Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Graph Terminology and Representations

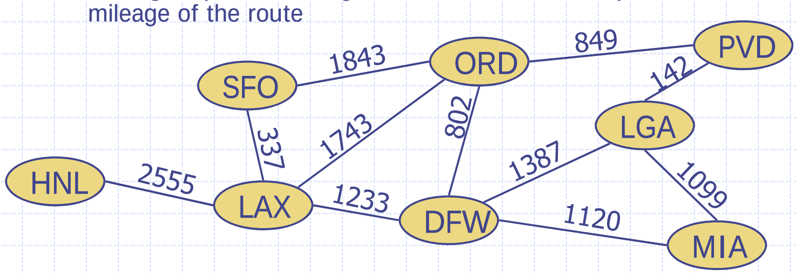
The metropolitan area of Milan, Italy at night. Astronaut photograph ISS026-E-28829, 2011. U.S. government image. NASA-JSC.

1

1

Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route

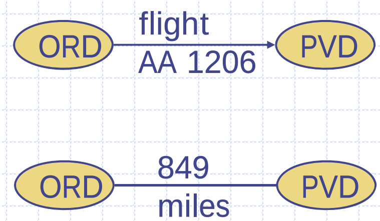


2

2

Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

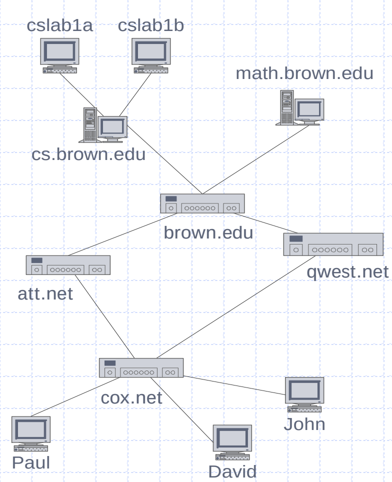


3

3

Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram

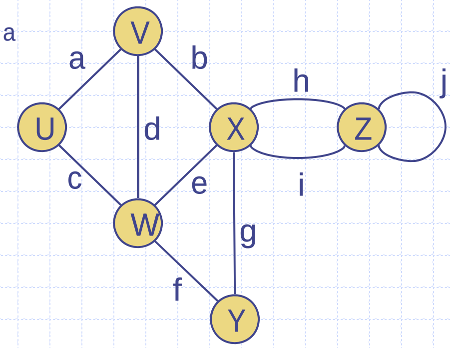


4

4

Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Loop
  - j is a loop

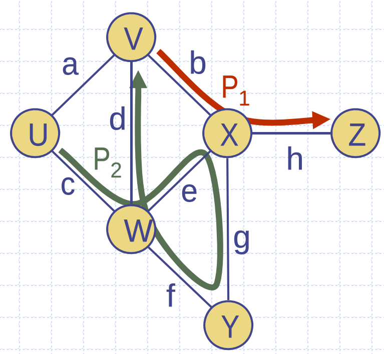


5

5

Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
  - it contains at least one edge
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple

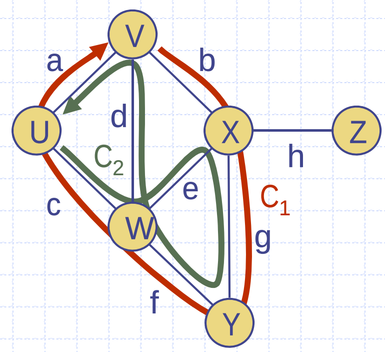


6

6

Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - $C_1=(V,b,X,g,Y,f,W,c,U,a,\lrcorner)$ is a simple cycle
  - $C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,\lrcorner)$ is a cycle that is not simple
- Edges can be dropped if no multiple edges exist.



From now on, the default is that a graph has no multiple edges.

7

7

Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

In an undirected graph with no loops and no multiple edges

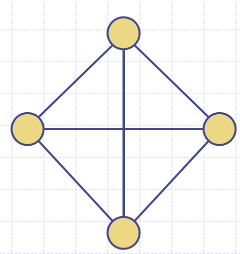
$m \le n(n-1)/2$

Proof: each vertex has degree at most $(n-1)$

What is the bound for a directed graph?

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

8

8

Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- A **Vertex** is can be an abstract unlabeled object or it can be labeled (e.g., with an integer number or an airport code) or it can store other objects
- An **Edge** can likewise be an abstract unlabeled object or it can be labeled (e.g., a flight number, travel distance, cost), or it can also store other objects.

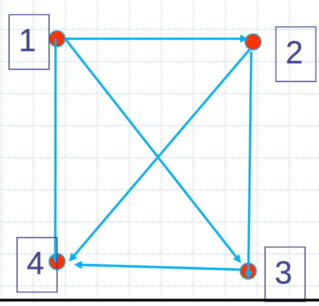
9

9

Relations vs Graph

- A relation R on the set A is a subset of A×A.
- There is 1-to-1 correspondence between R and (directed) G=(A, R).

- **Example:** Let A = {1, 2, 3, 4}. Which ordered pairs are in the relation R = {(a, b) | a < b} ?

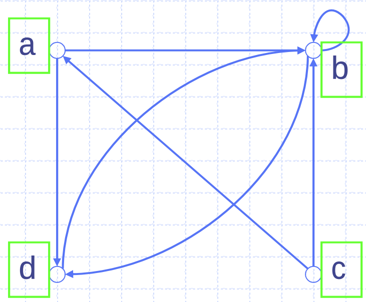


R = { (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) }

10

Representing Relations Using Digraphs

- **Example:** Display the digraph with V = {a, b, c, d}, E = {(a, b), (a, d), (b, b), (b, d), (c, a), (c, b), (d, b)}.



An edge of the form (b, b) is called a **loop**.

11

Relations on a Set

- **How many different relations can we define on a set A with n elements?**
- A relation on a set A is a subset of A×A.
- How many elements are in A×A ?
- The number of subsets that we can form out of a set with m elements is 2^m. Therefore, 2^{n^2} subsets can be formed out of A×A.
- **Answer:** We can define 2^{n^2} different relations on A. As a result, we have that much directed graphs on n points.

12

Possible Quiz Question

- **How many different undirected graphs over n points?**
- **How many different loop-free directed graphs over n points?**

13

13

Properties of Relations

- **Definition:** A relation R on a set A is called **reflexive** if (a, a)∈R for every element a∈A.
- The graph that each node has a loop represents a reflexive relation.

14

Properties of Relations

Definitions:

- A relation R on a set A is called **symmetric** if (b, a)∈R whenever (a, b)∈R for all a, b∈A.
  - Every undirected graph represents a symmetric relation.
- A relation R on a set A is called **antisymmetric** if a = b whenever (a, b)∈R and (b, a)∈R.
  - (N, ≤) is antisymmetric
- A relation R on a set A is called **asymmetric** if (a, b)∈R implies that (b, a)∉R for all a, b∈A.
  - (N, <) is asymmetric

- What is the relation between “antisymmetric” and “asymmetric”?
- R is asymmetric iff R is antisymmetric and has no loops.

15

Properties of Relations

- **Definition:** A relation R on a set A is called **transitive** if whenever (a, b)∈R and (b, c)∈R, then (a, c)∈R for a, b, c∈A.

- Whenever there is a path that goes from a to b, then there is an edge (a, b) in the graph, then the graph represents a transitive relation.

- Are the following relation on {1, 2, 3} transitive?

R = {(1, 1), (1, 2), (2, 1), (3, 3)}

16

Combining Relations

- **Definition:** Let R be a relation from a set A to a set B and S a relation from B to a set C. The **composite** of R and S is the relation consisting of ordered pairs (a, c), where $a \in A$, $c \in C$, and for which there exists an element $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S$. We denote the composite of R and S by $\mathbf{S \circ R}$.

- If A = B = C, and S = R, then $R \circ R$ can be written as R^2.

- If R is represented by a graph, then (a, b) is in R^2 iff there is a path of length 2 from a to b.
- In general, (a, b) is in R^k iff there is a path of length k from a to b.

17

Combining Relations

- **Definition:** Let R be a relation on the set A. The powers R^k, k = 1, 2, 3, …, are defined inductively by
  - $R^1 = R$
  - $R^{k+1} = R^k \circ R$

- In other words: $R^k = R \circ R \circ \dots \circ R$ (k times the letter R)

- The relation $R^* = R^1 \cup R^2 \cup R^3 \cup \dots \cup R^{n-1}$, where n is the number of nodes, is called the transitive closure of R.

- To decide if (a, b) in R^*, we need to decide if there is a path from a to b in G = (A, R).

18

Combining Relations

- **Theorem:** The relation R on a set A is transitive if and only if $R^k \subseteq R$ for all positive integers k.

Remember the definition of transitivity:

- **Definition:** A relation R on a set A is called transitive if whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$ for a, b, $c \in A$.

- The composite of R with itself contains exactly these pairs (a, c).
- Therefore, for a transitive relation R, $R \circ R$ does not contain any pairs that are not in R, so $R \circ R \subseteq R$.
- Since $R \circ R$ does not introduce any pairs that are not already in R, it must also be true that $(R \circ R) \circ R \subseteq R$, and so on, so that $R^k \subseteq R$.

19

19

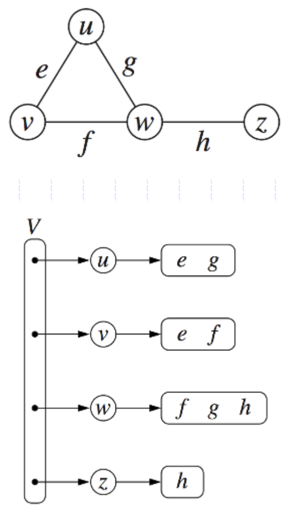
Equivalence Relations

- **Equivalence relations** are used to relate objects that are similar in some way.
- **Definition:** A relation on a set A is called an equivalence relation if it is reflexive, symmetric, and transitive.
- Two elements that are related by an equivalence relation R are called **equivalent**.
- The best representation of an equivalence relation is Sets.

20

Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices

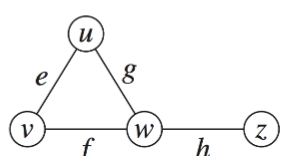


21

21

Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



		0	1	2	3
$u \rightarrow$	0		e	g	
$v \rightarrow$	1	e		f	
$w \rightarrow$	2	g	f		h
$z \rightarrow$	3			h	

e, f, g, h can be replaced by 1; blanks by 0.

22

22

Possible Quiz Question

Suppose direct graph G = (V, E) is represented by 0/1 adjacency matrix A and k is a positive integer.

- Let $B = A^k$. If B is the 0/1 adjacency matrix for another graph H over V, what is the relationship between H and G in terms of paths in G?
- How to compute A^k efficiently?

23

23

Graph Representations

```
Option 1:

Class Node
    String: Name
    Boolean: Visited
    List<Node>: Neighbors
    List<Integer>: Costs
End Node
```

```
Option 2:

Class Node
    String: Name
    Boolean: Visited
    List<Edge>: Edges
End Node

Class Edge
    Integer: Cost
    Node: toNode
    Node: fromNode
End Edge
```

24

Performance

(All bounds are big-oh running times, except for "Space")

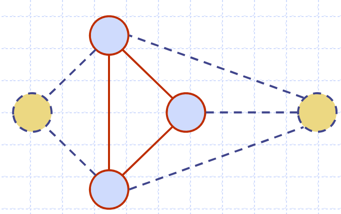
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	deg(v)	1

25

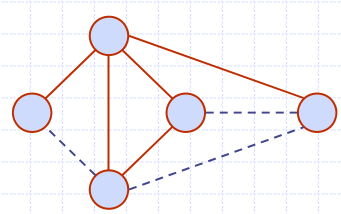
25

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



Subgraph



Spanning subgraph

26

26

Application: Web Crawlers

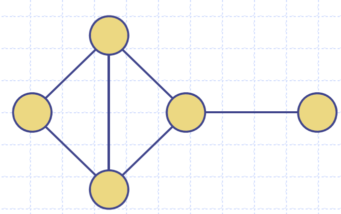
- A fundamental kind of algorithmic operation that we might wish to perform on a graph is **traversing the edges and the vertices** of that graph.
- A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.
- For example, a **web crawler**, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents.
- A traversal is efficient if it visits all the vertices and edges in linear time.

27

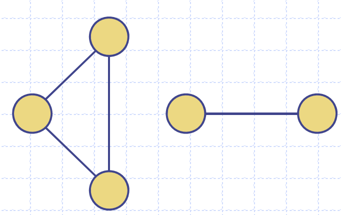
27

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Connected graph



Non connected graph with two connected components

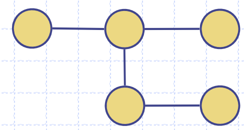
28

28

Trees and Forests

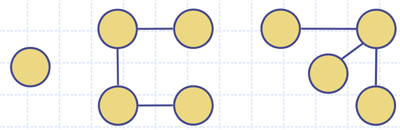
- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from rooted trees



Tree

- A forest is an undirected graph without cycles
- The connected components of a forest are trees



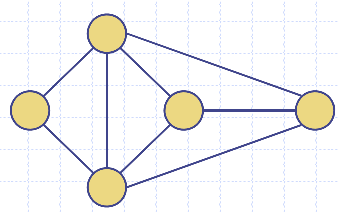
Forest

29

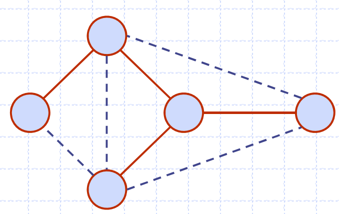
29

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

30

30

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search can be done iteratively or recursively, and the results are different if a node has multiple children.

31

31

Depth-First Traversal with Marking

```
DFS_recur(Node: node)
    <Process node>
    node.Visited = True
    for each edge in node.Edges
        if (not edge.toNode.Visited) then
            DFS_recur(edge.toNode)
            edge.toNode.parent = node
        end if
    end for
end DFS_recur

Complexity: O(n + m),
n and m are the numbers of nodes and edges, resp.
```

32

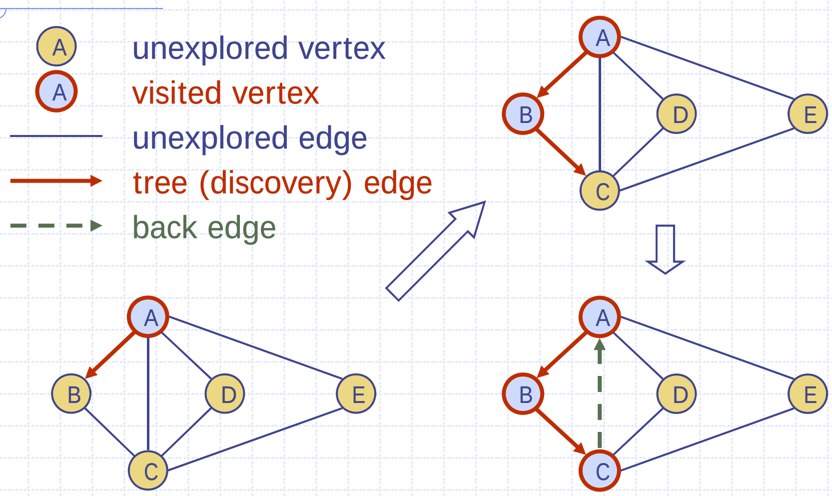
Depth-First Traversal with Time-Stamp

```
DFS_recur(Node: node)
    <Process node>
    node.StartTime = ++time  // time is global
    for each edge in Edges
        if (edge.toNode.StartTime == 0) then
            DFS_recur(edge.toNode)
            edge.toNode.parent = node
        end if
    end for
    node.FinishTime = ++time // optional
end DFS_recur
```

Color of a node: white if StartTime is undefined; gray if StartTime is defined but FinishTime is undefined; black if FinishTime is defined.

33

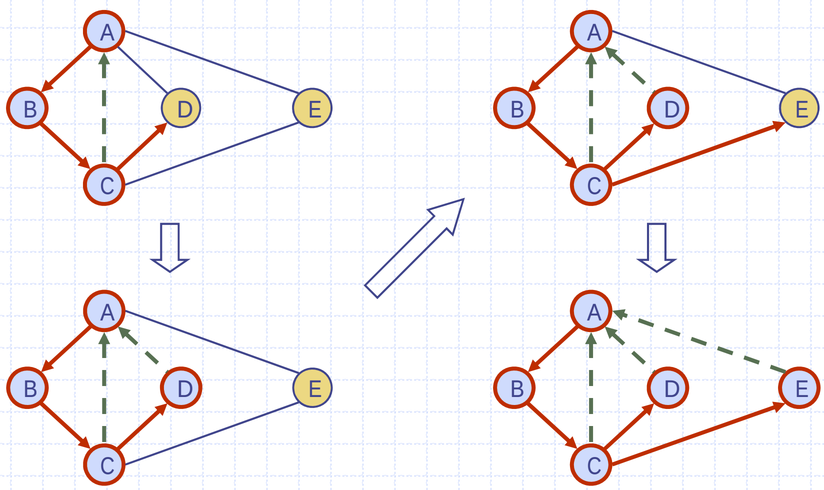
Example



34

34

Example (cont.)



35

35

Non-Recursive DFS

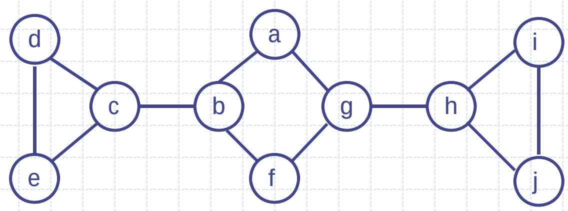
```
DepthFirstTraverse(Node: start_node)
    start_node.Visited = True   // Visit this node.
    // Make a stack and put the start node in it.
    Stack[Node]: stack;    stack.Push(start_node);
    // Repeat as long as the stack isn't empty.
    while not stack.IsEmpty() do
        Node node = stack.Pop() // Get the next node from the stack.
        for each edge in node.Edges     // Process the node's Edges.
            // if toNode hasn't been visited…
            if (not Edge.toNode.Visited) then
                // Mark the node as visited and may set StartTime
                Edge.toNode.Visited = True
                Edge.toNode.parent = node
                stack.Push(Edge.toNode) // Push the node onto the stack.
            end if
        end for   // may set FinishTime of node.
    end while // Continue processing the stack until empty
end DepthFirstTraverse
```

3 stages of a node: not visited (white), in stack (grey), exited stack (black)

36

Depth-First Search

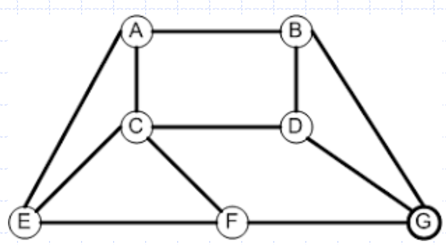
- Starting from a, give the finishing time for each vertex when the recursive DFS is used.
- Repeat the above exercise when non-recursive DFS is used.



37

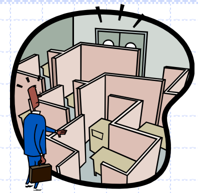
Possible Quiz Question

- Starting from A, using Depth-First Search to give the finishing time for each vertex when the recursive DFS is used. Neighbors of any vertex are listed in alphabet order.
- Repeat the above exercise when non-recursive DFS is used.

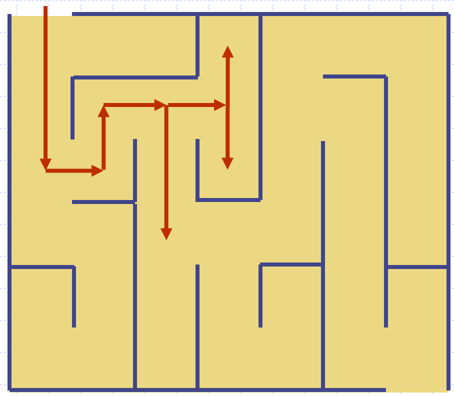


38

DFS and Maze Traversal



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



39

39

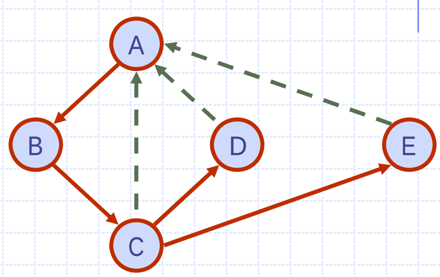
Properties of DFS

Property 1

DFS(*G, v*) visits all the vertices and edges in the connected component of *v*

Property 2

The tree (discovery) edges labeled by *DFS*(*G, v*) form a spanning tree of the connected component of *v* (Recursive and Non-recursive DFS produce different trees, and different start and finish times).



40

40

The General DFS Algorithm

- Perform a DFS from each unexplored vertex, and produce a forest of DFS trees:

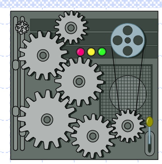
Algorithm DFS(G):
  Input: A graph G
  Output: A labeling of the vertices in each connected component of G as explored

  Initially label each vertex in v as unexplored
  for each vertex, v, in G **do**
    if v is unexplored **then**
      DFS(G, v)

41

41

Analysis of DFS



- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED initially
  - once as VISITED
- Each edge in an undirected graph is seen twice
  - once as TREE (i.e., DISCOVERY edge)
  - once as BACK
- Each edge in a directed graph is seen once
  - as TREE, BACK, CROSS, or FORWARD edges
- DFS runs in $O(n + m)$ time if the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

42

42

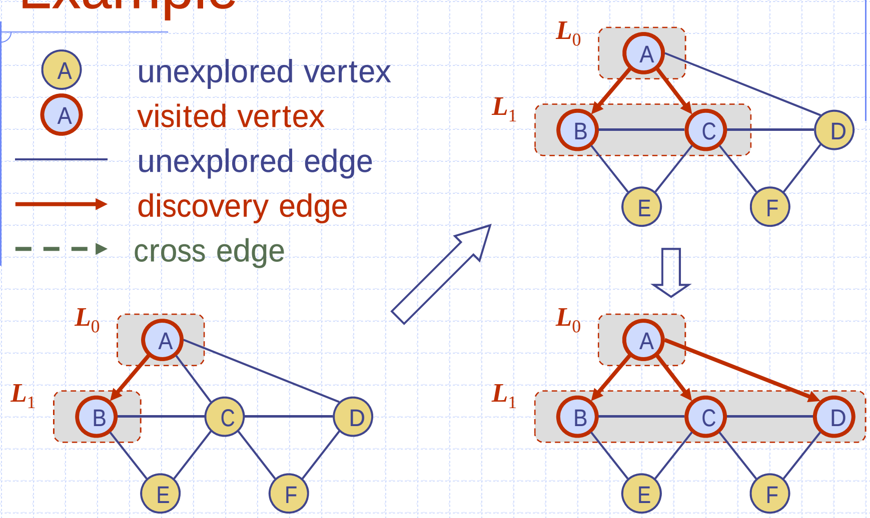
Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

43

43

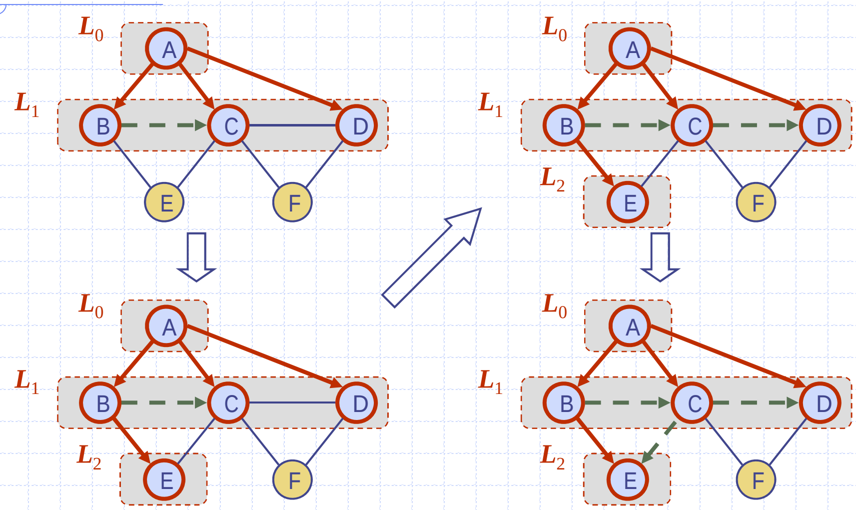
Example



44

44

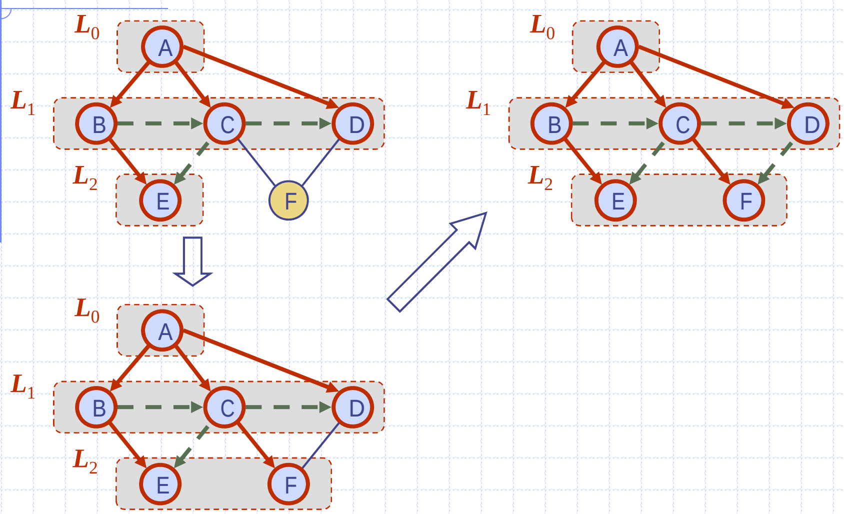
Example (cont.)



45

45

Example (cont.)



46

46

Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

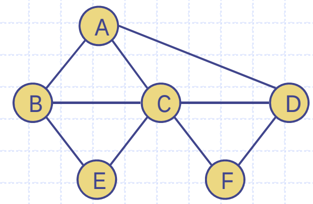
Property 2

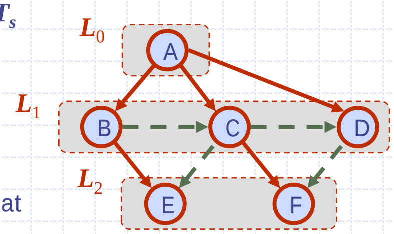
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges





47

47

BFS Algorithm

- The algorithm uses "levels" L_i and a mechanism for setting and getting "labels" of vertices and edges.

Algorithm BFS(G, s):
  Input: A graph G and a vertex s of G
  Output: A labeling of the edges in the connected component of s as discovery edges and cross edges

```
Create an empty list, L_0
Mark s as explored and insert s into L_0
i ← 0
while L_i is not empty do
    create an empty list, L_{i+1}
    for each vertex, v, in L_i do
        for each edge, e = (v, w), incident on v in G do
            if edge e is unexplored then
                if vertex w is unexplored then
                    Label e as a discovery edge
                    Mark w as explored and insert w into L_{i+1}
                else
                    Label e as a cross edge
    i ← i + 1
```

48

48

Breadth-First Traversal

```
BreadthFirstTraverse(Node: start_node)
    start_node.Visited = True   // Visit this node.
    // Make a stack and put the start node in it.
    Queue[Node]: queue;    queue.add(start_node);
    // Repeat as long as the stack isn't empty.
    While <queue isn't empty>
        Node node = queue.remove() // Get the next node from the queue.
        // Process the node's Edges.
        For each edge In node.Edges
            // if toNode hasn't been visited…
            If (Not Edge.toNode.Visited) Then
                // Mark the node as visited and set StartTime
                Edge.toNode.Visited = True
                // Push the node onto the stack.
                stack.Push(Edge.toNode)
            End If
        End for   // Set FinishTime of node.
    Loop // Continue processing the queue until empty
End DepthFirstTraverse
```

3 stages of a node: not visited (white), in queue (grey), exited queue (black)

49

Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

50

50

Applications

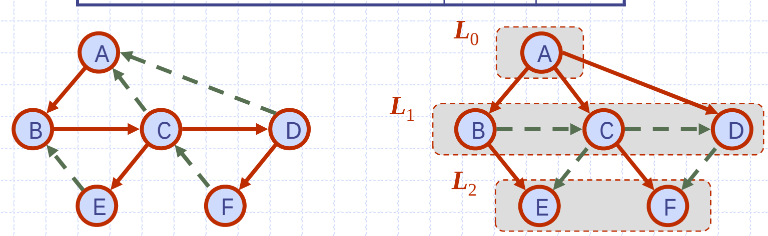
- We can use the BFS traversal algorithm, for a graph G, to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

51

51

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS BFS

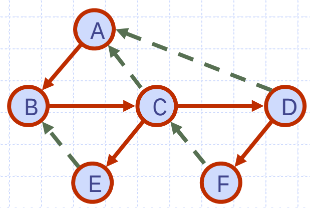
52

52

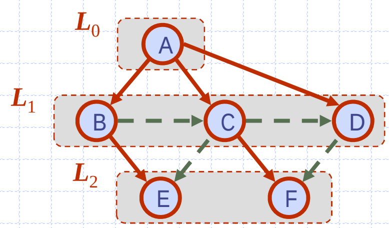
DFS vs. BFS (cont.)

Back edge (v,w)
- w is an ancestor of v in the DFS tree

Cross edge (v,w)
- w is in the same level as v or in the next level



DFS



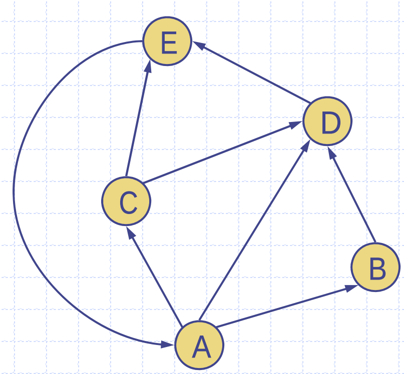
BFS

53

53

Recall: Digraphs

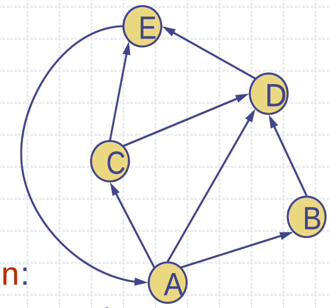
- A digraph is a shorthand for directed graph whose edges are all directed
- Applications
  - one-way streets
  - flights
  - task scheduling



54

54

Digraph Properties



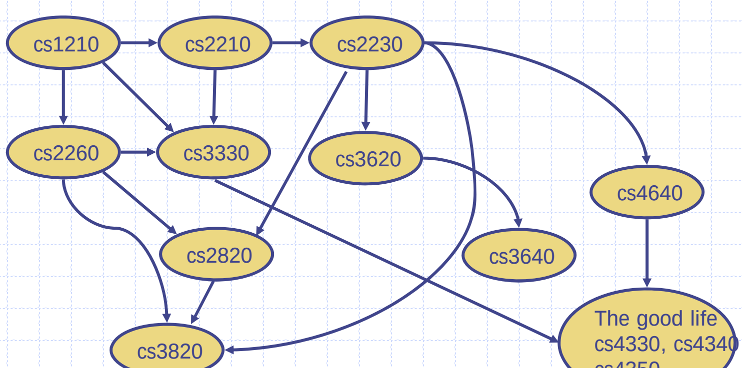
- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a, b) goes from a to b, but not b to a
- If G is simple, $m \leq n \cdot (n - 1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

55

55

Digraph Application

- Scheduling: edge (a,b) means task a must be completed before b can be started

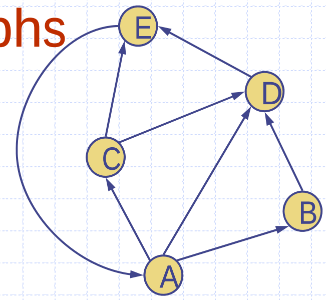


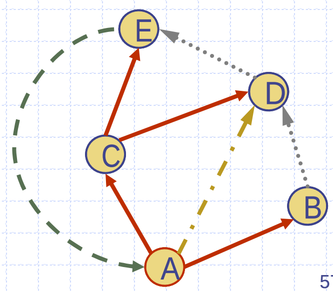
56

56

DFS for Directed Graphs

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - tree edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex *s* determines the vertices reachable from *s*



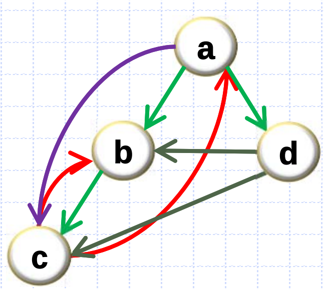


57

57

Edge classification by DFS

Tree edges: parent to child
Forward edges: descendent to ancestor
Back edges: descendent to ancestor
Cross edges: none of above

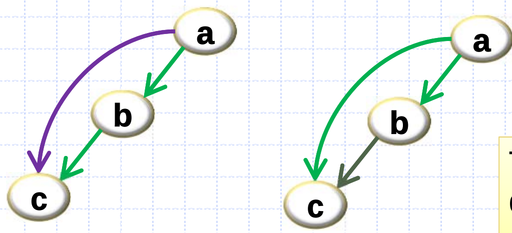


The edge classification depends on the particular DFS tree!

58

Edge classification by DFS

Tree edges: parent to child
Forward edges: descendent to ancestor
Back edges: descendent to ancestor
Cross edges: none of above



Both are valid

The edge classification depends on the particular DFS tree!

59

Edge classification by DFS

Edge (**u**,**v**) of **G** is classified as:

(1) **Tree** edge iff **u** discovers **v** during the DFS: **P**[**v**] = **u**

i.e., v.StartTime is undefined (v is white).

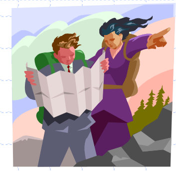
If (**u**,**v**) is NOT a tree edge then it is:

(2) **Back** edge iff **u** is a descendant of **v** in the DFS tree
i.e., v.FinishTime is undefined (v is grey).

(3) **Forward** edge iff **u** is an ancestor of **v** in the DFS tree
i.e., v.FinishTime is defined (v is black) and
u.StartTime < v.StartTime and P[v] != u

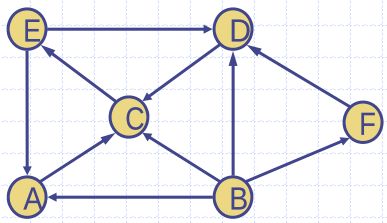
(4) **Cross** edge iff **u** is neither an ancestor nor a descendant of **v**
i.e. v.FinishTime is defined (v is black) and
u.StartTime > v.FinishTime (v is black).

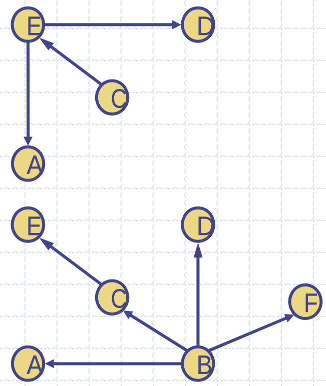
60



Reachability

- DFS tree rooted at v: vertices reachable from v via directed paths





61

61

DAGs and back edges

- Can there be a **back** edge in a DFS on a Directed Acyclic Graph (DAG)?
- NO! Back edges form a cycle!
- A graph **G** is a DAG <=> there is no back edge classified by DFS(**G**)

62

DAGs and Topological Ordering

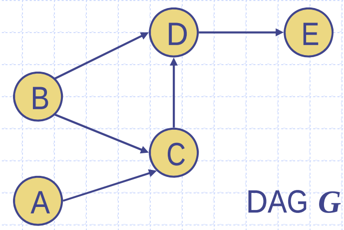
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $v_1, \ldots, v_n$

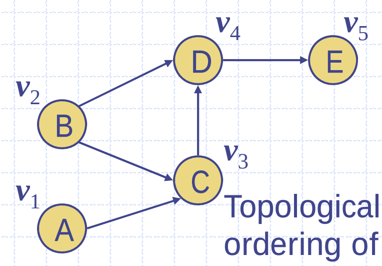
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG



DAG G

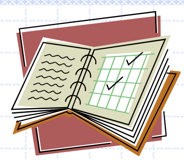


Topological ordering of G

63

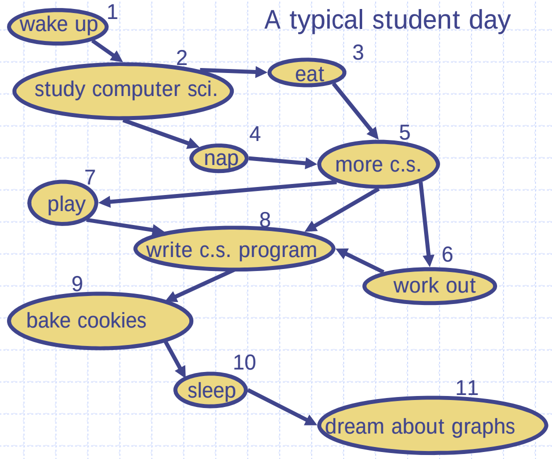
63

Topological Sorting



- Number vertices, so that (u,v) in E implies u < v

A typical student day



64

64

Algorithm for Topological Sorting

- Note: This algorithm is different than the one in the book

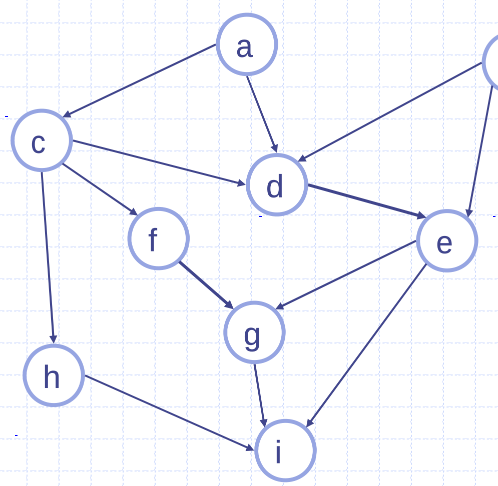
```
Algorithm TopologicalSort(G)
    H ← G          // Temporary copy of G
    t ← 1
    while H is not empty do
        Let v be a vertex with no ingoing edges
        Label v ← t
        t ← t + 1
        Remove v from H
```

- Running time: O(n + m)

65

65

Topological Sorting Example



66

66

Topological sorting with DFS

- Simulate the algorithm by using depth-first search
- O(n+m) time.

Algorithm *topologicalDFS*(*G*)
  Input dag *G*
  Output topological ordering of *G*
  $t \leftarrow$ *G.numVertices*()
  for all $u \in$ *G.vertices*()
    setLabel(*u, UNEXPLORED*)
  for all $v \in$ *G.vertices*()
    if *getLabel*(v) = *UNEXPLORED*
      topologicalDFS(*G, v*)

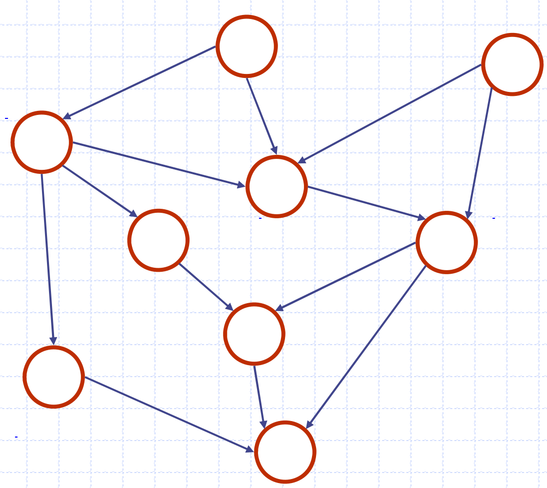
Algorithm *topologicalDFS*(*G, v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the vertices of *G* in the connected component of *v*
  setLabel(*v, VISITED*)
  for all $e \in$ *G.outEdges*(v)
    { outgoing edges **}**
    $w \leftarrow$ *theOtherEnd*(*v, e*)
    if *getLabel*(w) = *UNEXPLORED*
      { *e* is a discovery edge **}**
      topologicalDFS(*G, w*)
    else
      { *e* is a forward or cross edge **}**
  Label *v* with topological number t
  $t \leftarrow t - 1$

Topological number = ***n*** **–** finishTime + 1

67

67

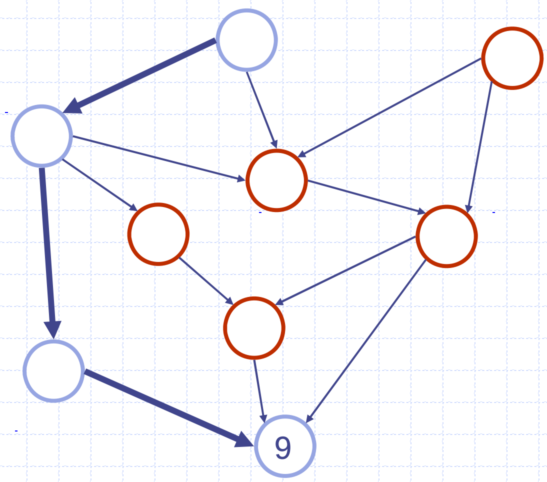
Topological Sorting Example



68

68

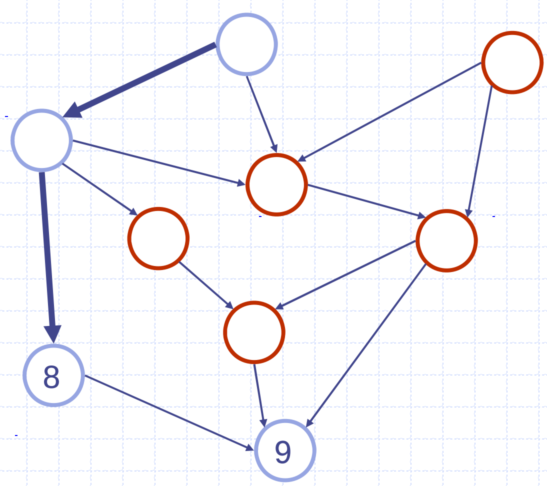
Topological Sorting Example



69

69

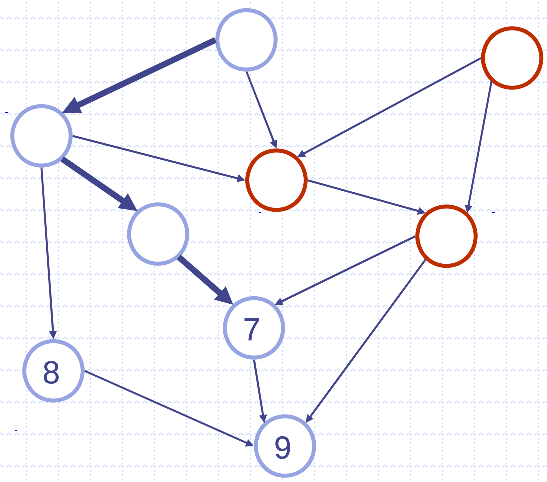
Topological Sorting Example



70

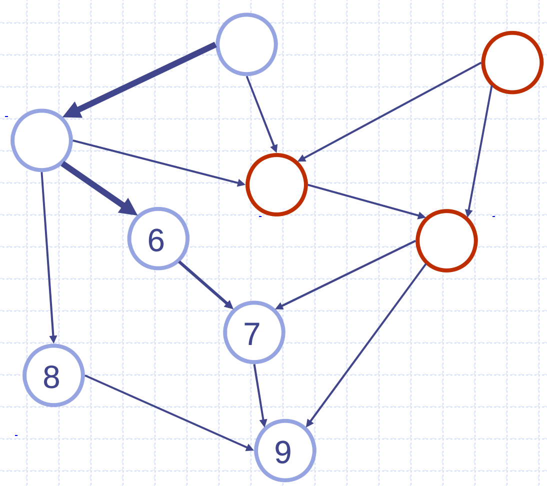
70

Topological Sorting Example



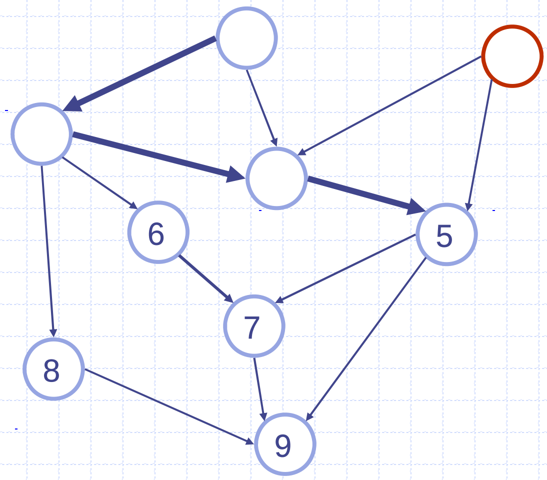
71

Topological Sorting Example



72

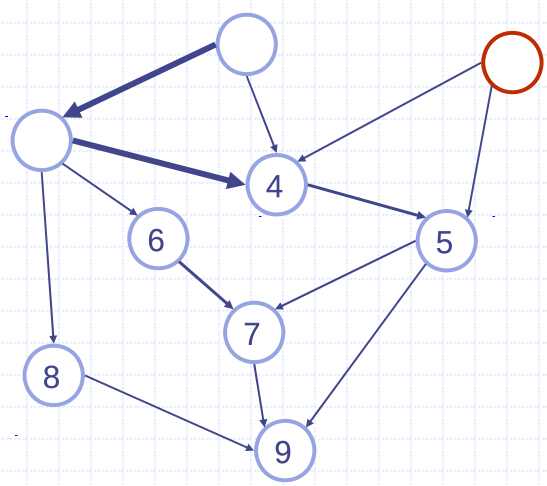
Topological Sorting Example



73

73

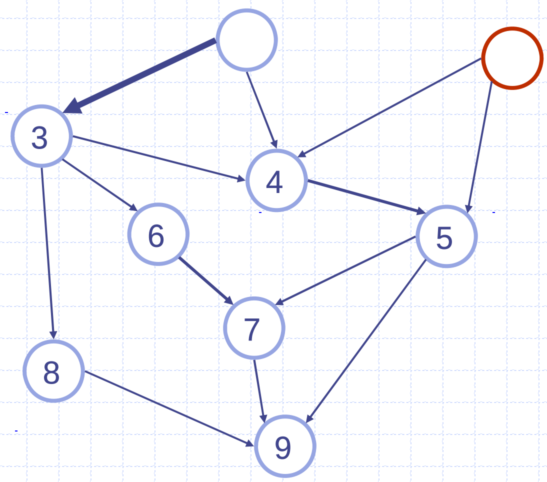
Topological Sorting Example



74

74

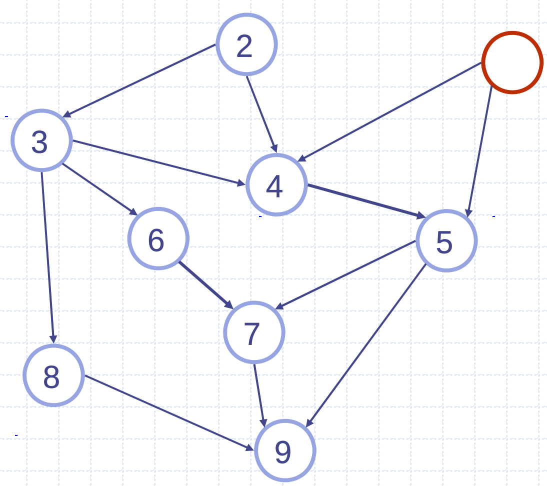
Topological Sorting Example



75

75

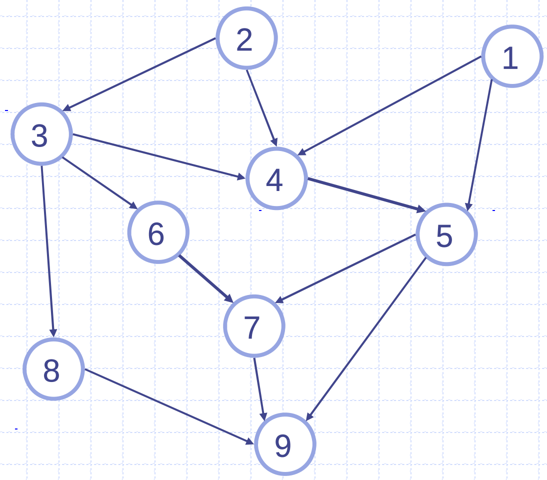
Topological Sorting Example



76

76

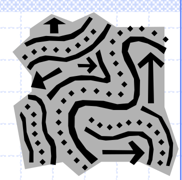
Topological Sorting Example



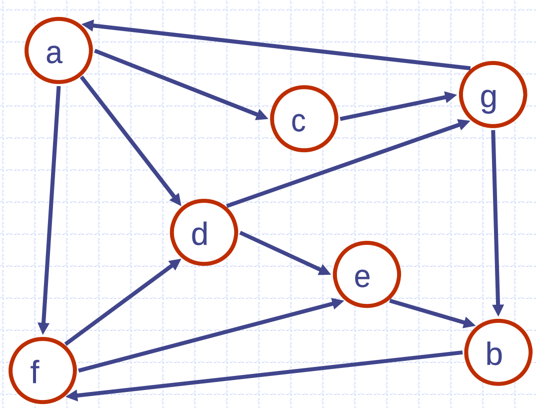
77

77

Strong Connectivity



- Each vertex can reach all other vertices



78

78

Application: Networking

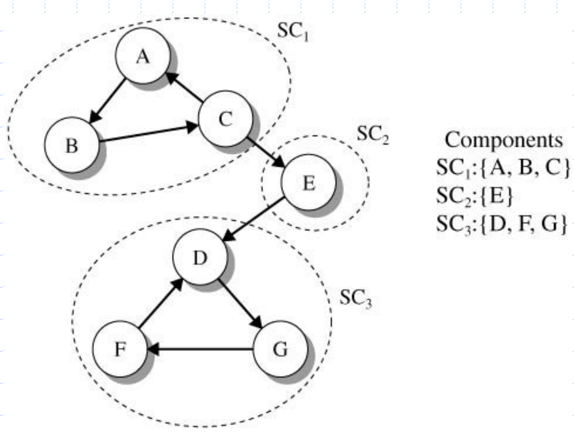
- A computer network can be modeled as a graph, where vertices are routers and edges are network connections between edges.
- A router can be considered **critical** if it can disconnect the network for that router to fail.
- It would be nice to identify which routers are critical.
- We can do such an identification by solving the biconnected components problem.

79

79

Strongly Connected Components

- Any directed graph can be partitioned into a unique set of strong components.



Strongly connected components in a directed graph.

80

Strongly Connected Components

- The algorithm for finding the strong components of a directed graph G uses the transpose of the graph.
  - The transpose G^T has the same set of vertices V as graph G, but a new edge set consisting of the edges of G but with the opposite direction.

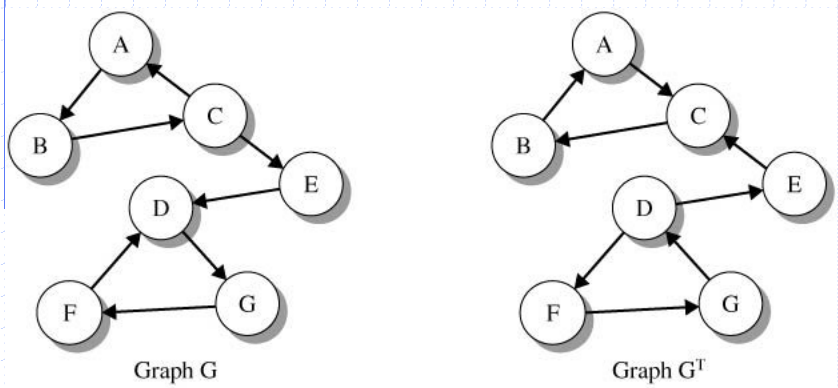
81

Strongly Connected Components

- Execute the depth-first search dfs() for the graph G which creates the list dfsList consisting of the vertices in G in the reverse order of their finishing times.
- Generate the transpose graph G^T.
- Using the order of vertices in dfsList, make repeated calls to dfs() for vertices in G^T. The list returned by each call is a strongly connected component of G.

82

Strongly Connected Components



83

Running Time of strongComponents()

- Recall that the depth-first search has running time O(V+E), and the computation for G^T is also O(V+E). It follows that the running time for the algorithm to compute the strong components is O(V+E).

84

Strongly Connected Components

dfsList: [A, B, C, E, D, G, F]

Using the order of vertices in dfsList, make successive calls to dfs() for graph G^T

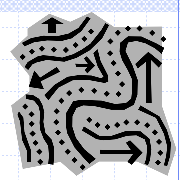
Vertex A: dfs(A) returns the list [A, C, B] of vertices reachable from A in G^T.

Vertex E: The next unvisited vertex in dfsList is E. Calling dfs(E) returns the list [E].

Vertex D: The next unvisited vertex in dfsList is D; dfs(D) returns the list [D, F, G] whose elements form the last strongly connected component..

85

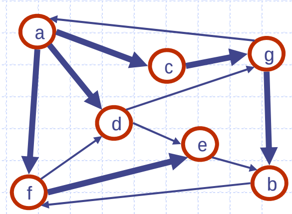
Strong Connectivity Algorithm



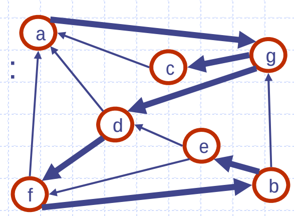
- Pick a vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G' be G with edges reversed
- Perform a DFS from v in G'
  - If there's a w not visited, print "no"
  - Else, print "yes"

- Running time: O(n+m)

G:



G' :

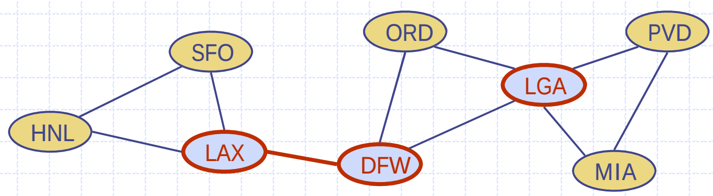


86

86

Separation Edges and Vertices

- Definitions
  - Let *G* be a connected graph
  - A separation edge (bridge edge) of *G* is an edge whose removal disconnects *G*
  - A separation (articulation) vertex of *G* is a vertex whose removal disconnects *G*
- Example
  - DFW, LGA and LAX are separation vertices
  - (DFW,LAX) is a separation edge



87

87

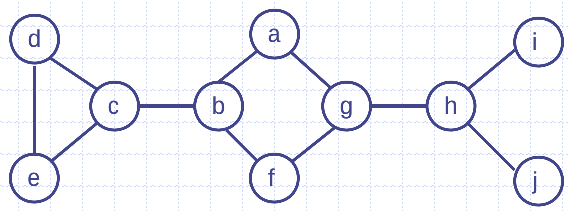
Finding Articulation (Separation) Points in a Graph

- A vertex *v* in an undirected graph *G* with more than two vertices is called an **articulation point** if there exist two vertices *u* and *w* different from *v* such that any path between *u* and *w* must pass through *v*.
- If *G* is connected, the removal of *v* and its incident edges will result in a disconnected subgraph of *G*.
- A graph is called biconnected if it is connected and has neither articulation points nor points of degree 1.

88

Finding Articulation (Separation) Points in a Graph

- Example: c, b, g, h are articulation points.



89

Finding Articulation (Separation) Points in a Graph

- To find the set of articulation points, we perform a depth-first search traversal on G.
- During the traversal, we maintain two labels with each vertex $v \in V$: $\alpha[v]$ and $\beta[v]$.
- $\alpha[v]$ is simply v's start time in the depth-first search algorithm. $\beta[v]$ is initialized to $\alpha[v]$, but may change later on during the traversal.

90

Finding Articulation (Separation) Points in a Graph

- For each vertex v visited, we let $\beta[v]$ be the minimum of the following:
  - $\alpha[v]$
  - $\alpha[u]$ for each vertex u such that (v, u) is a *back edge*
  - $\beta[w]$ for each vertex w such that (v, w) is a *tree edge*

Thus, $\beta[v]$ is the smallest α of those points that v can reach through back edges or tree edges.

91

Finding Articulation (Separation) Points in a Graph

The articulation points are determined as follows:

- The root is an articulation point if and only if it has two or more children in the depth-first search tree.
- A vertex v other than the root is an articulation point if and only if v has a child w with $\beta[w] \geq \alpha[v]$.

92

Finding Articulation (Separation) Points in a Graph

- **Input**: A connected undirected graph $G=(V, E)$;
- **Output**: Boolean array $artpoint[1 \ldots n]$ indicates the articulation points of G, if any.

- 1. for each vertex $v \in V$
- 2. { $\alpha[v] \leftarrow 0$; artpoint$[v] \leftarrow$ false; }
- 3. $time \leftarrow 0$; $rootdegree \leftarrow 0$; $root \leftarrow s$;
- 4. $dfs2(s)$; // s is the start vertex

93

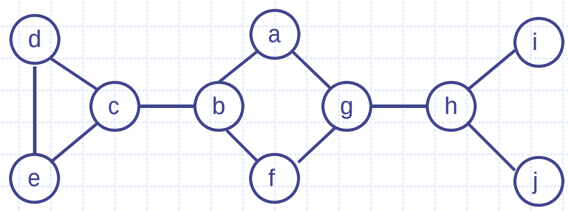
Finding Articulation (Separation) Points in a Graph

- $dfs2\,(v)$
- 2. $\alpha[v] \leftarrow \beta\,[v] \leftarrow$ ++$time$; // $\alpha[v]$ is the start time
- 3. **for** each edge (v, w) **in** v.Edges
- 4.   **if** $(\alpha[w]$ == 0) **then** // (v, w) is a tree edge
- 5.   p$[w] \leftarrow v$; $dfs2(w)$;
- 6.   **if** (v == $root$) **then** // v is the root
- 7.   ++$rootdegree$;
- 8.   **if** $rootdegree > 1$ **then** $artpoint\,[v] \leftarrow$true;
- 9.   **else** // v is not the root;
- 10. $\beta\,[v] \leftarrow \min\{\beta\,[v], \beta\,[w]\}$;
- 11.   **if** $\beta\,[w] \geq \alpha\,[v]$ **then** $artpoint\,[v] \leftarrow$true;
- 12.   **end if**;
- 13.   **else if** (p$[v]$!= w) // (v, w) is a back edge
- 14.   **then** $\beta\,[v] \leftarrow \min\{\beta\,[v], \alpha\,[w]\}$;
- 15.   **end if**;
- 16. **end for**;

94

Finding Articulation (Separation) Points in a Graph

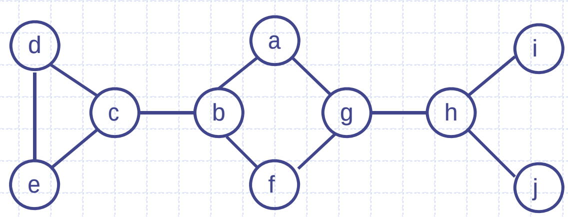
- Example:



95

How to find separation edges (bridges) in a Graph

- Example: (c, b), (g, h), (h, i), (h, j) are bridges.

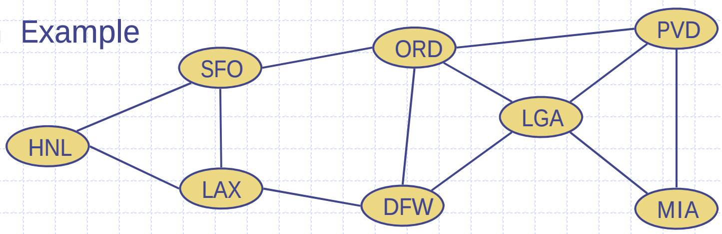


- An edge (u, v) is a bridge if u and v are either separation (articulation) points or degree 1.

96

Biconnected Graph

- Equivalent definitions of a biconnected graph ***G***
  - Graph ***G*** has no separation edges and no separation vertices.
  - For any two vertices ***u*** and ***v*** of ***G***, there are two disjoint simple paths between ***u*** and ***v*** (i.e., two simple paths between ***u*** and ***v*** that share no other vertices or edges).
  - For any two vertices ***u*** and ***v*** of ***G***, there is a simple cycle containing ***u*** and ***v***.
- Example

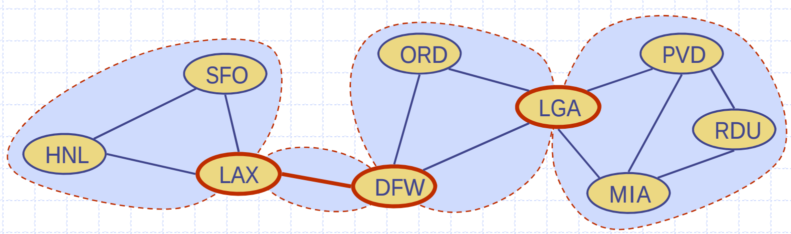


97

97

Biconnected Components

- Biconnected component of a graph ***G***
  - A maximal biconnected subgraph of ***G***, or
  - A subgraph consisting of a separation edge of ***G*** and its end vertices
- Interaction of biconnected components
  - An edge belongs to exactly one biconnected component
  - A nonseparation vertex belongs to exactly one biconnected component
  - A separation vertex belongs to two or more biconnected components
- Example of a graph with four biconnected components



98

Equivalence Classes

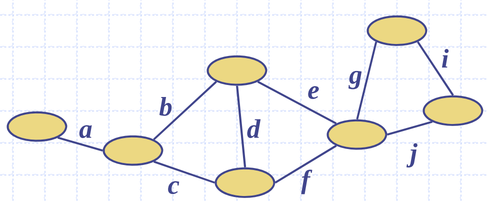
- An equivalence relation R on S induces a partition of the elements of S into equivalence classes.
- For undirected graph, connectivity is an equivalence relation on points, which generate classes of points (components).
  - Let V be the set of vertices of an undirected graph G
  - Define the relation
    $C = \{ (v,w) \in V\times V$ such that G has a path from v to $w\}$
  - Relation C is an equivalence relation
  - The equivalence classes of relation C are the vertices in each connected component of graph G
- For directed graph, strong connectivity is an equivalence classes on points (strongly connected components).

99

99

Biconnectivity Relation

- Edges e and f of connected graph G are biconnected if
  - $e = f$, or
  - G has a simple cycle containing e and f



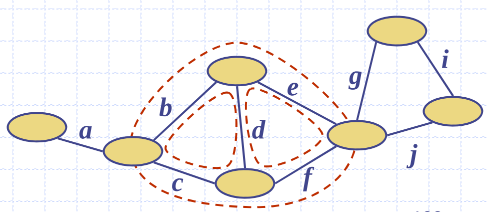
Theorem:

The biconnectivity relation on the edges of a graph is an equivalence relation

Proof Sketch:

- The reflexive and symmetric properties follow from the definition
- For the transitive property, consider two simple cycles sharing an edge

Equivalence classes of biconnected edges: $\{a\}$ $\{b, c, d, e, f\}$ $\{g, i, j\}$

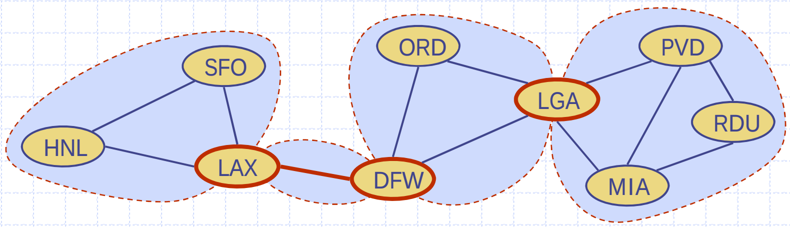


100

100

Biconnected Components

- The biconnected components of a graph ***G*** are the equivalence classes of edges with respect to the biconnectivity relation
- A biconnected component of ***G*** is the subgraph of ***G*** induced by an equivalence class of linked edges
- A separation edge is a single-element equivalence class of linked edges
- A separation vertex has incident edges in at least two distinct equivalence classes of linked edge



101

101