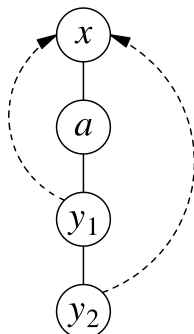
Different Cases:



adjList(x) = (a, y_1, b, y_2): e_1, e_3, e_2, e_4
adjList(x) = (a, b, y_1, y_2): e_1, e_2, e_3, e_4
adjList(x) = (a, b, y_2, y_1): e_1, e_2, e_4, e_3



adjList(x) = (a, y_1, y_2): e_1, e_2, e_3, e_4
adjList(x) = (a, y_2, y_1): e_1, e_2, e_4, e_3



adjList(x) = (a, y_1, y_2): e_1, e_2, e_3, e_4
adjList(x) = (a, y_1, y_2): e_2, e_1, e_3, e_4
adjList(x) = (a, y_2, y_1): e_1, e_2, e_4, e_3
adjList(x) = (a, y_2, y_1): e_2, e_1, e_4, e_3