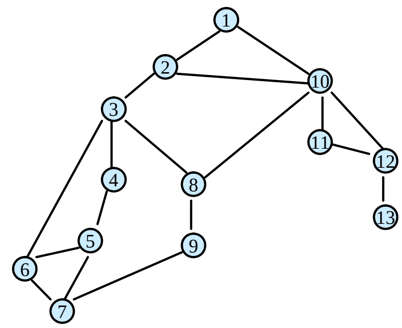
CSE 421: Introduction to Algorithms

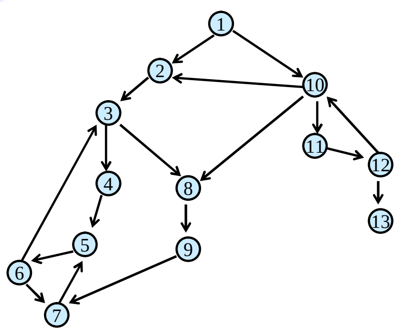
Graphs & Graph Traversal

Winter 2003
Paul Beame

Undirected Graph G = (V,E)



Directed Graph G = (V,E)

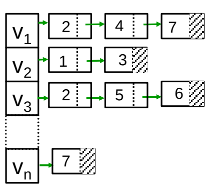


Representing Graph G=(V,E) n vertices, m edges

- Vertex set $V=\{v_1,\ldots,v_n\}$
- Adjacency Matrix **A**
  - $A[i,j]=1$ iff $(v_i,v_j) \in E$
  - Space is n^2 bits
- Advantages:
  - $O(1)$ test for presence or absence of edges.
  - compact in packed binary form for large **m**
- Disadvantages:
  - inefficient for sparse graphs

Representing Graph G=(V,E) n vertices, m edges

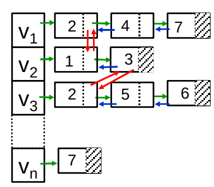
- Adjacency List:
  - $O(n+m)$ words
  - $O(\log n)$ bits each



- Advantages:
  - Compact for sparse graphs

Representing Graph G=(V,E) n vertices, m edges

- Adjacency List:
  - $O(n+m)$ words
  - $O(\log n)$ bits each



- Back- and cross pointers more work to build, but allow easier traversal and deletion of edges
  - usually assume this format

Graph Traversal

- Learn the basic structure of a graph
- Walk from a fixed starting vertex s to find all vertices reachable from s
- Three states of vertices
  - **unvisited**
  - **visited**
  - **fully-explored**

7

Generic Graph Traversal Algorithm

Find: set **R** of vertices reachable from **s Î V**

Reachable(**s**):
  R ¬ {**s**}
  While there is a (**u,v**) Î **E** where **u** Î **R** and **v** Ï **R**
    Add **v** to **R**

8

Generic Traversal Always Works

- Claim: At termination **R** is the set of nodes reachable from **s**
- Proof
  - ⊆: For every node **v** Î **R** there is a path from s to v
  - ⊇: Suppose there is a node **w** Ï **R** reachable from s via a path **P**
    - Take first node **v** on **P** such that **v** Ï **R**
    - Predecessor **u** of **v** in **P** satisfies
      - **u Î R**
      - **(u,v) Î E**
    - But this contradicts the fact that the algorithm exited the while loop.

9

Breadth-First Search

- Completely explore the vertices in order of their distance from s
- Naturally implemented using a queue

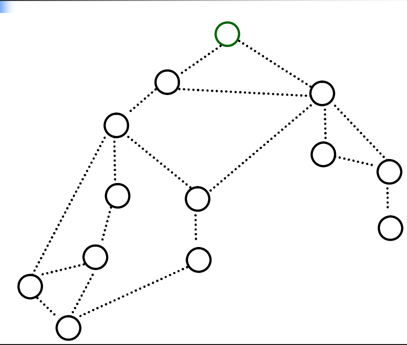
10

BFS(s)

Global initialization: mark all vertices "unvisited"
BFS(**s**)
  mark **s** "visited"; R←{s}; layer L_0←{s}
  while $\mathbf{L_i}$ not empty
    $\mathbf{L_{i+1}}$ ¬ Æ
    For each **u** Î $\mathbf{L_i}$
      for each edge {**u**,**v**}
        if (**v** is "unvisited")
          mark **v** "visited"
          Add **v** to set **R** and to layer $\mathbf{L_{i+1}}$
    mark **u** "fully-explored"

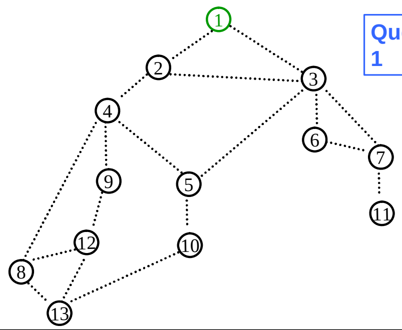
11

BFS(s)



12

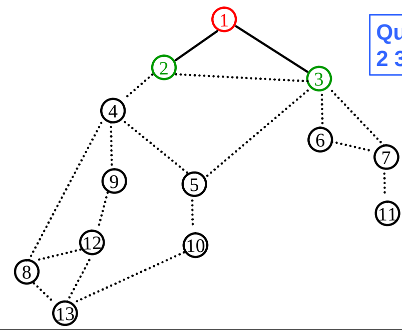
BFS(s)



Queue:
1

13

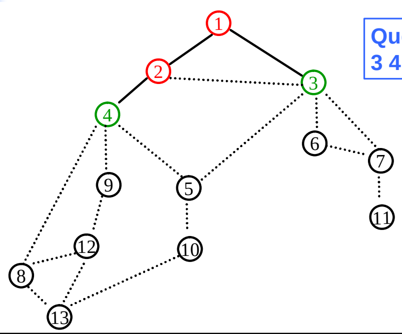
BFS(s)



Queue:
2 3

14

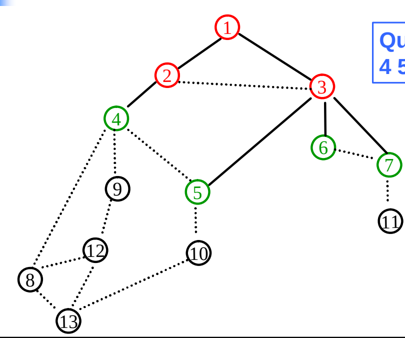
BFS(s)



Queue:
3 4

15

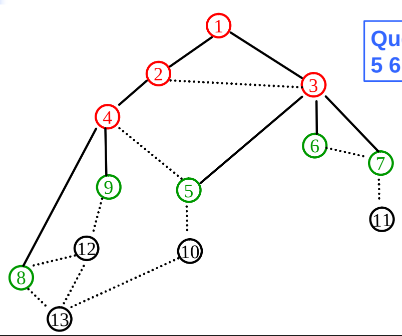
BFS(s)



Queue:
4 5 6 7

16

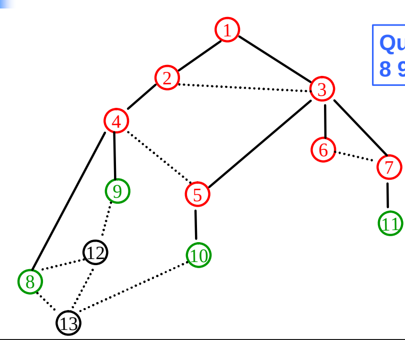
BFS(s)



Queue:
5 6 7 8 9

17

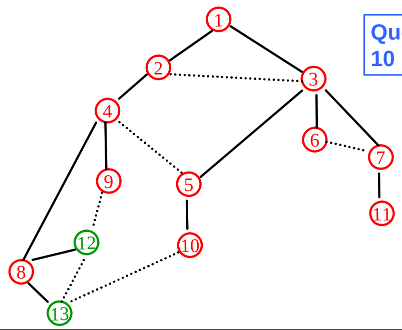
BFS(s)



Queue:
8 9 10 11

18

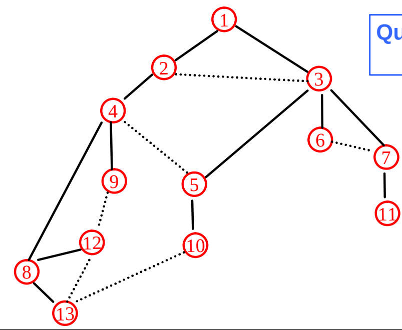
BFS(s)



Queue:
10 11 12 13

19

BFS(s)



Queue:

20

BFS analysis

- Each edge is explored once from each end-point (at most)
- Each vertex is discovered by following a different edge
- Total cost **O(m)** where **m**=# of edges

21

Properties of BFS(v)

- BFS(s) visits **x** if and only if there is a path in **G** from **s** to **x**.
- Edges followed to undiscovered vertices define a **tree**
  - "breadth first spanning tree" of **G**
- Layer **i** in this tree, L_i
  - those vertices **u** such that the shortest path in **G** from the root **s** is of length **i**.
- On undirected graphs
  - All non-tree edges join vertices on the same or adjacent layers

22

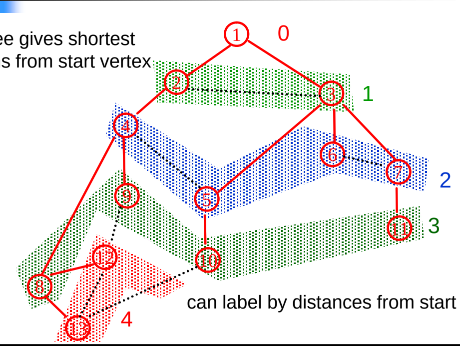
Properties of BFS

- On undirected graphs
  - All non-tree edges join vertices on the same or adjacent layers
  - Suppose not
    - Then there would be vertices (x,y) such that **x** Î L_i and **y** Î L_j and **i<j-1**
    - Then, when vertices incident to **x** are considered in BFS **y** would be added to L_{i+1} and not to L_j

23

BFS Application: Shortest Paths

Tree gives shortest paths from start vertex



can label by distances from start

24

Graph Search Application: Connected Components

- Want to answer questions of the form:
  - Given: vertices **u** and **v** in **G**
  - Is there a path from **u** to **v**?
- Idea: create array **A** such that
  A[**u**] = smallest numbered vertex that is connected to **u**
  - question reduces to whether **A[u]=A[v]**?

Q: Why not create an array **Path[u,v]**?

Graph Search Application: Connected Components

- initial state: all **v** unvisited
  for **s**←**1** to **n** do
  if state(**s**) ≠ "fully-explored" then
  BFS(**s**): setting **A[u]** ←**s** for each **u** found
  (and marking **u** visited/fully-explored)
  endif
  endfor
- Total cost: **O(n+m)**
  - each vertex is touched once in this outer procedure and the edges examined in the different BFS runs are disjoint
  - works also with Depth First Search

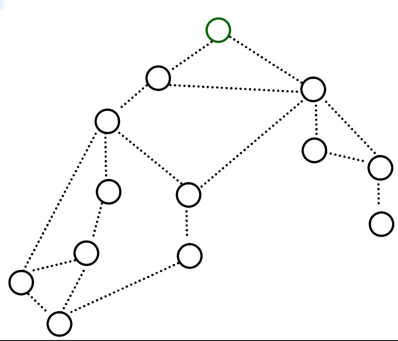
Depth-First Search

- Follow the first path you find as far as you can go
- Back up to last unexplored edge when you reach a dead end, then go as far you can
- Naturally implemented using recursive calls or a stack

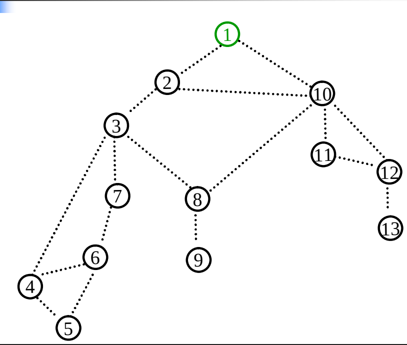
DFS(u) – Recursive version

Global Initialization: mark all vertices "unvisited"
DFS(**u**)
mark **u** "visited" and add **u** to **R**
for each edge {**u**,**v**}
if (**v** is "unvisited")
DFS(**v**)
end for
mark **u** "fully-explored"

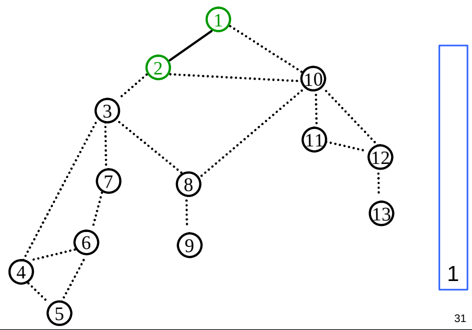
DFS(u)



DFS(u)

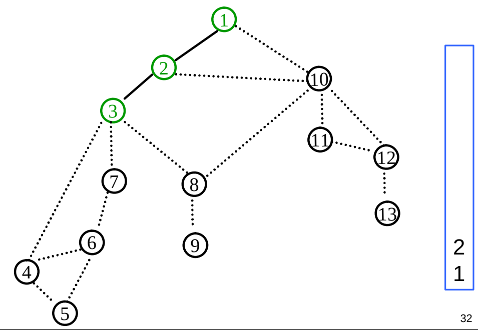


DFS(u)



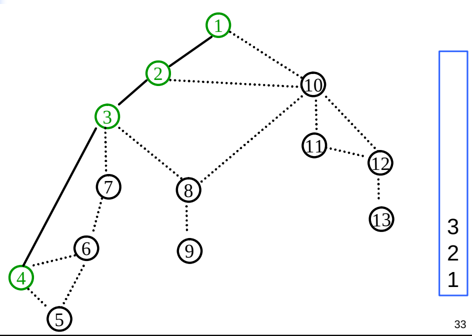
31

DFS(u)



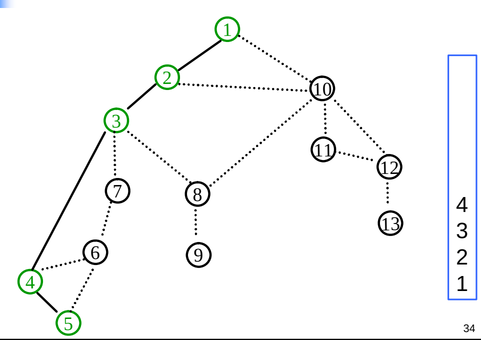
32

DFS(u)



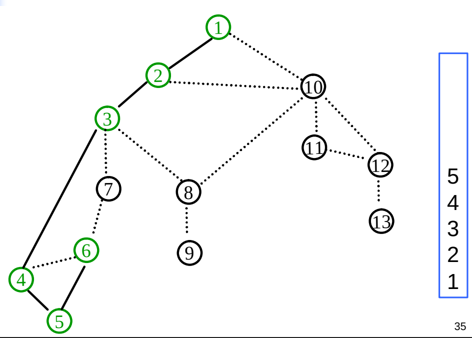
33

DFS(u)



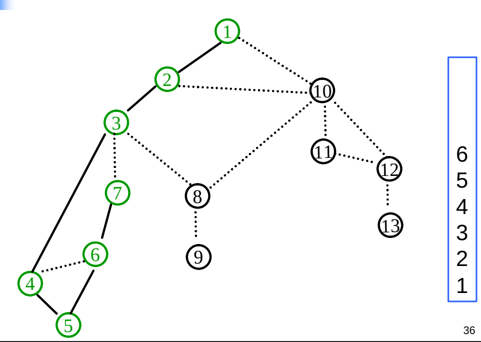
34

DFS(u)



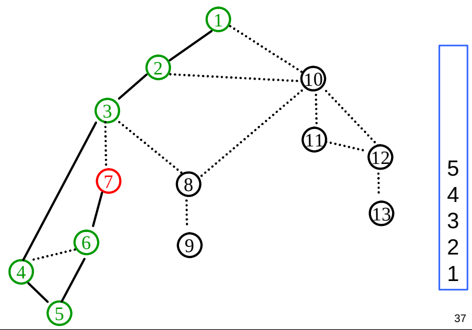
35

DFS(u)



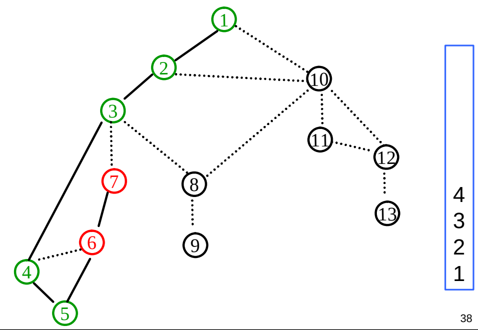
36

DFS(u)



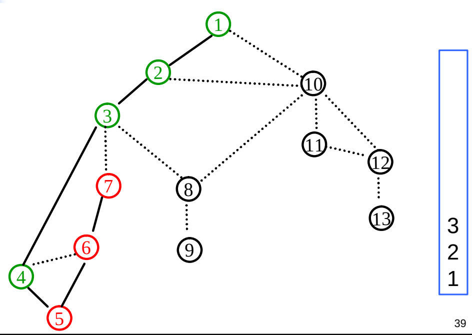
37

DFS(u)



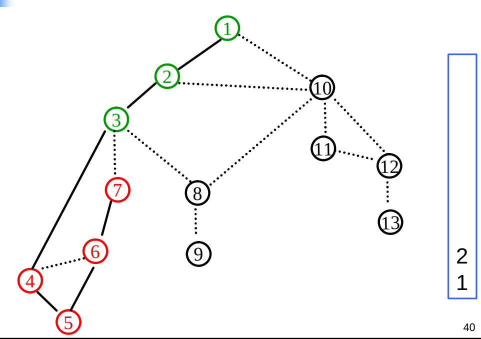
38

DFS(u)



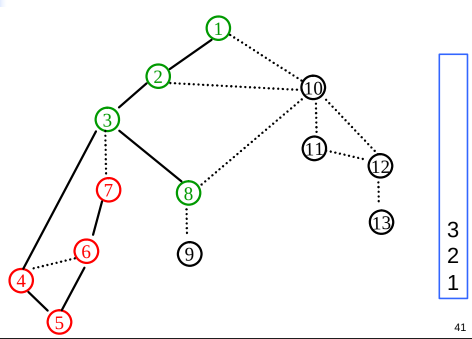
39

DFS(u)



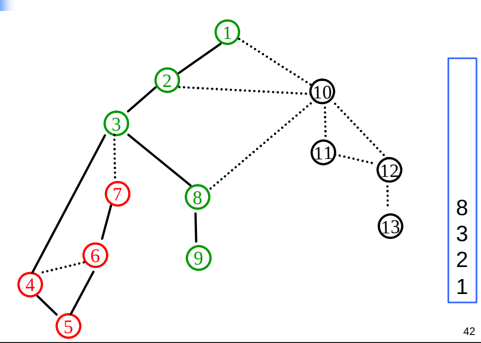
40

DFS(u)



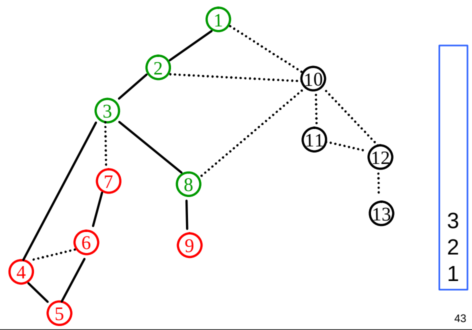
41

DFS(u)

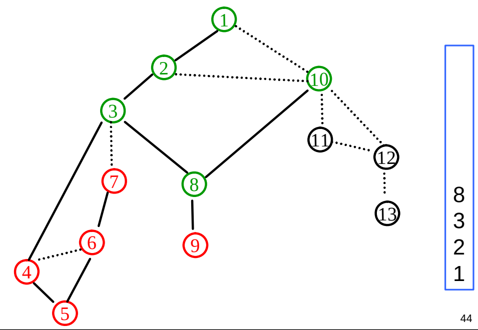


42

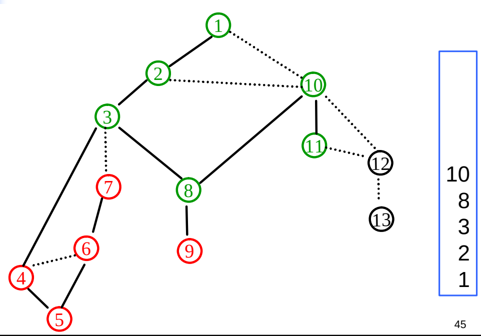
DFS(u)



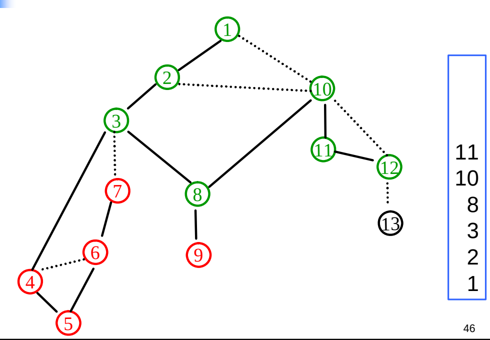
DFS(u)



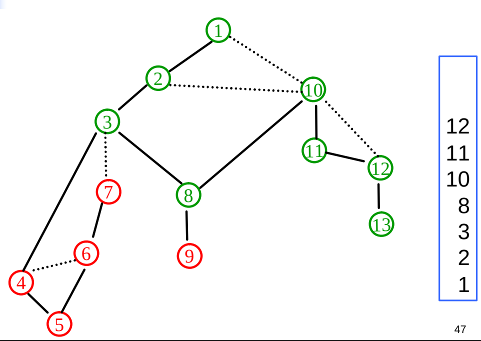
DFS(u)



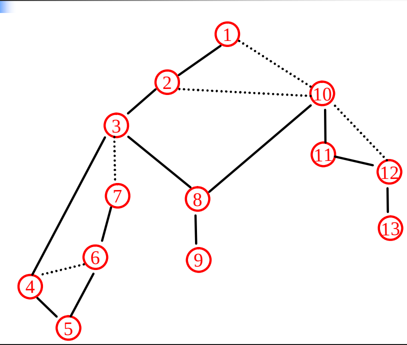
DFS(u)



DFS(u)



DFS(u)

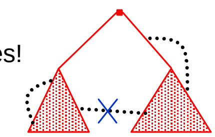


Properties of DFS(s)

- Like BFS(s):
  - DFS(s) visits **x** if and only if there is a path in **G** from **s** to **x**
  - Edges into undiscovered vertices define a tree
    - "depth first spanning tree" of **G**
- Unlike the BFS tree:
  - the DFS spanning tree isn't minimum depth
  - its levels don't reflect min distance from the root
  - non-tree edges never join vertices on the same or adjacent levels
- BUT…

Non-tree edges

- All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree
- No cross edges!



No cross edges in DFS on undirected graphs

- Claim: During **DFS(x)** every vertex marked visited is a descendant of **x** in the DFS tree **T**
- Claim: For every **x,y** in the DFS tree **T**, if **(x,y)** is an edge not in **T** then one of **x** or **y** is an ancestor of the other in **T**
- Proof:
  - One of **x** or **y** is visited first, suppose WLOG that **x** is visited first and therefore **DFS(x)** was called before **DFS(y)**
    - During **DFS(x)**, the edge **(x,y)** is examined
  - Since **(x,y)** is a not an edge of **T**, **y** was visited when the edge **(x,y)** was examined during **DFS(x)**
  - Therefore **y** was visited during the call to **DFS(x)** so **y** is a descendant of **x**.

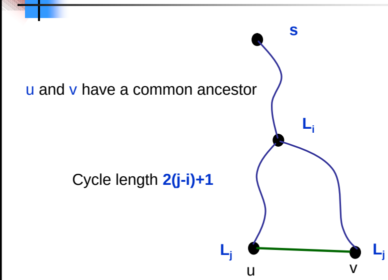
Applications of Graph Traversal: Bipartiteness Testing

- Easy: A graph G is not bipartite if it contains an odd length cycle
- WLOG: G is connected
  - Otherwise run on each component
- Simple idea: start coloring nodes starting at a given node s
  - Color s red
  - Color all neighbors of s blue
  - Color all their neighbors red
  - If you ever hit a node that was already colored
    - the same color as you want to color it, ignore it
    - the opposite color, output error

BFS gives Bipartiteness

- Run BFS assigning all vertices from layer L_i the color i mod 2
  - i.e. red if they are in an even layer, blue if in an odd layer
- If there is an edge joining two vertices from the same layer then output "Not Bipartite"

Why does it work?

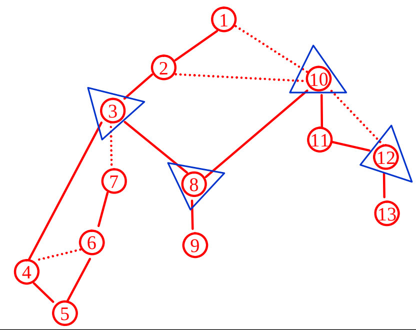


Application: Cut Points

- A node in an undirected graph is an **cut point** iff removing it disconnects the graph
- cut points represent vulnerabilities in a network – single points whose failure would split the network into 2 or more disconnected components

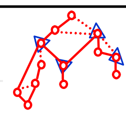
55

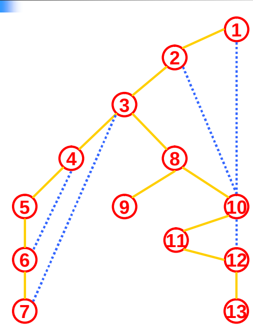
Cut Points



56

Cut Points





57

Cut Points from DFS

- Non-tree edges eliminate cut points
- Root node **r** is cut point ⇔ it has more than one child in the DFS tree **T**
  - If **r** has only one child in **T**, call it **u**
    - every node in **T** is reachable from **u** so removing **r** leaves **T** connected
  - Is **r** has more than one child, the fact that there are no cross edges means removing **r** disconnects the graph
- Leaf nodes are never cut points, more generally…
- Non-root node **u** is a cut point ⇔
  - There is some child **v** of **u** that does not have a non-tree edge leading from the subtree rooted at **v** to above **u** in the tree

58

Understanding cut points

- Notation:
  - For nodes **u** and **v** write **u** £ **v** if u is visited before **v** during a given DFS,
    - "**u** is earlier than **v** in the DFS"
  - For a node **u**, define **earliest(u)** to be the earliest node that is adjacent to some node in the subtree of the DFS tree rooted at **u**.
- Characterization:
  - Non-root node **u** is a cut point ⇔ there is some child **v** of **u** such that **u** £ **earliest(v)**

59

Proving characterization

- Suppose there is some child **v** of **u** such that **u** £ **earliest(v)**
  - Let **X** be set of nodes in subtree rooted at **v**
  - Only tree edge out of **X** goes to **u**
  - Any non-tree edges out of **X** must go up the tree but no earlier than **u** so can at best go to **u**
  - ∴ Removing **u** disconnects **X** from the rest of the graph
  - ∴ **u** is a cut point

60

Proving characterization

- Suppose every child **v** of **u** has **earliest(v)<u**
  - Let **G'** be **G-{u}**
  - **Claim:** **u** is not a cut point, i.e., **G'** is connected
  - We will find paths in **G'** from **r** to each node **w** of **G'**
  - If **w** ≠ **u** is not in subtree rooted at **u** then the original path is still there
  - If **w** ≠ **u** is in the subtree rooted at **u** then **w** lies in some subtree, call it T_v, below some child **v** of **u**
    - Since **earliest(v)<u** there is a path from **r** to **earliest(v)** and from **earliest(v)** to some node of T_v and therefore to **w**

Implementing Cut Points from DFS

- Number each node **v**, **dfsnum(v)** to get order
- For each vertex **v** compute
  - earliest(**v**)
    - the smallest number of a node pointed at by any descendant of **v** in the DFS tree (including **v** itself)
  - Can compute earliest(v) for every **v** during DFS at minimal extra cost
- Non-root node **u** is a cut point ⇔ for some child **v** of **u**
  - **dfsnum(u)≤earliest(v)**
  - Easy to compute and check during DFS

DFS(v)

Global Initialization:
  mark all vertices **u** "unvisited" via **dfsnum**[**u**] ← **-1**
  dfscounter ← **0**

```
DFS(v)
  dfscounter ← dfscounter+1
  dfsnum[v] ← dfscounter    // mark v "visited"
  for each edge (v,x)
    if (dfsnum[x] = -1)        // x previously unvisited
       add edge (v,x) to DFStree
       DFS(x)
  // mark v "fully-explored"
```


DFS(v) for Finding Cut Points

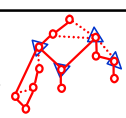
Global initialization: **dfsnum**[**u**] ← **-1** for all **u**; **dfscounter** ←**0**

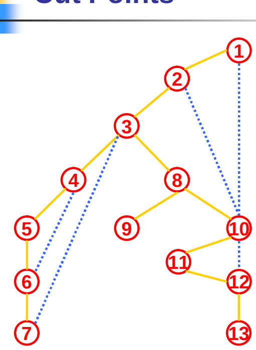
```
DFS(v)
 dfscounter ← dfscounter+1
 dfsnum[v] ← dfscounter
 earliest[v] ← dfsnum[v]              // initialization
 for each edge {v,x}
     if (dfsnum[x] = -1)         // x is unvisited
        DFS(x)
        if (earliest[x] ≥ dfsnum[v])
           print "v is a cut point, separating x"
        earliest[v] ← min(earliest[v], earliest[x])
     else if (x is not v's parent)
        earliest[v] ← min(earliest[v], dfsnum[x])
```

Check that {v,x} is a non-tree edge

Note: need a separate check for the root

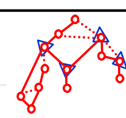
Cut Points

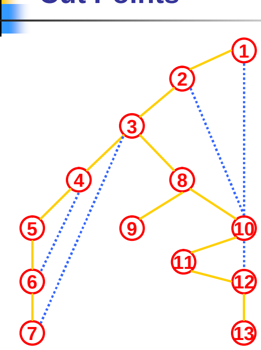




DFS #	Earliest
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	

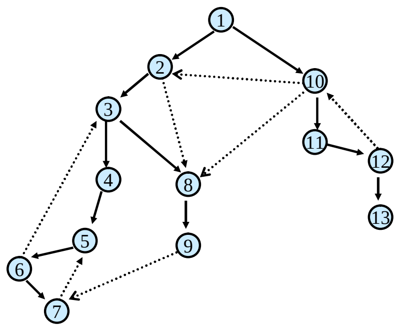
Cut Points





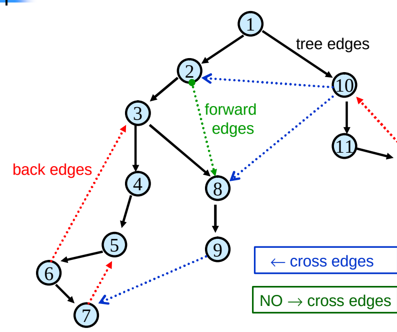
DFS #	Early	Cut
1	1	
2	1	
3	1	Y
4	3	
5	3	
6	3	
7	3	
8	1	Y
9	9	
10	1	Y
11	10	
12	10	Y
13	13	

DFS(v) for a directed graph



67

DFS(v)



68

Properties of Directed DFS

- Before DFS(s) returns, it visits all previously unvisited vertices reachable via directed paths from s
- Every cycle contains a back edge in the DFS tree

69

Directed Acyclic Graphs

- A directed graph G=(V,E) is acyclic if it has no directed cycles
- Terminology: A directed acyclic graph is also called a DAG

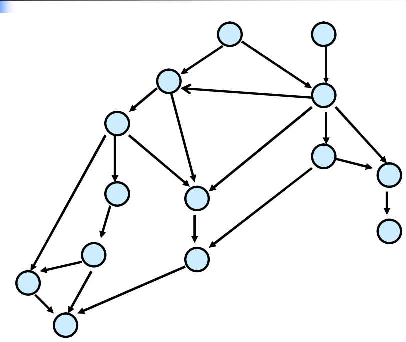
70

Topological Sort

- **Given:** a directed acyclic graph (DAG) **G=(V,E)**
- **Output:** numbering of the vertices of **G** with distinct numbers from **1** to **n** so edges only go from lower number to higher numbered vertices
- Applications
  - nodes represent tasks
  - edges represent precedence between tasks
  - topological sort gives a sequential schedule for solving them

71

Directed Acyclic Graph



72

In-degree 0 vertices

- Every DAG has a vertex of in-degree 0
- Proof: By contradiction
  - Suppose every vertex has some incoming edge
  - Consider following procedure:

    while (true) do

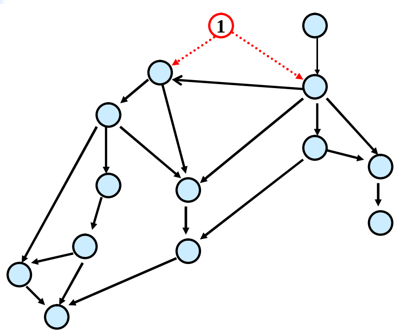
    v←some predecessor of **v**

  - After **n+1** steps where **n=|V|** there will be a repeated vertex
    - This yields a cycle, contradicting that it is a DAG

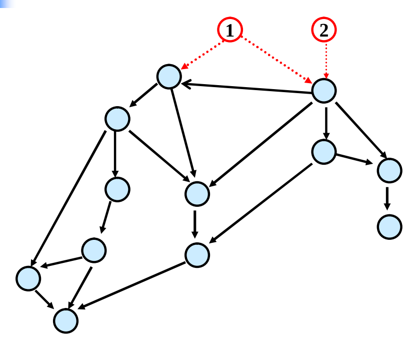
Topological Sort

- Can do using DFS
- Alternative simpler idea:
  - Any vertex of in-degree 0 can be given number 1 to start
  - Remove it from the graph and then give a vertex of in-degree 0 number 2, etc.

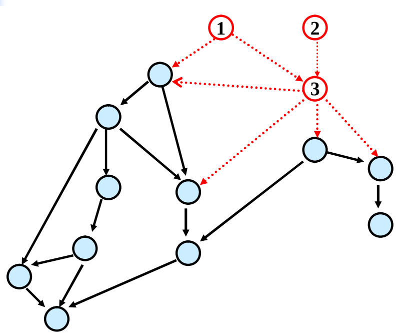
Topological Sort



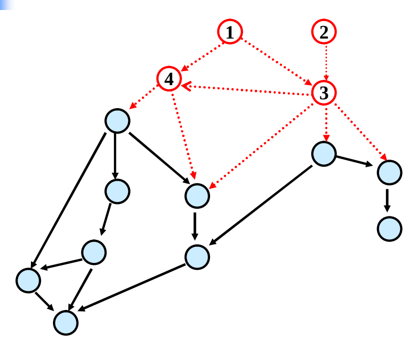
Topological Sort



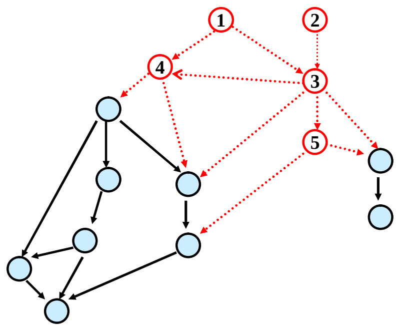
Topological Sort



Topological Sort

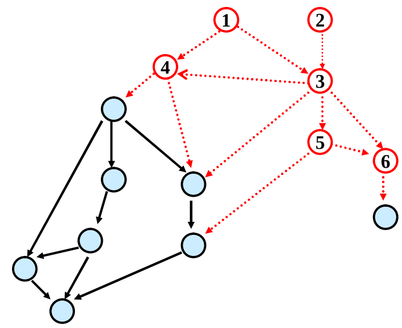


Topological Sort



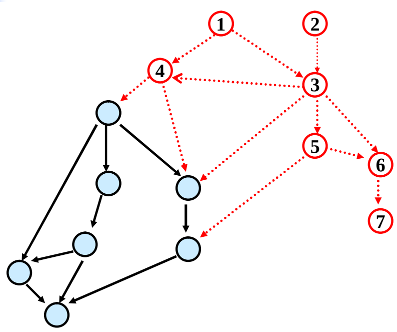
79

Topological Sort



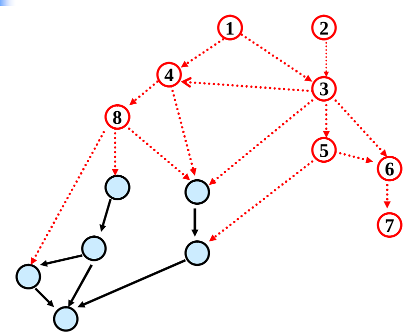
80

Topological Sort



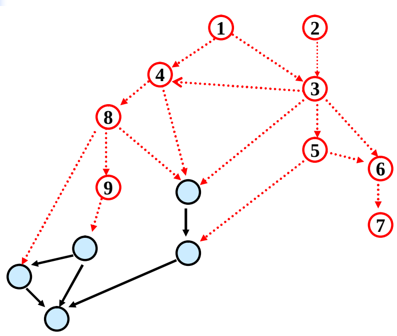
81

Topological Sort



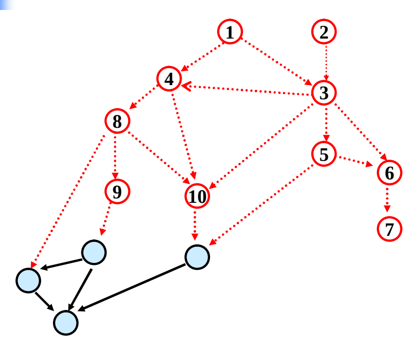
82

Topological Sort



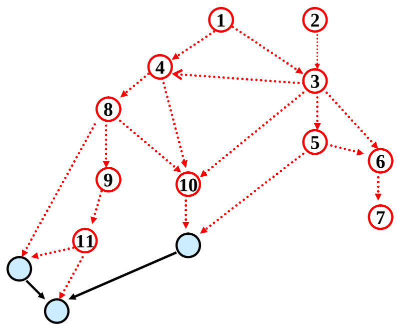
83

Topological Sort

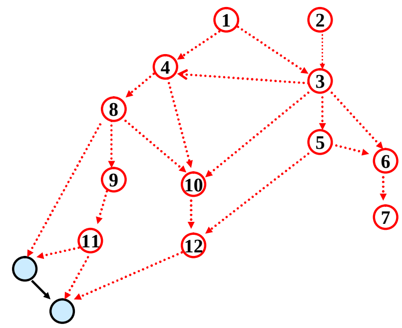


84

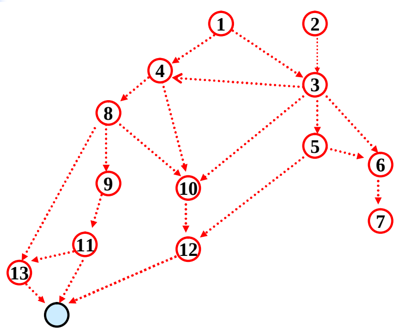
Topological Sort



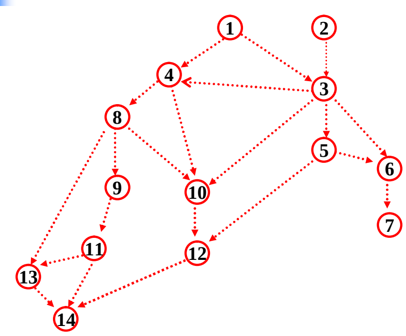
Topological Sort



Topological Sort



Topological Sort



Implementing Topological Sort

- Go through all edges, computing in-degree for each vertex **O(m+n)**
- Maintain a queue (or stack) of vertices of in-degree **0**
- Remove any vertex in queue and number it
- When a vertex is removed, decrease in-degree of each of its neighbors by **1** and add them to the queue if their degree drops to **0**
- Total cost **O(m+n)**