9 Graph Algorithms

8 December 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

version `2025-12-09 13:56`

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 characteristica.
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.

Outline

9 Graph Algorithms

9.1 Introduction & Definitions

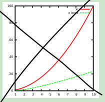
Clicker Question

List all matching pairs:

(A) Graph

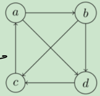
(B) Graf

(C) Grave

(1) 

(4) 

(2) 

(5) 

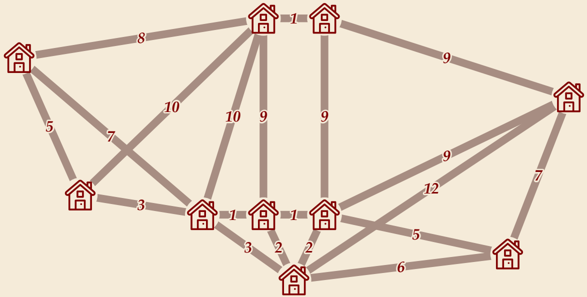
(3) 

(6) à

→ sli.do/cs566

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	*directed* graph (*digraph*)	*undirected* graph
≤ 1 edge between u and v	*simple* graph	*multigraph* / with *parallel* edges
edges can lead from v to v	with *loops*	(loop-free)
edges have weights	*(edge-) weighted* graph	*unweighted* 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*
  - $E \subseteq [V]^2$ a set of *edges*, which are 2-subsets of V: $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

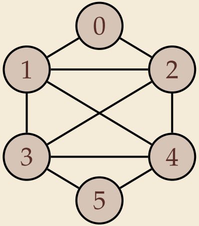
Graph Theory

- default: unweighted, undirected, loop-free & simple graphs
- *Graph* $G = (V, E)$ with
  - V a finite 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

$$\begin{aligned} V &= \{0,1,2,3,4,5\} \\ E &= \big\{\{0,1\},\{1,2\},\{1,4\},\{1,3\},\{0,2\}, \\ &\quad \{2,4\},\{2,3\},\{3,4\},\{3,5\},\{4,5\}\big\}. \end{aligned}$$

Graphical representation



like so . . .

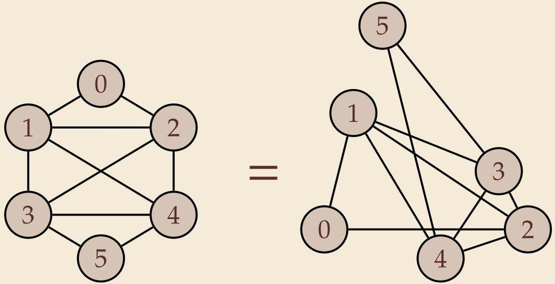
Graph Theory

- 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

$$\begin{aligned} V &= \{0, 1, 2, 3, 4, 5\} \\ E &= \big\{\{0,1\}, \{1,2\}, \{1,4\}, \{1,3\}, \{0,2\}, \\ &\qquad \{2,4\}, \{2,3\}, \{3,4\}, \{3,5\}, \{4,5\}\big\}. \end{aligned}$$

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

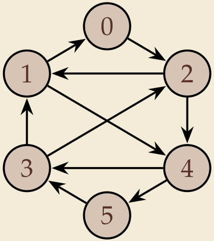
Digraphs

- 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

$$
\begin{aligned}
V &= \{0, 1, 2, 3, 4, 5\} \\
E &= \{(0,2), (1,0), (1,4), (2,1), (2,4), \\
&\quad (3,1), (3,2), (4,3), (4,5), (5,3)\}
\end{aligned}
$$

Graphical representation



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: $E(v)$ $= \{e : v \in e\}$
- 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* $d(v) = |E(v)|$

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 $d_{\text{in}}(v)$, $d_{\text{out}}(v)$

- *walk* („*Weg*“) $w[0..n]$ of length n: sequence of vertices with $\forall i \in [0..n) : w[i]w[i+1] \in E$
- *path* („*Pfad*“) p is a (vertex-) simple walk: no duplicate vertices except possibly its endpoints
- *edge-simple* walk: no edge used twice
- *cycle* c is a closed path, i. e., $c[0] = c[n]$

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* $d(v) = |E(v)|$

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 $d_{\text{in}}(v)$, $d_{\text{out}}(v)$

- *walk* („*Weg*“) $w[0..n]$ of length n: sequence of vertices with $\forall i \in [0..n) : w[i]w[i+1] \in E$
- *path* („*Pfad*“) p is a (vertex-) simple walk: no 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

- *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?

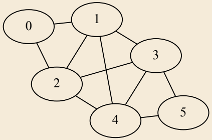
can vary a lot, despite superficial similarity of problems

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`
- *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

$\rightsquigarrow$ 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

$\rightsquigarrow$ need to choose a *representation* for graphs.

- which is better depends on the required operations

Key Operations:

- `isAdjacent(`u`,`v`)`
  Test whether $uv \in E$
- `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

$\rightsquigarrow$ need to choose a *representation* for graphs.

- which is better depends on the required operations

Key Operations:

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

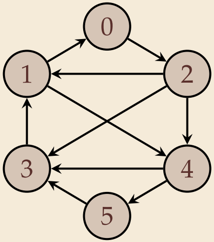
Conventions:

- (di)graph $G = (V, E)$ (omitted if clear from context)
- $n = |V|$, $m = |E|$
- in implementations assume $\boxed{V = [0..n)}$ (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 $A[u,v] = [uv \in E]$
  - 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:

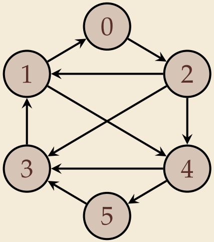


$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

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:



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

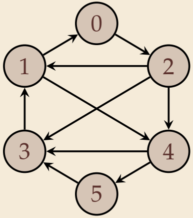
👍 `isAdjacent` in $O(1)$ time

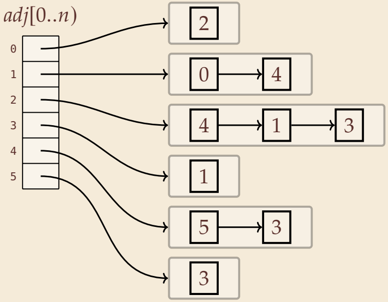
👎 $O(n^2)$ (bits of) space wasteful for sparse graphs

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

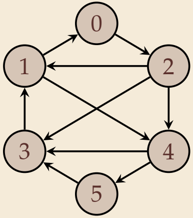
Adjacency List Representation

- Store a linked list of neighbors for each vertex v:
  - $adj[0..n)$ 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

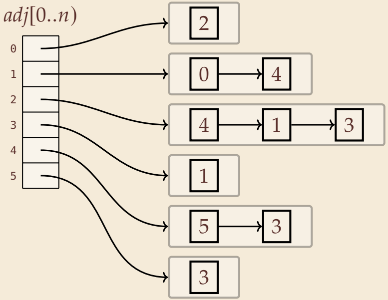




Adjacency List Representation



- 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

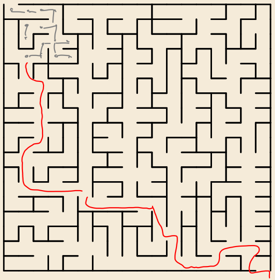
9.3 Graph Traversal

Generic Graph Traversal

- Plethora of graph algorithms can be expressed as a systematic exploration of a graph

  visiting all nodes & edges

  - 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
  - ...

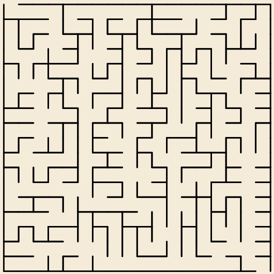


Generic Graph Traversal

- Plethora of graph algorithms can be expressed as a systematic exploration of a graph

  visiting all nodes & edges

  - 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
  - ...



$\rightsquigarrow$ Formulate generic traversal algorithm

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

Tricolor Graph Traversal

- maintain vertices in 3 (dynamic) sets
  - **Gray: unseen vertices**
    The traversal has not reached these vertices so far.
  - **Red: active vertices** (a.k.a. frontier („Rand“) of traversal)
    Vertices that have been reached and some unexplored edges remain;
    initially some selected start vertices $\underset{\smile}{S}$.
  - **Green: done vertices** (a.k.a. visited vertices)
    Visited vertices with all their edges explored.
- maintain edge status: either **unused** or **used**

Tricolor Graph Traversal

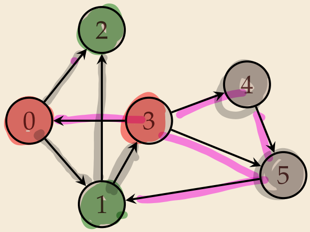
- maintain vertices in 3 (dynamic) sets
  - **Gray: unseen vertices**
    The traversal has not reached these vertices so far.
  - **Red: active vertices** (a.k.a. frontier („Rand“) of traversal)
    Vertices that have been reached and some unexplored edges remain;
    initially some selected start vertices S.
  - **Green: done vertices** (a.k.a. visited vertices)
    Visited vertices with all their edges explored.
- maintain edge status: either **unused** or **used**

Vertices *"want"* to be ***done***.
To do so, they turn neighbors ***active***.

Tricolor Graph Search:

- Repeat until no more changes:

(T1) Pick arbitrary ***active*** vertex v
(T2) If no more ***unused*** edges vw, mark v ***done*** (***D step***)
(T3) Else pick arbitrary ***unused*** edge vw, mark vw *used*
(T4) If w ***unseen***, mark w ***active*** (***A step***)



Tricolor Graph Traversal

- maintain vertices in 3 (dynamic) sets
  - **Gray: unseen vertices**
    The traversal has not reached these vertices so far.
  - **Red: active vertices** (a.k.a. frontier („Rand“) of traversal)
    Vertices that have been reached and some unexplored edges remain;
    initially some selected start vertices S.
  - **Green: done vertices** (a.k.a. visited vertices)
    Visited vertices with all their edges explored.

Vertices *"want"* to be ***done***.
To do so, they turn neighbors ***active***.

- maintain edge status: either **unused** or **used**

Tricolor Graph Search:

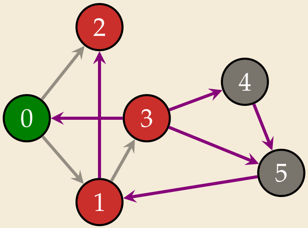
- Repeat until no more changes:

(T1) Pick arbitrary ***active*** vertex v

(T2) If no more ***unused*** edges vw, mark v ***done*** (***D step***)

(T3) Else pick arbitrary ***unused*** edge vw, mark vw ***used***

(T4) If w ***unseen***, mark w ***active*** (***A step***)



Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:
$v \in V$ is reachable from S *iff* $v \in$ ***done***.

$\exists$ path $P[0..\ell]$ $P[0] \in S \land P[\ell] = v$

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:
$v \in V$ is reachable from S *iff* $v \in \textit{done}$.

Proof:

- We prove the following invariant (next slide)
- **Invariant:** For every *done* or *active* vertex v, there exists a path from S to v.

$P[0..\ell] \quad P[0] \in S \;\wedge\; P[\ell] = v$

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:
$v \in V$ is reachable from S *iff* $v \in$ ***done***.

Proof:

- We prove the following invariant (next slide)
- **Invariant:** For every ***done*** or ***active*** vertex v, there exists a path from S to v.

$\rightsquigarrow$ in final state:

- $v \in$ ***done*** $\underset{\text{Inv.}}{\rightsquigarrow}$ $\exists$ path from S to v $\rightsquigarrow$ reachable from S

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:
$v \in V$ is reachable from S *iff* $v \in$ ***done***.

Proof:

- We prove the following invariant (next slide)
- **Invariant:** For every ***done*** or ***active*** vertex v, there exists a path from S to v.

$\rightsquigarrow$ in final state:

"$\Leftarrow$" ▶ $v \in$ ***done*** $\rightsquigarrow$ $\exists$ path from S to v $\rightsquigarrow$ reachable from S

"$\Rightarrow$" ▶ Let v be reachable from S, i. e., $\exists$ path $p[0..\ell]$ from $p[0] \in S$ to $p[\ell] = v$ of length $\ell \le n$.

Assume towards a contradiction $v \notin$ ***done***. $\rightsquigarrow$ $v \in$ ***unseen*** (***active*** $= \emptyset$ upon termination).

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:
$v \in V$ is reachable from S *iff* $v \in$ ***done***.

Proof:

- We prove the following invariant (next slide)
- **Invariant:** For every ***done*** or ***active*** vertex v, there exists a path from S to v.

$\rightsquigarrow$ in final state:

- $v \in$ ***done*** $\rightsquigarrow$ $\exists$ path from S to v $\rightsquigarrow$ reachable from S
- Let v be reachable from S, i. e., $\exists$ path $p[0..\ell]$ from $p[0] \in S$ to $p[\ell] = v$ of length $\ell \leq n$.

  Assume towards a contradiction $v \notin$ ***done***. $\rightsquigarrow$ $v \in$ ***unseen*** (***active*** $= \emptyset$ upon termination).
  Let v be such a vertex with *minimal distance* ℓ from S.

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:
$v \in V$ is reachable from S *iff* $v \in$ ***done***.

Proof:

- We prove the following invariant (next slide)
- **Invariant:** For every ***done*** or ***active*** vertex v, there exists a path from S to v.

$\rightsquigarrow$ in final state:

- $v \in$ ***done*** $\rightsquigarrow$ $\exists$ path from S to v $\rightsquigarrow$ reachable from S
- Let v be reachable from S, i. e., $\exists$ path $p[0..\ell]$ from $p[0] \in S$ to $p[\ell] = v$ of length $\ell \leq n$.

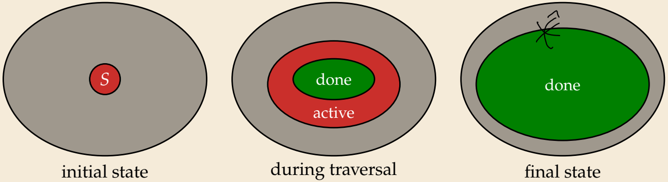
  Assume towards a contradiction $v \notin$ ***done***. $\rightsquigarrow$ $v \in$ ***unseen*** (***active*** $= \emptyset$ upon termination).
  Let v be such a vertex with *minimal distance* ℓ from S. $\rightsquigarrow$ $p[\ell-1] = w \in$ ***done***.

  But then wv ***unused*** and yet w was marked ***done*** ϟ (T2).

$p[\ell-1]\ p[\ell]$

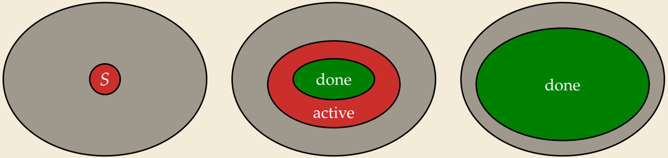
Generic Reachability – Invariant

- **Invariant:** For every *done* or *active* vertex v, there exists a path from S to v.



Generic Reachability – Invariant

- **Invariant:** For every *done* or *active* vertex v, there exists a path from S to v.



initial state — during traversal — final state

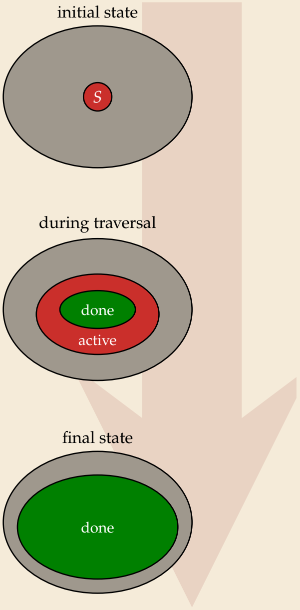
Proof:
By induction over the number of executed steps of tricolor traversal.

- **IB:** (1) no *done* vertices yet. (2) ✓ trivial path (w/o edges)
- **IH:** Invariant fulfilled for first k steps.
- **IS:** Step $k+1$ is either *A step* (T3)–(T4) or *D step* (T2)
  - *A step*: new *active* vertex w reached via vw with $v \in$ *active*
    $\exists$ path $P[0..\ell]$ with $P[0] \in S$ and $P[\ell] = v$ by IH $\rightsquigarrow$ path Pw from S to w.
  - *D step*: new *done* vertex previously was *active*, so $\exists$ path from S to v by IH.

$\blacksquare$

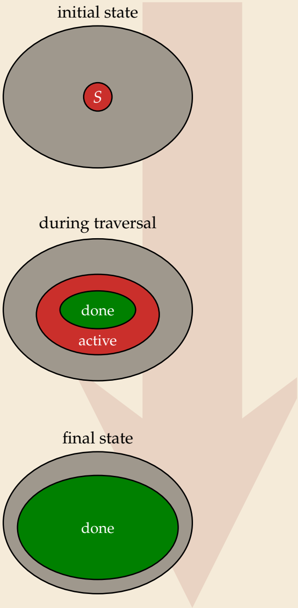
Generic Tricolor Graph Traversal – Code

```
procedure genericGraphTraversal(G, S):
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // Color array, all cells initialized to unseen
    for s ∈ S do C[s] := active end for
    unusedEdges := E
    while ∃v : C[v] == active
        v := nextActiveVertex()  // Freedom 1: Which frontier vertex?
        if ∄vw ∈ unusedEdges // no more edges from v ⇝ done with v
            C[v] := done
        else
            w := nextUnusedEdge(v)  // Freedom 2: Which of its edges?
            if C[w] == unseen
                C[w] := active
            unusedEdges.remove(vw)
        end if
    end while
```



Generic Tricolor Graph Traversal – Code

```
procedure genericGraphTraversal(G, S):
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // Color array, all cells initialized to unseen
    for s ∈ S do C[s] := active end for
    unusedEdges := E
    while ∃v : C[v] == active
        v := nextActiveVertex() // Freedom 1: Which frontier vertex?
        if ∄vw ∈ unusedEdges // no more edges from v ⇝ done with v
            C[v] := done
        else
            w := nextUnusedEdge(v) // Freedom 2: Which of its edges?
            if C[w] == unseen
                C[w] := active
            unusedEdges.remove(vw)
        end if
    end while
```



- Any implementation of nextActiveVertex() and nextUnusedEdge(v) suffices for correctness
- Choice depends on (and defines!) specific traversal-based graph algorithms

9.4 Breadth-First Search

Data Structures for Frontier

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

Data Structures for Frontier

- We need efficient support for
  - test $\exists v : C[v] = \textit{active}$, nextActiveVertex()
  - test $\exists vw \in \textit{unusedEdges}$, 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
    $\rightsquigarrow$ *unusedEdges*.remove(vw) doesn't need to do anything

Data Structures for Frontier

- We need efficient support for
  - test $\exists v : C[v] = \textit{active}$, nextActiveVertex()
  - test $\exists vw \in \textit{unusedEdges}$, 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
    $\rightsquigarrow$ *unusedEdges*.remove(vw) doesn't need to do anything
  - Implement $\exists v : C[v] = \textit{active}$ via *frontier*.isEmpty()
  - Implement $\exists vw \in \textit{unusedEdges}$ via i.hasNext() assuming $(v, i) \in \textit{frontier}$
  - Implement nextUnusedEdge(v) via i.next() assuming $(v, i) \in \textit{frontier}$

$\rightsquigarrow$ all operations apart from nextActiveVertex() in $O(1)$ time

$\rightsquigarrow$ *frontier* requires $O(n)$ extra space

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)
- **Unweighted shortest path distance:**

$$dist_G(s,t) = \min\{\ell : \exists \text{ path } P[0..\ell] : P[0] = s \wedge P[\ell] = t\} \cup \{\infty\}$$

$$dist_G(S,t) = \min_{s \in S} dist_G(s,t)$$

$P[0] \rightarrow P[1] \rightarrow P[2]$

Breadth-First Search

- Maintain *frontier* in a **queue** (FIFO: first in, first out)
- **Unweighted shortest path distance:**

$$dist_G(s,t) = \min\left\{\ell : \exists \text{ path } P[0..\ell] : P[0] = s \wedge P[\ell] = t\right\} \cup \{\infty\}$$
$$dist_G(S,t) = \min_{s \in S} dist_G(s,t)$$

- Like generic tricolor search, BFS finds vertices reachable from S. But it does more:

Theorem 9.2 (BFS Correctness)

A BFS from $S \subseteq V$ reaches all vertices reachable from S via a shortest path. ◀

Breadth-First Search

- Maintain *frontier* in a **queue** (FIFO: first in, first out)
- **Unweighted shortest path distance:**

$$dist_G(s,t) = \min\left\{\ell : \exists \text{ path } P[0..\ell] : P[0] = s \wedge P[\ell] = t\right\} \cup \{\infty\}$$

$$dist_G(S,t) = \min_{s \in S} dist_G(s,t)$$

- Like generic tricolor search, BFS finds vertices reachable from S. But it does more:

Theorem 9.2 (BFS Correctness)

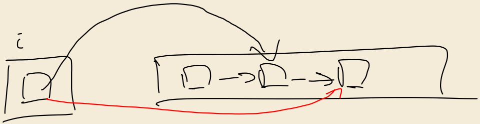
A BFS from $S \subseteq V$ reaches all vertices reachable from S via a shortest path. ◀

- To preserve paths, we collect extra information during traversal:
  - $parent[v]$ stores predecessor on path from S via which v was first reached (made ***active***)
  - $distFromS[v]$ stores the length of this path

Breadth-First Search – Code

```
procedure bfs(G, S):
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // New array initialized to all unseen
    frontier := new Queue;
    parent[0..n) := NOT_VISITED;  distFromS[0..n) := ∞
    for s ∈ S
        parent[s] := NONE;  distFromS[s] := 0
        C[s] := active;  frontier.enqueue((s, G.adj[s].iterator()))
    end for
    while ¬frontier.isEmpty()
        (v, i) := frontier.peek()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.dequeue()
        else
            w := i.next() // Advance i in adj[v]
            if C[w] == unseen
                parent[w] := v;  distFromS[w] := distFromS[v] + 1
                C[w] := active;  frontier.enqueue((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

head of adj(s)



i.next

Breadth-First Search – Code

```
procedure bfs(G, S):
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // New array initialized to all unseen
    frontier := new Queue;
    parent[0..n) := NOT_VISITED;  distFromS[0..n) := ∞
    for s ∈ S
        parent[s] := NONE;  distFromS[s] := 0
        C[s] := active;  frontier.enqueue((s, G.adj[s].iterator()))
    end for
    while ¬frontier.isEmpty()
        (v, i) := frontier.peek()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.dequeue()
        else
            w := i.next() // Advance i in adj[v]
            if C[w] == unseen
                parent[w] := v;  distFromS[w] := distFromS[v] + 1
                C[w] := active;  frontier.enqueue((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

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

Breadth-First Search – Code

```
procedure bfs(G, S):
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // New array initialized to all unseen
    frontier := new Queue;
    parent[0..n) := NOT_VISITED;  distFromS[0..n) := ∞
    for s ∈ S
        parent[s] := NONE;  distFromS[s] := 0
        C[s] := active;  frontier.enqueue((s, G.adj[s].iterator()))
    end for
    while ¬frontier.isEmpty()
        (v, i) := frontier.peek()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.dequeue()
        else
            w := i.next() // Advance i in adj[v]
            if C[w] == unseen
                parent[w] := v;  distFromS[w] := distFromS[v] + 1
                C[w] := active;  frontier.enqueue((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

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

Breadth-First Search – Correctness

- BFS correctness directly follows from the following invariant.

- **BFS Invariant:**

  fewest edges

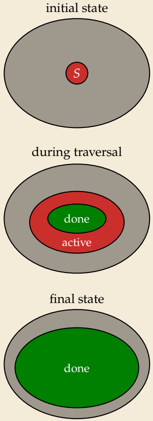
  1. All ***done*** or ***active*** vertices were reached via a **shortest path** from S
  2. Vertices enter and leave *frontier* in order of increasing $\underbrace{\text{distance from } S}_{\delta(v)}$

Proof:

By induction over number of steps. Abbreviate $\delta(v) := dist_G(S, v)$

- **IB:** (1) only S ***active***, reached via path of length 0.
  (2) only S in *frontier*, minimal by δ. ✓

- **IH:** Invariant fulfilled for first k steps.



Theorem 9.2 (BFS Correctness)
A BFS from $S \subseteq V$ reaches all vertices reachable from S via a shortest path.

Breadth-First Search – Correctness

- BFS correctness directly follows from the following invariant.
- **BFS Invariant:**

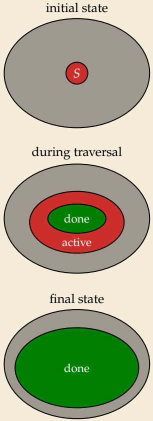
  fewest edges

  1. All ***done*** or ***active*** vertices were reached via a **shortest path** from S
  2. Vertices enter and leave *frontier* in order of increasing distance from S

Proof:

By induction over number of steps. Abbreviate $\delta(v) := dist_G(S, v)$

- **IB:** (1) only S ***active***, reached via path of length 0.
  (2) only S in *frontier*, minimal by δ. ✓
- **IH:** Invariant fulfilled for first k steps.
- **IS:** Step $k+1$ can be an *A step* or a *D step* on $v \in$ ***active***
  - *D step:* v moved from ***active*** to ***done*** $\rightsquigarrow$ (1) unchanged. (IH)
    By IH, v *entered frontier* at correct time, queue keeps order $\rightsquigarrow$ (2) ✓



Breadth-First Search – Correctness

- BFS correctness directly follows from the following invariant.
- **BFS Invariant:**

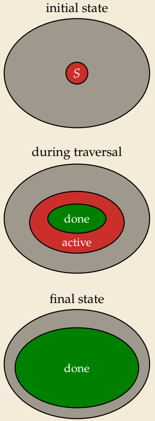
  fewest edges

  1. All ***done*** or ***active*** vertices were reached via a **shortest path** from S
  2. Vertices enter and leave *frontier* in order of increasing distance from S

Proof:

By induction over number of steps. Abbreviate $\delta(v) := dist_G(S, v)$

- **IB:** (1) only S ***active***, reached via path of length 0.
  (2) only S in *frontier*, minimal by δ. ✓
- **IH:** Invariant fulfilled for first k steps.
- **IS:** Step $k+1$ can be an *A step* or a *D step* on $v \in$ ***active***
  - *D step:* v moved from ***active*** to ***done*** $\rightsquigarrow$ (1) unchanged.
    By IH, v *entered* *frontier* at correct time, queue keeps order $\rightsquigarrow$ (2) ✓
  - *A step, (i):* $vw \in$ *unusedEdges* leads to $w \in$ ***active*** $\cup$ ***done***
    no changes, (1) and (2) ✓



Breadth-First Search – Correctness [2]

Proof (cont.):

- *A step, (ii):* $vw \in \mathit{unusedEdges}$ leads to $w \in \mathit{unseen}$
  $\rightsquigarrow$ w is now marked ***active*** and enqueued in *frontier*.

Breadth-First Search – Correctness [2]

Proof (cont.):

- *A step, (ii):* $vw \in \mathit{unusedEdges}$ leads to $w \in \mathit{unseen}$
  $\rightsquigarrow$ w is now marked ***active*** and enqueued in *frontier*.

  By IH, we reached v by shortest path of $\delta(v)$ edges,
  and any node u with $\delta(u) < \delta(v)$ is ***done*** (since dequeued before v).

Breadth-First Search – Correctness [2]

Proof (cont.):

- *A step, (ii):* $vw \in \mathit{unusedEdges}$ leads to $w \in \mathit{unseen}$
  $\rightsquigarrow$ w is now marked ***active*** and enqueued in *frontier*.

  By IH, we reached v by shortest path of $\delta(v)$ edges,
  and any node u with $\delta(u) < \delta(v)$ is ***done*** (since dequeued before v).

$\rightsquigarrow$ $\delta(w) = \delta(v) + 1$, and we reach w with shortest path $\rightsquigarrow$ (1) ✓

$s \rightsquigarrow v \to w$

Breadth-First Search – Correctness [2]

Proof (cont.):

- *A step, (ii):* $vw \in \mathit{unusedEdges}$ leads to $w \in \mathit{unseen}$
  $\leadsto$ w is now marked ***active*** and enqueued in *frontier*.

  By IH, we reached v by shortest path of $\delta(v)$ edges,
  and any node u with $\delta(u) < \delta(v)$ is ***done*** (since dequeued before v).

$\leadsto$ $\delta(w) = \delta(v) + 1$, and we reach w with shortest path $\leadsto$ (1) ✓

$< \delta(w)$

- It remains to show that any x with $\delta(x) = \delta(v)$ is ***active*** or ***done*** by now;
  then adding w to *frontier* respects the order by δ.

Breadth-First Search – Correctness [2]

Proof (cont.):

- *A step, (ii):* $vw \in \textit{unusedEdges}$ leads to $w \in \textit{unseen}$
  $\rightsquigarrow$ w is now marked ***active*** and enqueued in *frontier*.

  By IH, we reached v by shortest path of $\delta(v)$ edges,
  and any node u with $\delta(u) < \delta(v)$ is ***done*** (since dequeued before v).

$\rightsquigarrow$ $\delta(w) = \delta(v) + 1$, and we reach w with shortest path $\rightsquigarrow$ (1) ✓

- It remains to show that any x with $\delta(x) = \delta(v)$ is ***active*** or ***done*** by now;
  then adding w to *frontier* respects the order by δ. $S \rightsquigarrow u \to x$

  Any shortest path from S to $x \notin S$ must go via some u with $\delta(u) < \delta(v)$, so u is ***done***.
  $\rightsquigarrow$ all edges from u, including ux, have been ***used***, thus x is ***active*** or ***done***.

  $\Rightarrow$ (2) ∎

9.5 Depth-First Search

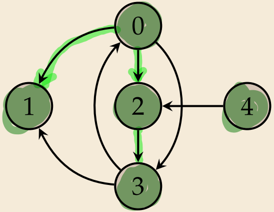
Depth-First Search

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

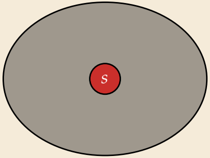
Depth-First Search

3
1 2 2 2
0 0 0 0 0 0 0 4
time

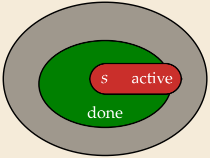
- Maintain *frontier* in a **stack** (LIFO: last in, first out)
  - only consider $S = \{s\}$
  - usual mode of operation: call `dfs`(v) for all *unseen* v, for $v = 0, \ldots, n-1$



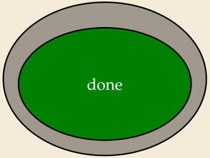
- **DFS Invariant:**
  1. All ***done*** or ***active*** vertices are reached via a path from s
  2. The ***active*** vertices form a **single path** from s



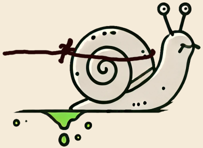
initial state



during traversal



final state



Depth-First Search – Code

```
procedure dfsTraversal(G):
    C[0..n) := unseen
    for v := 0, ..., n − 1
        if C[v] == unseen
            dfs(G, v)

procedure dfs(G, s):
    frontier := new Stack;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop();  postorderVisit(v)
        else
            w := i.next();  visitEdge(vw)
            if C[w] == unseen
                preorderVisit(w)
                C[w] := active;  frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    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

```
procedure dfsTraversal(G):
    C[0..n) := unseen
    for v := 0, ..., n − 1
        if C[v] == unseen
            dfs(G, v)

procedure dfs(G, s):
    frontier := new Stack;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop();  postorderVisit(v)
        else
            w := i.next();  visitEdge(vw)
            if C[w] == unseen
                preorderVisit(w)
                C[w] := active;  frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    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

```
procedure connectedComponents(G):
    // undirected graph G = (V, E) with V = [0..n)
    C[0..n) := unseen
    CC[0..n) := NONE
    id := 0
    for v := 0, ..., n − 1
        if C[v] == unseen
            dfs(G, v)
            id := id + 1
    return CC

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

```
// same as before
procedure dfs(G, s):
    frontier := new Stack;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop()
            postorderVisit(v)
        else
            w := i.next();  visitEdge(vw)
            if C[w] == unseen
                preorderVisit(w)
                C[w] := active
                frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

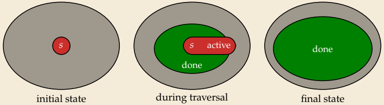
Dijkstra's Algorithm & Prim's Algorithm

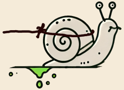
- On edge-weighted graphs, we can use tricolor traversal with a *priority queue* as *frontier*
- Dijkstra's Algorithm for shortest paths from s in digraphs with weakly positive edge weights
  - priority of vertex v = length of shortest path known so far from s to v
- Prim's Algorithm for finding a minimum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

$\rightsquigarrow$ Detailed discussion in Unit 11

9.6 Advanced Uses of DFS I

Properties of DFS



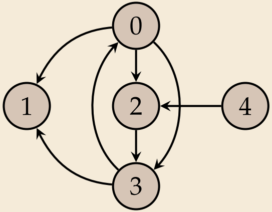


- **Recall DFS Invariant (2):**
  The *active* vertices form a **single path** from s

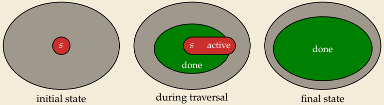
input graph G

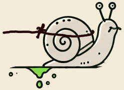
DFS forest

stack over time



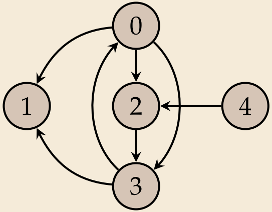
Properties of DFS



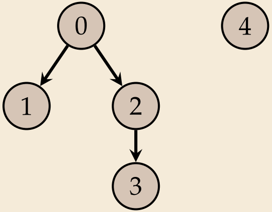


- **Recall DFS Invariant (2):**
  The *active* vertices form a **single path** from s

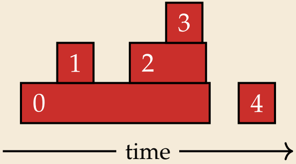
input graph G



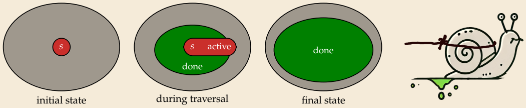
DFS forest



stack over time

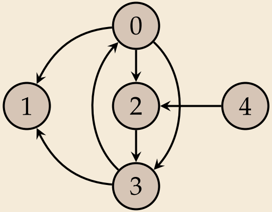


Properties of DFS

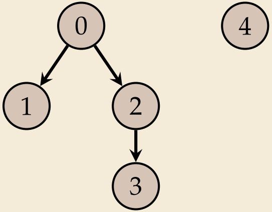


- **Recall DFS Invariant (2):**
  The *active* vertices form a **single path** from s

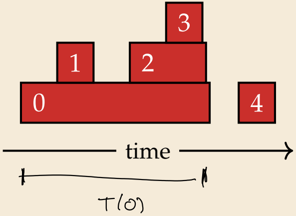
input graph G



DFS forest

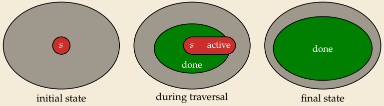


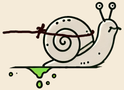
stack over time



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

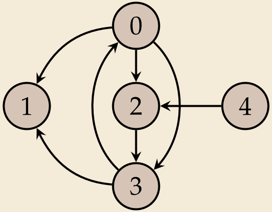
Properties of DFS



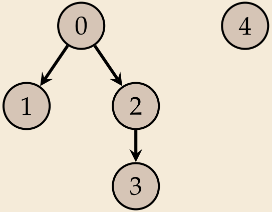


- **Recall DFS Invariant (2):**
  The ***active*** vertices form a **single path** from s

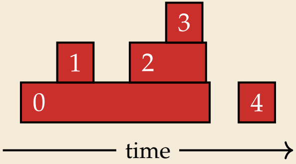
input graph G



DFS forest



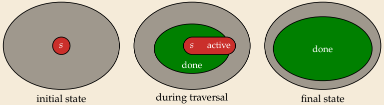
stack over time

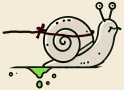


$\rightsquigarrow$ Each vertex v spends *time interval* $T(v)$ as ***active*** vertex

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

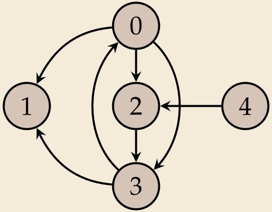
Properties of DFS



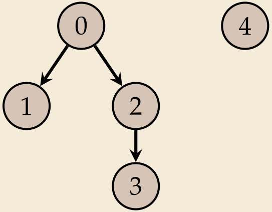


- **Recall DFS Invariant (2):**
  The ***active*** vertices form a **single path** from s

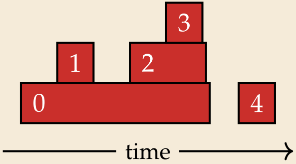
input graph G



DFS forest



stack over time



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

1. *frontier* is stack $\rightsquigarrow$ $\{T(v) : v \in V\}$ 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:** $T(v) \supseteq T(w)$ **iff** v is ancestor of w 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 `preorderVisit`(v), there is a path from v to w using only *unseen* vertices.