

# 9

# Graph Algorithms

9 December 2024

Prof. Dr. Sebastian Wild

# Learning Outcomes

## Unit 9: *Graph Algorithms*

1. Know basic terminology from graph theory, including types of graphs.
2. Know adjacency matrix and adjacency list representations and their performance characteristics.
3. Know graph-traversal based algorithm, including efficient implementations.
4. Be able to proof correctness of graph-traversal-based algorithms.
5. Know algorithms for maximum flows in networks.
6. Be able to model new algorithmic problems as graph problems.

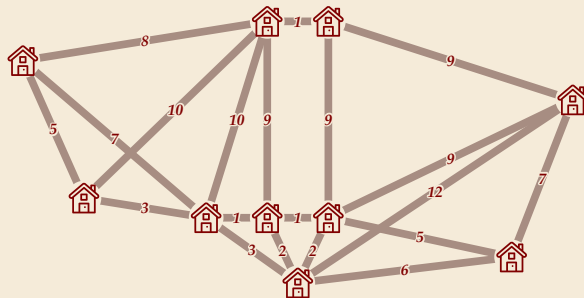
## 9 Graph Algorithms

- 9.1 Introduction & Definitions
- 9.2 Graph Representations
- 9.3 Graph Traversal
- 9.4 BFS and DFS
- 9.5 Advanced Uses of DFS
- 9.6 Network flows
- 9.7 The Ford-Fulkerson Method

## **9.1 Introduction & Definitions**

# Graphs in real life

- ▶ a graph is an abstraction of *entities* with their (pairwise) *relationships*
- ▶ abundant examples in real life (often called network there)
  - ▶ social networks: e. g. persons and their friendships, ... *Five/Six? degrees of separation*
  - ▶ physical networks: cities and highways, roads networks, power grids etc., the Internet, ...
  - ▶ content networks: world wide web, ontologies, ...
  - ▶ ...



Many More examples, e. g., in Sedgewick & Wayne's videos:

<https://www.coursera.org/learn/algorithms-part2>

# Flavors of Graphs

- ▶ Since graphs are used to model so many different entities and relations, they come in several variants

Property	Yes	No
edges are one-way	<i>directed</i> graph ( <i>digraph</i> )	<i>undirected</i> graph
$\leq 1$ edge between $u$ and $v$	<u><i>simple</i></u> graph	<i>multigraph</i> / with <i>parallel</i> edges
edges can lead from $v$ to $v$	with <i>loops</i> ( <i>Self-Loops, Self-Loops</i> )	<u>(loop-free)</u>
edges have weights	<i>(edge-) weighted</i> graph	<u><i>unweighted</i></u> graph

☺ any combination of the above can make sense ...

- ▶ Synonyms:
  - ▶ **vertex** („Knoten“) = node = point = „Ecke“
  - ▶ **edge** („Kante“) = arc = line = relation = arrow = „Pfeil“
  - ▶ **graph** = network

# Graph Theory

► default: unweighted, undirected, loop-free & simple graphs

► *Graph*  $G = (V, E)$  with

►  $V$  a finite of *vertices*

$$e = \{u, v\}$$

►  $E \subseteq [V]^2$  a set of *edges*, which are 2-subsets of  $V$ :  $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

# Graph Theory

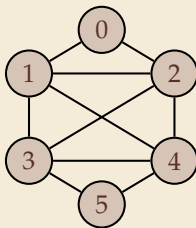
- ▶ default: unweighted, undirected, loop-free & simple graphs
- ▶ *Graph*  $G = (V, E)$  with
  - ▶  $V$  a finite set of *vertices*
  - ▶  $E \subseteq [V]^2$  a set of *edges*, which are 2-subsets of  $V$ :  $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

## Example

$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \{\{0, 1\}, \{1, 2\}, \{1, 4\}, \{1, 3\}, \{0, 2\}, \\ \{2, 4\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}.$$

## Graphical representation



like so ...



# Graph Theory

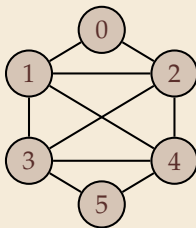
- ▶ default: unweighted, undirected, loop-free & simple graphs
- ▶ *Graph*  $G = (V, E)$  with
  - ▶  $V$  a finite of *vertices*
  - ▶  $E \subseteq [V]^2$  a set of *edges*, which are 2-subsets of  $V$ :  $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

## Example

$$V = \{0, 1, 2, 3, 4, 5\}$$

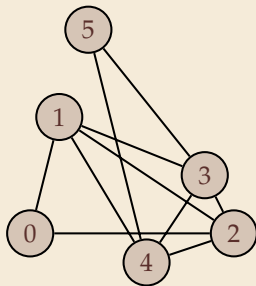
$$E = \{\{0, 1\}, \{1, 2\}, \{1, 4\}, \{1, 3\}, \{0, 2\}, \\ \{2, 4\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}.$$

## Graphical representation



like so ...

=

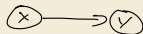


... or so

(same graph)

# Digraphs

- ▶ default digraph: unweighted, loop-free & simple
- ▶ *Digraph (directed graph)*  $G = (V, E)$  with
  - ▶  $V$  a finite of *vertices*
  - ▶  $E \subseteq V^2 \setminus \{(v, v) : v \in V\}$  a set of (*directed*) *edges*,  
 $V^2 = V \times V = \{(x, y) : x \in V \wedge y \in V\}$  2-tuples / ordered pairs over  $V$



# Digraphs

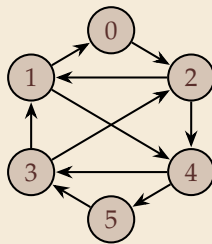
- ▶ default digraph: unweighted, loop-free & simple
- ▶ *Digraph (directed graph)*  $G = (V, E)$  with
  - ▶  $V$  a finite of *vertices*
  - ▶  $E \subseteq V^2 \setminus \{(v, v) : v \in V\}$  a set of (*directed*) *edges*,  
 $V^2 = V \times V = \{(x, y) : x \in V \wedge y \in V\}$  2-tuples / ordered pairs over  $V$

## Example

$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \{(0, 2), (1, 0), (1, 4), (2, 1), (2, 4), \\ (3, 1), (3, 2), (4, 3), (4, 5), (5, 3)\}$$

## Graphical representation



# Graph Terminology

## Undirected Graphs

- ▶  $V(G)$  set of vertices,  $E(G)$  set of edges
- ▶ write  $\underline{uv}$  (or  $vu$ ) for edge  $\{u, v\}$
- ▶ edges *incident* at vertex  $v$ :  $E(v)$
- ▶  $u$  and  $v$  are *adjacent* iff  $\{u, v\} \in E$ ,
- ▶ *neighborhood*  $N(v) = \{w \in V : w \text{ adjacent to } v\}$
- ▶ *degree*  $d(v) = |E(v)|$

## Directed Graphs (where different)

- ▶  $uv$  for  $(u, v)$
- ▶ iff  $(u, v) \in E \vee (v, u) \in E$
- ▶ in-/out-neighbors  $N_{\text{in}}(v), N_{\text{out}}(v)$
- ▶ in-/out-degree  $d_{\text{in}}(v), d_{\text{out}}(v)$

# Graph Terminology

## Undirected Graphs

- ▶  $V(G)$  set of vertices,  $E(G)$  set of edges
- ▶ write  $uv$  (or  $vu$ ) for edge  $\{u, v\}$
- ▶ edges *incident* at vertex  $v$ :  $E(v)$
- ▶  $u$  and  $v$  are *adjacent* iff  $\{u, v\} \in E$ ,
- ▶ *neighborhood*  $N(v) = \{w \in V : w \text{ adjacent to } v\}$
- ▶ *degree*  $d(v) = |E(v)|$
- ▶ <sup>Kantenweg</sup> *walk*  $w$  of length  $n$ : sequence of vertices  $w[0..n]$  with  $\forall i \in [0..n) : w[i]w[i+1] \in E$
- ▶ <sup>↕</sup> *path*  $p$  is a (vertex-) simple walk: without duplicate vertices except possibly its endpoints
- ▶ *edge-simple* walk: no edge used twice
- ▶ *cycle*  $c$  is a closed path, i. e.,  $c[0] = c[n]$

## Directed Graphs (where different)

- ▶  $uv$  for  $(u, v)$
- ▶ iff  $(u, v) \in E \vee (v, u) \in E$
- ▶ in-/out-neighbors  $N_{\text{in}}(v), N_{\text{out}}(v)$
- ▶ in-/out-degree  $d_{\text{in}}(v), d_{\text{out}}(v)$

(ges. losse Weg, Zykkel, Kreis, Zykkel)

# Graph Terminology

## Undirected Graphs

- ▶  $V(G)$  set of vertices,  $E(G)$  set of edges
- ▶ write  $uv$  (or  $vu$ ) for edge  $\{u, v\}$
- ▶ edges *incident* at vertex  $v$ :  $E(v)$
- ▶  $u$  and  $v$  are *adjacent* iff  $\{u, v\} \in E$ ,
- ▶ *neighborhood*  $N(v) = \{w \in V : w \text{ adjacent to } v\}$
- ▶ *degree*  $d(v) = |E(v)|$
- ▶ *walk*  $w$  of length  $n$ : sequence of vertices  $w[0..n]$  with  $\forall i \in [0..n) : \underline{w[i]w[i+1]} \in E$
- ▶ *path*  $p$  is a (vertex-) simple walk: without duplicate vertices except possibly its endpoints
- ▶ *edge-simple* walk: no edge used twice
- ▶ *cycle*  $c$  is a closed path, i. e.,  $c[0] = c[n]$
- ▶  $G$  is *connected*  
iff for all  $u \neq v \in V$  there is a path from  $u$  to  $v$
- ▶  $G$  is *acyclic* iff  $\nexists$  cycle (of length  $n \geq 1$ ) in  $G$

## Directed Graphs (where different)

- ▶  $uv$  for  $(u, v)$
- ▶ iff  $(u, v) \in E \vee (v, u) \in E$
- ▶ in-/out-neighbors  $N_{\text{in}}(v), N_{\text{out}}(v)$
- ▶ in-/out-degree  $d_{\text{in}}(v), d_{\text{out}}(v)$
- ▶ *strongly connected* for digraphs  
(*weakly connected* = connected ignoring directions)

# Typical graph-processing problems

- ▶ **Path:** Is there a path between  $s$  and  $t$ ?  
**Shortest path:** What is the shortest path (distance) between  $s$  and  $t$ ?
- ▶ **Cycle:** Is there a cycle in the graph?  
**Euler tour:** Is there a cycle that uses each edge exactly once?  
**Hamilton(ian) cycle:** Is there a cycle that uses each vertex exactly once. |
- ▶ **Connectivity:** Is there a way to connect all of the vertices?  
**MST:** What is the best way to connect all of the vertices?  
**Biconnectivity:** Is there a vertex whose removal disconnects the graph?
- ▶ **Planarity:** Can you draw the graph in the plane with no crossing edges?
- ▶ **Graph isomorphism:** Are two graphs the same up to renaming vertices? |

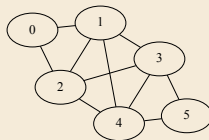
↖ can vary a lot, despite superficial similarity of problems

**Challenge:** Which of these problems  
can be computed in (near) linear time?  
in reasonable polynomial time?  
are intractable?

# Tools to work with graphs

- ▶ Convenient GUI to edit & draw graphs: *yEd live*  
[yworks.com/yed-live](http://yworks.com/yed-live)
- ▶ *graphviz* cmdline utility to draw graphs
  - ▶ Simple text format for graphs: DOT

```
graph G {  
    0 -- 2;    2 -- 4;  
    1 -- 0;    2 -- 3;  
    1 -- 4;    3 -- 4;  
    1 -- 3;    3 -- 5;  
    2 -- 1;    4 -- 5;  
}
```



`dot -Tpdf graph.dot -Kfdp > graph.pdf`

- ▶ graphs are typically not built into programming languages, but libraries exist
  - ▶ e. g. part of *Google Guava* for Java
  - ▶ they usually allow arbitrary objects as vertices
  - ▶ aimed at ease of use



## 9.2 Graph Representations

# Graphs in Computer Memory

- ▶ We defined graphs in set-theoretic terms. . .  
but computers can't directly deal with sets efficiently
- ↪ need to choose a *representation* for graphs.
  - ▶ which is better depends on the required operations

# Graphs in Computer Memory

- ▶ We defined graphs in set-theoretic terms. . .  
but computers can't directly deal with sets efficiently

↪ need to choose a *representation* for graphs.

- ▶ which is better depends on the required operations

## Key Operations:

- ▶  $\text{isAdjacent}(u, v)$   
Test whether  $uv \in E$
- ▶  $\text{adj}(v)$   
Adjacency list of  $v$  (iterate through (out-)neighbors of  $v$ )
- ▶ most others can be computed based on these

# Graphs in Computer Memory

- ▶ We defined graphs in set-theoretic terms. . .  
but computers can't directly deal with sets efficiently

↪ need to choose a *representation* for graphs.

- ▶ which is better depends on the required operations

## Key Operations:

- ▶  $\text{isAdjacent}(u, v)$   
Test whether  $uv \in E$
- ▶  $\text{adj}(v)$   
Adjacency list of  $v$  (iterate through (out-)neighbors of  $v$ )
- ▶ most others can be computed based on these

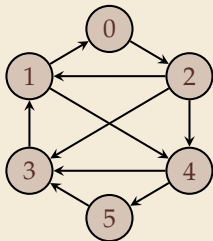
## Conventions:

- ▶ (di)graph  $G = (V, E)$  (omitted if clear from context)
- ▶  $n = |V|$ ,  $m = |E|$
- ▶ in implementations assume  $V = [0..n)$  (if needed, use symbol table to map complex objects to  $V$ )

# Adjacency Matrix Representation

- ▶ adjacency matrix  $A \in \{0, 1\}^{n \times n}$  of  $G$ : matrix with  $A[u, v] = [uv \in E] = \begin{cases} 1 & uv \in E \\ 0 & \text{sonst} \end{cases}$ 
  - ▶ works for both directed and undirected graphs (undirected  $\rightsquigarrow A = A^T$  symmetric)
  - ▶ can use a weight  $w(uv)$  or multiplicity in  $A[u, v]$  instead of 0/1
  - ▶ can represent loops via  $A[v, v]$

Example:

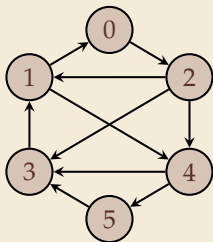


$$A = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{matrix}$$

# Adjacency Matrix Representation

- ▶ adjacency matrix  $A \in \{0, 1\}^{n \times n}$  of  $G$ : matrix with  $A[u, v] = [uv \in E]$ 
  - ▶ works for both directed and undirected graphs (undirected  $\rightsquigarrow A = A^T$  symmetric)
  - ▶ can use a weight  $w(uv)$  or multiplicity in  $A[u, v]$  instead of 0/1
  - ▶ can represent loops via  $A[v, v]$

Example:



$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



isAdjacent in  $O(1)$  time



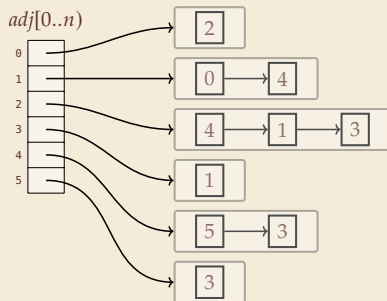
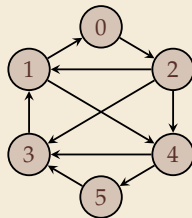
$O(n^2)$  (bits of) space wasteful for sparse graphs



adj( $v$ ) iteration takes  $O(n)$  (independent of  $d(v)$ )

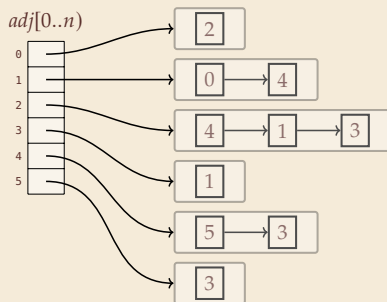
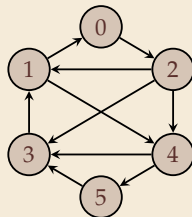
# Adjacency List Representation

- Store a linked list of neighbors for each vertex  $v$ :
  - $\underline{adj[0..n]}$  bag of neighbors (as linked list)
  - undirected edge  $\{u, v\} \rightsquigarrow v$  in  $adj[u]$  and  $u$  in  $adj[v]$
  - weighted edge  $\underline{uv} \rightsquigarrow$  store pair  $(v, w(uv))$  in  $adj[u]$
  - multiple edges and loops can be represented



# Adjacency List Representation

- ▶ Store a linked list of neighbors for each vertex  $v$ :
  - ▶  $adj[0..n)$  bag of neighbors (as linked list)
  - ▶ undirected edge  $\{u, v\} \rightsquigarrow v$  in  $adj[u]$  and  $u$  in  $adj[v]$
  - ▶ weighted edge  $uv \rightsquigarrow$  store pair  $(v, w(uv))$  in  $adj[u]$
  - ▶ multiple edges and loops can be represented



👎  $isAdjacent(u, v)$  takes  $\Theta(d(u))$  time (worst case)

👍  $adj(v)$  iteration  $O(1)$  per neighbor

👍  $\Theta(n + m)$  (words of) space for any graph ( $\ll \Theta(n^2)$  bits for moderate  $m$ )

$\rightsquigarrow$  de-facto standard for graph algorithms



# Graph Types and Representations

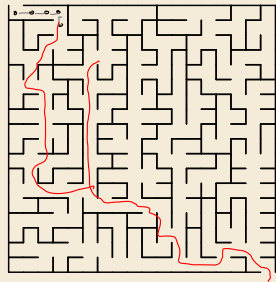
- ▶ Note that adj matrix and lists for undirected graphs effectively are representation of directed graph with directed edges both ways
  - ▶ conceptually still important to distinguish!
- ▶ multigraphs, loops, edge weights all naturally supported in adj lists
  - ▶ good if we allow and use them
  - ▶ but requires explicit checks to enforce simple / loopfree / bidirectional!
- ▶ we focus on **static graphs**  
dynamically changing graphs much harder to handle

## 9.3 Graph Traversal

# Generic Graph Traversal

- ▶ Plethora of graph algorithms can be expressed as a systematic exploration of a graph
  - ▶ depth-first search, breadth-first search
  - ▶ connected components
  - ▶ detecting cycles
  - ▶ topological sorting
  - ▶ Hierholzer's algorithm for Euler walks
  - ▶ strong components
  - ▶ testing bipartiteness
  - ▶ Dijkstra's algorithm
  - ▶ Prim's algorithm
  - ▶ Lex-BFS for perfect elimination orders of chordal graphs
  - ▶ ...

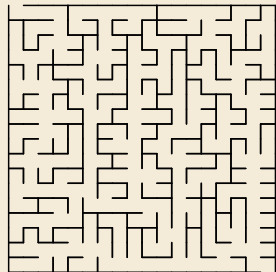
visiting all nodes & edges



# Generic Graph Traversal

- ▶ Plethora of graph algorithms can be expressed as a systematic exploration of a graph
  - ▶ depth-first search, breadth-first search
  - ▶ connected components
  - ▶ detecting cycles
  - ▶ topological sorting
  - ▶ Hierholzer's algorithm for Euler walks
  - ▶ strong components
  - ▶ testing bipartiteness
  - ▶ Dijkstra's algorithm
  - ▶ Prim's algorithm
  - ▶ Lex-BFS for perfect elimination orders of chordal graphs
  - ▶ ...

visiting all nodes & edges



↪ Formulate generic traversal algorithm

- ▶ first in abstract terms to argue about correctness
- ▶ then again for concrete instance with efficient data structures

# Tricolor Graph Traversal

## Tricolor Graph Search:

- ▶ maintain vertices in 3 (dynamic) sets

- ▶ **Gray: unseen vertices**

The traversal has not reached these vertices so far.

- ▶ **Green: done vertices** (a.k.a. visited vertices)

These vertices have been visited and all their edges have been explored already.

- ▶ **Red: active vertices** (a.k.a. frontier („Rand“) of traversal)

All others, i. e., vertices that have been reached and some unexplored edges remain; initially some selected start vertices  $S$ .

### Invariant:

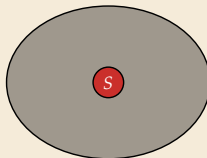
No edges from **done** to **unseen** vertices

- ▶ (implicitly) maintain status of each edge

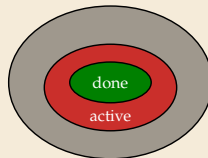
- ▶ **not yet used**

- ▶ **used edge**

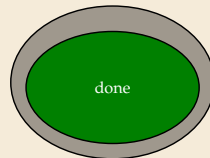
want to color green  
→ to do that need neighbors here red



initial state



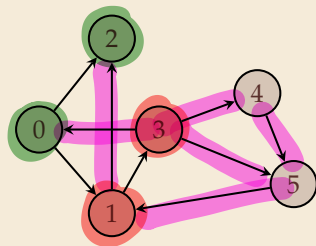
during traversal



final state

# Generic Tricolor Graph Traversal – Code

```
1 procedure genericGraphTraversal( $G, S$ )
2   // (di)graph  $G = (V, E)$  and start vertices  $S \subseteq V$ 
3    $C[0..n] := \text{unseen}$  // Color array, all cells initialized to unseen
4   for  $s \in S$  do  $C[s] := \text{active}$  end for
5    $\text{unusedEdges} := E$ 
6   while  $\exists v : C[v] == \text{active}$ 
7      $v := \text{nextActiveVertex}()$  // Freedom 1: Which frontier vertex?
8     if  $\nexists vw \in \text{unusedEdges}$  // no more edges from  $v \rightsquigarrow$  done with  $v$ 
9        $C[v] := \text{done}$ 
10    else
11       $w := \text{nextUnusedEdge}(v)$  // Freedom 2: Which of its edges?
12      if  $C[w] == \text{unseen}$ 
13         $C[w] := \text{active}$ 
14      end if
15       $\text{unusedEdges.remove}(vw)$ 
16    end if
17  end while
```

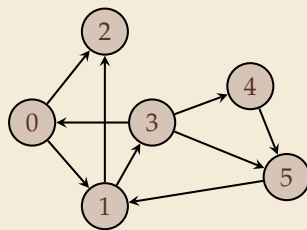


**Invariant:**

No edges from *done* to *unseen* vertices

# Generic Tricolor Graph Traversal – Code

```
1 procedure genericGraphTraversal( $G, S$ )
2   // (di)graph  $G = (V, E)$  and start vertices  $S \subseteq V$ 
3    $C[0..n] := \text{unseen}$  // Color array, all cells initialized to unseen
4   for  $s \in S$  do  $C[s] := \text{active}$  end for
5    $\text{unusedEdges} := E$ 
6   while  $\exists v : C[v] == \text{active}$ 
7      $v := \text{nextActiveVertex}()$  // Freedom 1: Which frontier vertex?
8     if  $\nexists vw \in \text{unusedEdges}$  // no more edges from  $v \rightsquigarrow$  done with  $v$ 
9        $C[v] := \text{done}$ 
10    else
11       $w := \text{nextUnusedEdge}(v)$  // Freedom 2: Which of its edges?
12      if  $C[w] == \text{unseen}$ 
13         $C[w] := \text{active}$ 
14      end if
15       $\text{unusedEdges.remove}(vw)$ 
16    end if
17  end while
```



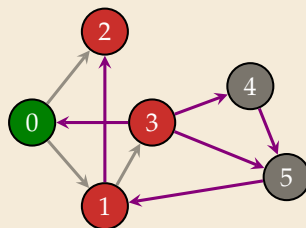
**Invariant:**

No edges from *done* to *unseen* vertices

- Implementations of `nextActiveVertex()` and `nextUnusedEdge( $v$ )` depends on (and defines!) specific traversal-based graph algorithms

# Generic Tricolor Graph Traversal – Code

```
1 procedure genericGraphTraversal( $G, S$ )
2   // (di)graph  $G = (V, E)$  and start vertices  $S \subseteq V$ 
3    $C[0..n] := \text{unseen}$  // Color array, all cells initialized to unseen
4   for  $s \in S$  do  $C[s] := \text{active}$  end for
5    $\text{unusedEdges} := E$ 
6   while  $\exists v : C[v] == \text{active}$ 
7      $v := \text{nextActiveVertex}()$  // Freedom 1: Which frontier vertex?
8     if  $\nexists vw \in \text{unusedEdges}$  // no more edges from  $v \rightsquigarrow$  done with  $v$ 
9        $C[v] := \text{done}$ 
10    else
11       $w := \text{nextUnusedEdge}(v)$  // Freedom 2: Which of its edges?
12      if  $C[w] == \text{unseen}$ 
13         $C[w] := \text{active}$ 
14      end if
15       $\text{unusedEdges.remove}(vw)$ 
16    end if
17  end while
```



**Invariant:**

No edges from *done* to *unseen* vertices

- Implementations of `nextActiveVertex()` and `nextUnusedEdge( $v$ )` depends on (and defines!) specific traversal-based graph algorithms



# Generic Reachability

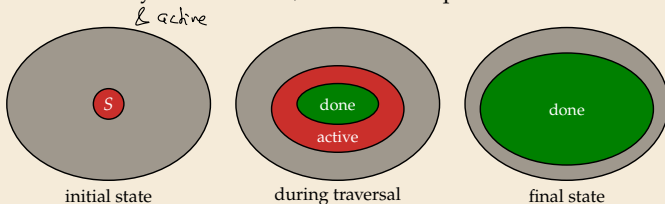
- Any choices `nextActiveVertex()` and `nextUnusedEdge( $v$ )` suffice to find exactly the vertices reachable from  $S$  in *done*

# Generic Reachability

- Any choices `nextActiveVertex()` and `nextUnusedEdge(v)` suffice to find exactly the vertices reachable from  $S$  in *done*

► **Invariant:**

- No edges from *done* to *unseen* vertices
- For every *done* vertex  $v$ , there exists a path from  $s \in S$  to  $v$ .



IB : (1) ✓ (2) ✓

IS : (i) line 9 color  $v$  green  
no edges to unseen from  $v$

(1) ✓

(2) by IH ✓

(ii) line 11 color  $w$  red  
(1) ✓

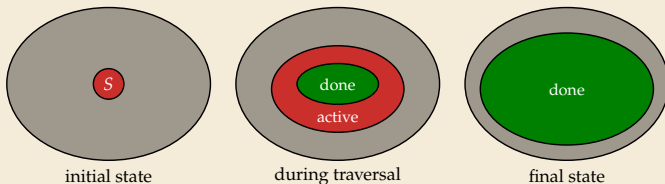
(2) by IH  $s \rightsquigarrow v \rightarrow w$   
hence  $s \rightsquigarrow w$  ✓

# Generic Reachability

- Any choices `nextActiveVertex()` and `nextUnusedEdge(v)` suffice to find exactly the vertices reachable from  $S$  in *done*

- Invariant:**

- No edges from *done* to *unseen* vertices
- For every *done* vertex  $v$ , there exists a path from  $s \in S$  to  $v$ .



↪ in final state:

- $v \in \text{done} \rightsquigarrow$  path from  $S \rightsquigarrow$  reachable from  $S$
- $v \in \text{unseen} \rightsquigarrow$  not reachable from  $\text{done} \supseteq S \rightsquigarrow$  not reachable from  $S$

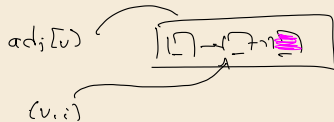
Assume not, then  $S \rightsquigarrow v$  first vertex on path that is unseen has done neighbor  $\nexists$  (1)

# Data Structures for Frontier

- ▶ We need efficient support for
  - ▶ test  $\exists v : C[v] = \text{active}$ , `nextActiveVertex()`
  - ▶ test  $\exists vw \in \text{unusedEdges}$ , `nextUnusedEdge(v)`
  - ▶ `unusedEdges.remove(vw)`

# Data Structures for Frontier

- ▶ We need efficient support for
  - ▶ test  $\exists v : C[v] = \text{active}$ , `nextActiveVertex()`
  - ▶ test  $\exists vw \in \text{unusedEdges}$ , `nextUnusedEdge(v)`
  - ▶ `unusedEdges.remove(vw)`
- ▶ Typical solution maintains **bag** “*frontier*” of *pairs*  $(v, i)$  where  $v \in V$  and  $i$  is an **iterator** in `adj[v]`
  - ▶ `unusedEdges` represented implicitly: edge used iff previously returned by  $i$ 
    - $\rightsquigarrow$  don't need `unusedEdges.remove(vw)`



# Data Structures for Frontier

- ▶ We need efficient support for
  - ▶ test  $\exists v : C[v] = \text{active}$ , `nextActiveVertex()`
  - ▶ test  $\exists vw \in \text{unusedEdges}$ , `nextUnusedEdge(v)`
  - ▶ `unusedEdges.remove(vw)`
- ▶ Typical solution maintains **bag** “*frontier*” of *pairs*  $(v, i)$  where  $v \in V$  and  $i$  is an **iterator** in `adj[v]`
  - ▶ `unusedEdges` represented implicitly: edge used iff previously returned by  $i$ 
    - $\rightsquigarrow$  don't need `unusedEdges.remove(vw)`
  - ▶ Implement  $\exists v : C[v] = \text{active}$  via `frontier.isEmpty()`
  - ▶ Implement  $\exists vw \in \text{unusedEdges}$  via `i.hasNext()` assuming  $(v, i) \in \text{frontier}$
  - ▶ Implement `nextUnusedEdge(v)` via `i.next()` assuming  $(v, i) \in \text{frontier}$
- $\rightsquigarrow$  all operations apart from `nextActiveVertex()` in  $O(1)$  time
- $\rightsquigarrow$  *frontier* requires  $O(n)$  extra space

## 9.4 BFS and DFS

# Breadth-First Search

- Maintain *frontier* in a **queue** (FIFO: first in, first out)

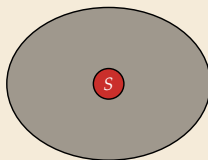


# Breadth-First Search

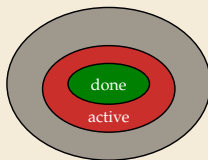
► Maintain *frontier* in a **queue** (FIFO: first in, first out)

► **Invariant:**

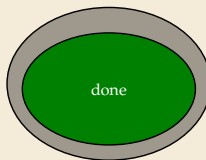
1. No edges from done to unseen vertices
2. All *done* <sup>& active</sup> vertices are reached via a **shortest path** from  $S$  ← fewest edges
3. *frontier* stores active vertices **sorted** by distance from  $S$



initial state



during traversal



final state

$$\text{distance}_G(v, S)$$

$$= \min_{s \in S} \text{distance}_G(v, s)$$

||  
#edges  
length of shortest  
path from  $s$  to  $v$

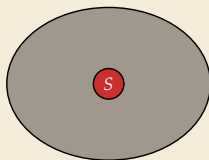
⇒ in final state, we reach all reachable vertices via shortest paths

# Breadth-First Search

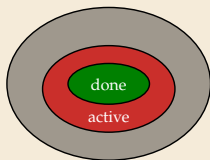
- Maintain *frontier* in a **queue** (FIFO: first in, first out)

## ► Invariant:

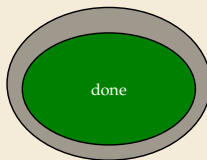
1. No edges from done to unseen vertices
2. All *active* vertices are reached via a **shortest path** from  $S$
3. *frontier* stores active vertices **sorted** by distance from  $S$



initial state



during traversal



final state

IS: (1) ✓ (2) ✓ (3) ✓ (4) ✓

IS: (i) no more edges  
v → done (1) ✓

(2) I.H ✓

(3) ✓ (4) If new distance of head  $k+1$  vertices at dist.  $k+1$  in queue.

(ii) visited edge  $vw$

(a) w unseen

to be continued

(5)

⇒ in final state, we reach all reachable vertices via shortest paths

- To preserve that knowledge, we collect extra information during traversal

- $parent[v]$  stores predecessor on path from  $S$  via which  $v$  was reached
- $distFromS[v]$  stores the length of this path

time when  $v$  was  
colored red

# Breadth-First Search – Code

---

```
1 procedure bfs( $G, S$ )
2   // (di)graph  $G = (V, E)$  and start vertices  $S \subseteq V$ 
3    $C[0..n] := \text{unseen}$  // New array initialized to all unseen
4    $\text{frontier} := \text{new Queue}$ ;
5    $\text{parent}[0..n] := \text{NOT\_VISITED}$ ;  $\text{distFromS}[0..n] := \infty$ 
6   for  $s \in S$ 
7      $\text{parent}[s] := \text{NONE}$ ;  $\text{distFromS}[s] := 0$ 
8      $C[s] := \text{active}$ ;  $\text{frontier.enqueue}((s, G.\text{adj}[s].\text{iterator}()))$ 
9   end for
10  while  $\neg \text{frontier.isEmpty}()$ 
11     $(v, i) := \text{frontier.peek}()$ 
12    if  $\neg i.\text{hasNext}()$  //  $v$  has no unused edge
13       $C[v] := \text{done}$ ;  $\text{frontier.dequeue}()$ 
14    else
15       $w := i.\text{next}()$  // Advance  $i$  in  $\text{adj}[v]$ 
16      if  $C[w] == \text{unseen}$ 
17         $\text{parent}[w] := v$ ;  $\text{distFromS}[w] := \text{distFromS}[v] + 1$ 
18         $C[w] := \text{active}$ ;  $\text{frontier.enqueue}((w, G.\text{adj}[w].\text{iterator}()))$ 
19      end if
20    end if
21  end while
```

---

# Breadth-First Search – Code

---

```
1 procedure bfs( $G, S$ )
2   // (di)graph  $G = (V, E)$  and start vertices  $S \subseteq V$ 
3    $C[0..n] := \text{unseen}$  // New array initialized to all unseen
4    $\text{frontier} := \text{new Queue}$ ;
5    $\text{parent}[0..n] := \text{NOT\_VISITED}$ ;  $\text{distFromS}[0..n] := \infty$ 
6   for  $s \in S$ 
7      $\text{parent}[s] := \text{NONE}$ ;  $\text{distFromS}[s] := 0$ 
8      $C[s] := \text{active}$ ;  $\text{frontier.enqueue}((s, G.\text{adj}[s].\text{iterator}()))$ 
9   end for
10  while  $\neg \text{frontier.isEmpty}()$ 
11     $(v, i) := \text{frontier.peek}()$ 
12    if  $\neg i.\text{hasNext}()$  //  $v$  has no unused edge
13       $C[v] := \text{done}$ ;  $\text{frontier.dequeue}()$ 
14    else
15       $w := i.\text{next}()$  // Advance  $i$  in  $\text{adj}[v]$ 
16      if  $C[w] == \text{unseen}$ 
17         $\text{parent}[w] := v$ ;  $\text{distFromS}[w] := \text{distFromS}[v] + 1$ 
18         $C[w] := \text{active}$ ;  $\text{frontier.enqueue}((w, G.\text{adj}[w].\text{iterator}()))$ 
19      end if
20    end if
21  end while
```

---

- ▶  $\text{parent}$  stores a shortest-path tree/forest
- ▶ can retrieve shortest path to  $v$  from some vertex  $s \in S$  (backwards) by following  $\text{parent}[v]$  iteratively

# Breadth-First Search – Code

---

```
1 procedure bfs( $G, S$ )
2   // (di)graph  $G = (V, E)$  and start vertices  $S \subseteq V$ 
3    $C[0..n] := \text{unseen}$  // New array initialized to all unseen
4    $\text{frontier} := \text{new Queue}$ ;
5    $\text{parent}[0..n] := \text{NOT\_VISITED}$ ;  $\text{distFromS}[0..n] := \infty$ 
6   for  $s \in S$ 
7      $\text{parent}[s] := \text{NONE}$ ;  $\text{distFromS}[s] := 0$ 
8      $C[s] := \text{active}$ ;  $\text{frontier.enqueue}((s, G.\text{adj}[s].\text{iterator}()))$ 
9   end for
10  while  $\neg \text{frontier.isEmpty}()$ 
11     $(v, i) := \text{frontier.peek}()$ 
12    if  $\neg i.\text{hasNext}()$  //  $v$  has no unused edge
13       $C[v] := \text{done}$ ;  $\text{frontier.dequeue}()$ 
14    else
15       $w := i.\text{next}()$  // Advance  $i$  in  $\text{adj}[v]$ 
16      if  $C[w] == \text{unseen}$ 
17         $\text{parent}[w] := v$ ;  $\text{distFromS}[w] := \text{distFromS}[v] + 1$ 
18         $C[w] := \text{active}$ ;  $\text{frontier.enqueue}((w, G.\text{adj}[w].\text{iterator}()))$ 
19      end if
20    end if
21  end while
```

---

- ▶  $\text{parent}$  stores a shortest-path tree/forest
- ▶ can retrieve shortest path to  $v$  from some vertex  $s \in S$  (backwards) by following  $\text{parent}[v]$  iteratively
- ▶ running time  $\Theta(n + m)$
- ▶ extra space  $\Theta(n)$

# Depth-First Search

- ▶ Maintain *frontier* in a **stack** (LIFO: last in, first out)
  - ▶ only consider  $S = \{s\}$
  - ▶ usual mode of operation: call  $\text{dfs}(v)$  for all *unseen*  $v$ , for  $v = 0, \dots, n - 1$

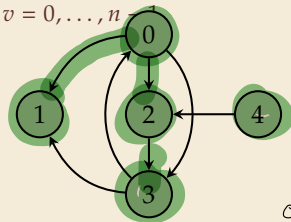
# Depth-First Search

- Maintain *frontier* in a **stack** (LIFO: last in, first out)

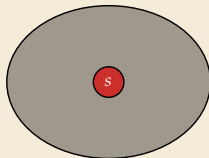
- only consider  $S = \{s\}$
- usual mode of operation: call  $\text{dfs}(v)$  for all *unseen*  $v$ , for  $v = 0, \dots, n$

- **Invariant:**

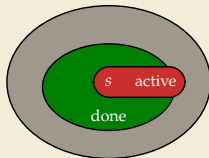
1. No edges from done to unseen vertices
2. All *done* <sup>& complete</sup> vertices are reached via a path from  $s$
3. The *active* vertices form a single **path** from  $s$



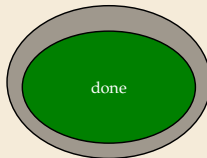
1 3  
0 0 0 0 0 0 4



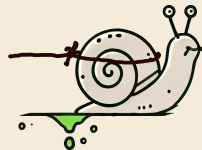
initial state



during traversal



final state



# Depth-First Search – Code

---

```
1 procedure dfsTraversal(G)
2   C[0..n] := unseen
3   for v := 0, ..., n - 1
4     if C[v] == unseen
5       dfs(G, v)
6
7 procedure dfs(G, s)
8   frontier := new Stack;
9   C[s] := active; frontier.push((s, G.adj[s].iterator()))
10  while ¬frontier.isEmpty()
11    (v, i) := frontier.top()
12    if ¬i.hasNext() // v has no unused edge
13      C[v] := done; frontier.pop(); postorderVisit(v)
14    else
15      w := i.next(); visitEdge(vw)
16      if C[w] == unseen
17        preorderVisit(w)
18        C[w] := active; frontier.push((w, G.adj[w].iterator()))
19      end if
20    end if
21  end while
```

---

- ▶ define *hooks* to implement further operations
  - ▶ preorder: visit  $v$  when made *active* (start of  $T(v)$ )
  - ▶ postorder: visit  $v$  when marked *done* (end of  $T(v)$ )
  - ▶ visitEdge: do something for every edge
- ▶ if needed, can store DFS forest via *parent* array



# Depth-First Search – Code

---

```
1 procedure dfsTraversal(G)
2   C[0..n] := unseen
3   for v := 0, ..., n - 1
4     if C[v] == unseen
5       dfs(G, v)
6
7 procedure dfs(G, s)
8   frontier := new Stack;
9   C[s] := active; frontier.push((s, G.adj[s].iterator()))
10  while ¬frontier.isEmpty()
11    (v, i) := frontier.top()
12    if ¬i.hasNext() // v has no unused edge
13      C[v] := done; frontier.pop(); postorderVisit(v)
14    else
15      w := i.next(); visitEdge(vw)
16      if C[w] == unseen
17        preorderVisit(w)
18        C[w] := active; frontier.push((w, G.adj[w].iterator()))
19      end if
20    end if
21  end while
```

- ▶ define *hooks* to implement further operations
  - ▶ preorder: visit  $v$  when made *active* (start of  $T(v)$ )
  - ▶ postorder: visit  $v$  when marked *done* (end of  $T(v)$ )
  - ▶ visitEdge: do something for every edge
- ▶ if needed, can store DFS forest via *parent* array
- ▶ running time  $\Theta(n + m)$
- ▶ extra space  $\Theta(n)$

# Simple DFS Application: Connected Components

- ▶ In an undirected graph, find all *connected components*.
  - ▶ **Given:** simple undirected  $G = (V, E)$
  - ▶ **Goal:** assign component ids  $CC[0..n)$ , s.t.  $CC[v] = CC[u]$  iff  $\exists$  path from  $v$  to  $u$

# Simple DFS Application: Connected Components


- ▶ In an undirected graph, find all *connected components*.
  - ▶ **Given:** simple undirected  $G = (V, E)$
  - ▶ **Goal:** assign component ids  $CC[0..n)$ , s.t.  $CC[v] = CC[u]$  iff  $\exists$  path from  $v$  to  $u$

---

```
1 procedure connectedComponents(G):
2   // undirected graph  $G = (V, E)$  with  $V = [0..n)$ 
3    $C[0..n) := \text{unseen}$ 
4    $\left\{ \begin{array}{l} CC[0..n) := \text{NONE} \\ id := 0 \end{array} \right.$ 
5   for  $v := 0, \dots, n - 1$ 
6     if  $C[v] == \text{unseen}$ 
7       dfs(G, v)
8        $id := id + 1$ 
9   return CC
```

---

```
12 procedure preorderVisit(v):
13    $CC[v] := id$ 
```



---

```
1 // same as before
2 procedure dfs(G, s)
3   frontier := new Stack;
4    $C[s] := \text{active};$  frontier.push((s, G.adj[s].iterator()))
5   while  $\neg \text{frontier.isEmpty}()$ 
6      $(v, i) := \text{frontier.top}()$ 
7     if  $\neg i.hasNext()$  // v has no unused edge
8        $C[v] := \text{done};$  frontier.pop()
9       postorderVisit(v)
10    else
11       $w := i.next();$  visitEdge(vw)
12      if  $C[w] == \text{unseen}$ 
13        preorderVisit(w)
14         $C[w] := \text{active}$ 
15        frontier.push((w, G.adj[w].iterator()))
16      end if
17    end if
18  end while
```

---

# Dijkstra's Algorithm & Prim's Algorithm

- ▶ On edge-weighted, we can use the tricolor traversal with a *priority queue* for frontier
- ▶ Dijkstra's Algorithm for shortest paths from  $s$  in digraphs with weakly positive edge weights
  - ▶ priority of vertex  $v$  = length of shortest path known so far from  $s$  to  $v$
- ▶ Prim's Algorithm for finding a minimum spanning tree
  - ▶ priority of vertex  $v$  = weight of cheapest edge connecting  $v$  to current tree

↪ Detailed discussion in Unit 11

## 9.5 Advanced Uses of DFS

# Properties of DFS

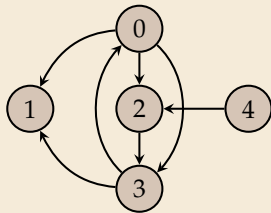
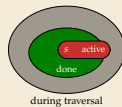
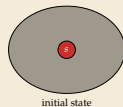
## ► Recall DFS Invariant 3:

The *active* vertices form a single **path** from  $s$

input graph  $G$

DFS forest

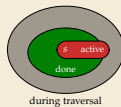
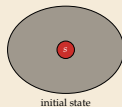
stack over time



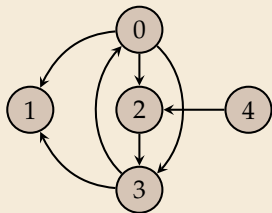
# Properties of DFS

## ► Recall DFS Invariant 3:

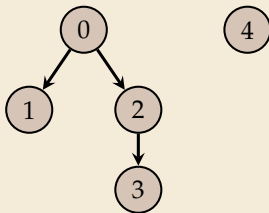
The *active* vertices form a single **path** from  $s$



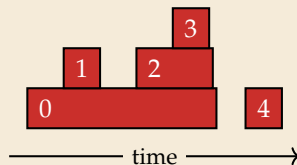
input graph  $G$



DFS forest



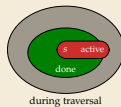
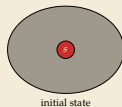
stack over time



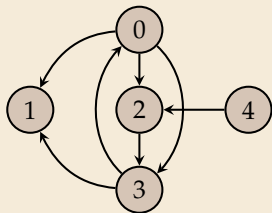
# Properties of DFS

## ► Recall DFS Invariant 3:

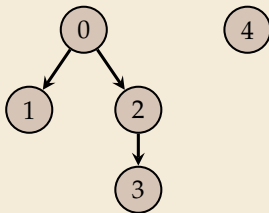
The **active** vertices form a single **path** from  $s$



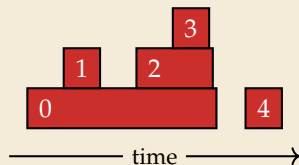
input graph  $G$



DFS forest



stack over time



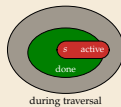
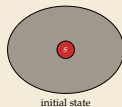
↪ Each vertex  $v$  spends *time interval*  $T(v)$  as **active** vertex



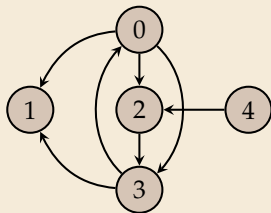
# Properties of DFS

## ► Recall DFS Invariant 3:

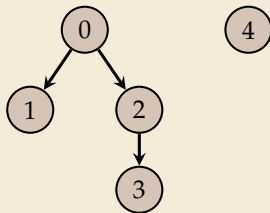
The **active** vertices form a single **path** from  $s$



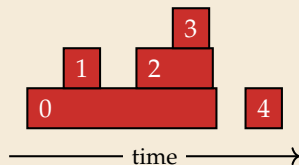
input graph  $G$



DFS forest



stack over time



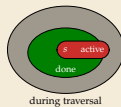
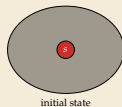
↪ Each vertex  $v$  spends *time interval*  $T(v)$  as **active** vertex

1. **frontier** is stack ↪  $\{T(v) : v \in V\}$  forms **laminar set family**: (“disjoint or contained”)  
either  $T(v) \cap T(w) = \emptyset$  or  $T(v) \subseteq T(w)$  or  $T(v) \supseteq T(w)$

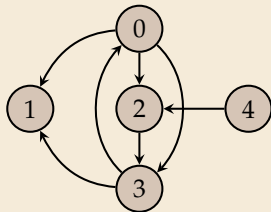
# Properties of DFS

## ► Recall DFS Invariant 3:

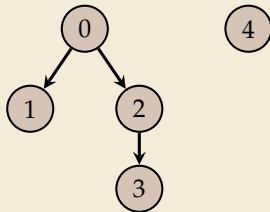
The **active** vertices form a single **path** from  $s$



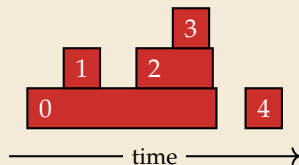
input graph  $G$



DFS forest



stack over time



↪ Each vertex  $v$  spends *time interval*  $T(v)$  as **active** vertex

1. **frontier** is stack ↪  $\{T(v) : v \in V\}$  forms **laminar set family**: (“disjoint or contained”)  
either  $T(v) \cap T(w) = \emptyset$  or  $T(v) \subseteq T(w)$  or  $T(v) \supseteq T(w)$

2. **Parenthesis Theorem**:  $T(v) \subseteq T(w)$  iff  $w$  is ancestor of  $v$  in DFS tree

‘ $\Rightarrow$ ’ during  $T(v)$ , all discovered vertices become descendants of  $v$

‘ $\Leftarrow$ ’  $T(v)$  covers  $v$ ’s entire subtree, which contains  $w$ ’s subtree

## Properties of DFS – Unseen-Path Theorem

- **Unseen-Path Theorem:** In a DFS forest of a (di)graph  $G$ ,  $w$  is a descendant of  $v$  iff at the time of  $\text{preorderVisit}(v)$ , there is a path from  $v$  to  $w$  using only *unseen* vertices.

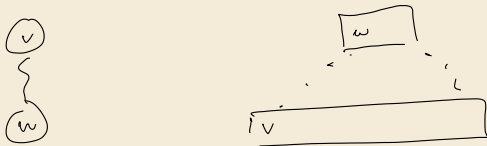


at the time  $v$  was made red

## Properties of DFS – Unseen-Path Theorem

- **Unseen-Path Theorem:** In a DFS forest of a (di)graph  $G$ ,  $w$  is a descendant of  $v$  iff at the time of `preorderVisit( $v$ )`, there is a path from  $v$  to  $w$  using only *unseen* vertices.

' $\Rightarrow$ '  $v = w$  trivial; for  $w$  strict descendant of  $v$ ,  $T(w) \subsetneq T(v)$  by the Parenthesis Thm

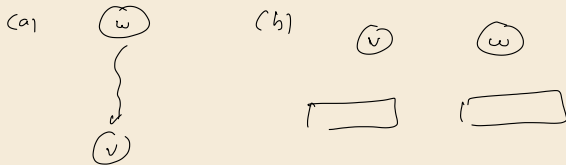


# Properties of DFS – Unseen-Path Theorem

- **Unseen-Path Theorem:** In a DFS forest of a (di)graph  $G$ ,  $w$  is a descendant of  $v$  iff at the time of  $\text{preorderVisit}(v)$ , there is a path from  $v$  to  $w$  using only *unseen* vertices.

' $\Rightarrow$ '  $v = w$  trivial; for  $w$  strict descendant of  $v$ ,  $T(w) \subsetneq T(v)$  by the Parenthesis Thm

' $\Leftarrow$ ' by contraposition. If  $v$  descendant of  $w$ ,  $w$  is **active** when in  $\text{preorderVisit}(v)$ . If neither is a descendant of the other,  $T(v) \cap T(w) = \emptyset$ , so one is strictly earlier.  $\Rightarrow$  no unseen path

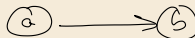


# Topological Sorting & Cycle Detection

- ▶ **Application:** Given a set of tasks with precedence constraints of the form " $a$  must be done before  $b$ ", can we find a legal ordering for all tasks?

↪ Model as directed graph!

- ▶ tasks are the vertices  $V$
- ▶ add an edge  $(a, b)$  when  $a$  must be done before  $b$



# Topological Sorting & Cycle Detection

- ▶ **Application:** Given a set of tasks with precedence constraints of the form “ $a$  must be done before  $b$ ”, can we find a legal ordering for all tasks?

↪ Model as directed graph!

- ▶ tasks are the vertices  $V$
  - ▶ add an edge  $(a, b)$  when  $a$  must be done before  $b$
- 
- ▶ **Definition:**  $R[0..n)$  is a *topological (order) ranking* of digraph  $G = (V, E)$  if
$$\forall (u, v) \in E : R[u] < R[v]$$

*think:  $R[v]$  time slot for  $v$*
  - ▶ **Lemma DAG iff topo:**  
A directed graph  $G$  has a topological ranking **iff** it does not contain a directed cycle.

# Topological Sorting & Cycle Detection

- ▶ **Application:** Given a set of tasks with precedence constraints of the form “ $a$  must be done before  $b$ ”, can we find a legal ordering for all tasks?

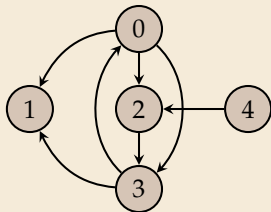
↪ Model as directed graph!

- ▶ tasks are the vertices  $V$
- ▶ add an edge  $(a, b)$  when  $a$  must be done before  $b$
- ▶ **Definition:**  $R[0..n)$  is a *topological (order) ranking* of digraph  $G = (V, E)$  if  $\forall (u, v) \in E : R[u] < R[v]$
- ▶ **Lemma DAG iff topo:**  
A directed graph  $G$  has a topological ranking **iff** it does not contain a directed cycle.
- ▶ **Topological Sorting**
  - ▶ **Given:** simple digraph  $G = (V, E)$
  - ▶ **Goal:** Compute topological ranking of vertices  $R[0..n)$  or output a directed cycle in  $G$ .
- ▶ Amazingly, can do all with one pass of DFS!



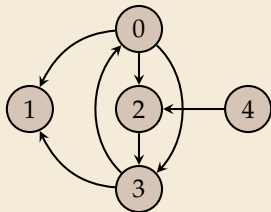
# DFS Edge Types

input digraph  $G$

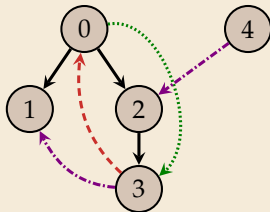


# DFS Edge Types

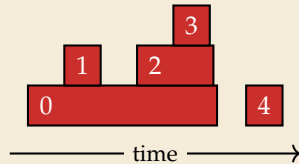
input digraph  $G$



DFS forest

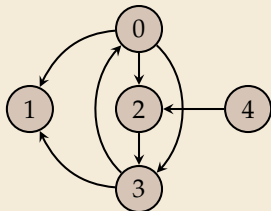


stack over time

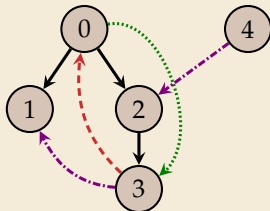


# DFS Edge Types

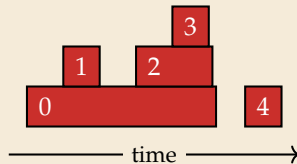
input digraph  $G$



DFS forest



stack over time



► During DFS traversal, an edge  $vw$  has one of these 4 types:

1. **tree edge:**  $\rightarrow w \in \text{unseen} \rightsquigarrow vw$  part of DFS forest.
2. **back edges:**  $--\rightarrow w \in \text{active}; \rightsquigarrow w$  points to ancestor of  $v$ .
3. **forward edges\*:**  $\cdots\rightarrow w \in \text{done} \wedge w$  is descendant of  $v$  in DFS tree.
4. **cross edges\*:**  $-\cdots\rightarrow w \in \text{done} \wedge w$  is not descendant of  $v$ .

\*only possible in directed graphs

**example:**

$(0, 1), (0, 2), (2, 3)$

$(3, 0)$

$(0, 3)$

$(3, 0)$

# Cycle Detection

If  $G$  contains a directed cycle, DFS will find a directed cycle:

- ▶ any back edge implies a cycle:
  - ▶ DFS visits an edge  $(v, w)$  where  $w \in \text{active}$ ,  $w$  is already on the stack
  - $\rightsquigarrow$  DFS tree contains path  $w \rightsquigarrow v$  and we have edge  $v \rightarrow w$ .



# Cycle Detection

If  $G$  contains a directed cycle, DFS will find a directed cycle:

- ▶ any back edge implies a cycle:
  - ▶ DFS visits an edge  $(v, w)$  where  $w \in \text{active}$ ,  $w$  is already on the stack
  - $\rightsquigarrow$  DFS tree contains path  $w \rightsquigarrow v$  and we have edge  $v \rightarrow w$ .
- ▶ conversely any cycle  $C[0..k]$  once reached must have some back edge or cross edge (tree and forward edges go from smaller to larger preorder index)
  - ▶ cannot be a cross edge since cycle is strongly connected  
all cycle vertices must be descendants of first reached cycle vertex
  - $\rightsquigarrow$  cycle contributes a back edge

# DFS Postorder Implementation

---

```
1 procedure dfsPostorder(G):
2   C[0..n) := unseen
3   P[0..n) := NONE; r := 0
4   parent[0..n) := NONE
5   cycle := NONE
6   for v := 0, ..., n - 1
7     if C[v] == unseen
8       dfs(G, v)
9   return (P, cycle)
10
11 procedure postorderVisit(v):
12   P[v] := r; r := r + 1
13
14 procedure visitEdge(vw):
15   if C[w] == active
16     if cycle ≠ NONE return
17   while v ≠ w
18     cycle.append(v)
19     v := parent[v]
20   cycle.append(v)
```

---

---

```
1 // dfs is as in CC but with parent
2 procedure dfs(G, s)
3   frontier := new Stack;
4   parent[s] := NONE;
5   C[s] := active; frontier.push((s, G.adj[s].iterator()))
6   while ¬frontier.isEmpty()
7     (v, i) := frontier.top()
8     if ¬i.hasNext() // v has no unused edge
9       C[v] := done; frontier.pop()
10      postorderVisit(v)
11   else
12     w := i.next() // Advance i in adj[v]
13     visitEdge(vw)
14     if C[w] == unseen
15       parent[w] := v;
16       preorderVisit(w)
17       C[w] := active; frontier.push((w, G.adj[w].iterator()))
18   end if
19 end if
20 end while
```

---

## DFS Postorder & Topological Sort

- **DFS Postorder:** The DFS postorder numbers is a numbering  $P[0..n)$  of  $V$  such that  $P[v] = r$  iff exactly  $r$  vertices reached state *done* before  $v$  in a DFS.

# DFS Postorder & Topological Sort

- ▶ **DFS Postorder:** The DFS postorder numbers is a numbering  $P[0..n)$  of  $V$  such that  $P[v] = r$  iff exactly  $r$  vertices reached state *done* before  $v$  in a DFS.
- ▶ **Lemma rev postorder:** directed acyclic graph  
Let  $G$  be a simple, connected DAG and  $R[0..n)$  a *reverse DFS postorder* of  $G$ , i. e.,  $R[v] = n - 1 - P[v]$  for a DFS postorder  $P[0..n)$ . Then  $R$  is a topological ranking of  $G$ .
- ▶ **Invariant:** If  $v \in$  *done* and  $(v, w) \in E$  then  $w \in$  *done* and  $R[v] < R[w]$ .
  - ▶ initially true (*done* =  $\emptyset$ )
  - ▶ upon postorderVisit( $v$ ), all outgoing edges  $vw$  lead to  $w \in$  *done* (Parenthesis Theorem)



# Topological Sorting & Cycle Detection – Summary

- ▶ Putting everything together we obtain topological sorting
  - ▶ can produce either the *ranking* or the *sequence of vertices* in topological order, whatever is more convenient

---

```
1 procedure topologicalRanking( $P$ ):  
2   ( $P[0..n], cycle$ ) := dfsPostorder( $G$ )  
3   if  $c \neq \text{NULL}$   
4     return NOT_A_DAG  
5    $R[0..n] := \text{NONE}$   
6   for  $v := 0, \dots, n - 1$   
7      $R[v] = n - 1 - P[v]$   
8   return  $P$ 
```

---

---

```
1 procedure topologicalSort( $P$ ):  
2   ( $P[0..n], cycle$ ) := dfsPostorder( $G$ )  
3   if  $c \neq \text{NULL}$   
4     return NOT_A_DAG  
5    $S[0..n] := \text{NONE}$   
6   for  $v := 0, \dots, n - 1$   
7      $S[P[v]] := v$  // reverse  
8   return  $S$ 
```

---

- ▶  $\Theta(n + m)$  time
- ▶  $\Theta(n)$  extra space