

9

Graph Algorithms

8 December 2025

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.

Outline

9 Graph Algorithms

- 9.1 Introduction & Definitions
- 9.2 Graph Representations
- 9.3 Graph Traversal
- 9.4 Breadth-First Search
- 9.5 Depth-First Search
- 9.6 Advanced Uses of DFS I
- 9.7 Advanced Uses of DFS II
- 9.8 Network flows
- 9.9 The Ford-Fulkerson Method
- 9.10 The Edmonds-Karp Algorithm

9.1 Introduction & Definitions

Clicker Question

List all matching pairs:



(A) Graph

(B) Graf

(C) Grave

(1)



(4)



(2)



(5)



(3)



~~(6)~~

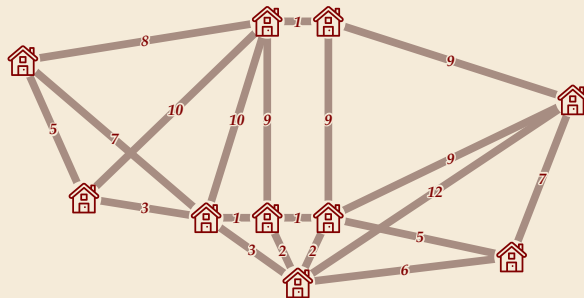
à



→ 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	<i>directed</i> graph (<i>digraph</i>)	<i>undirected</i> graph
≤ 1 edge between u and v	<i>simple</i> graph	<i>multigraph</i> / with <i>parallel</i> edges
edges can lead from v to v	with <i>loops</i>	(loop-free)
edges have weights	(<i>edge-</i>) <i>weighted</i> graph	<i>unweighted</i> 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

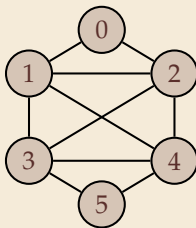
- ▶ default: unweighted, undirected, loop-free & simple graphs
- ▶ *Graph* $G = (V, E)$ with
 - ▶ V a finite set of *vertices*
 - ▶ $E \subseteq [V]^2$ a set of *edges*, which are 2-subsets of V : $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

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

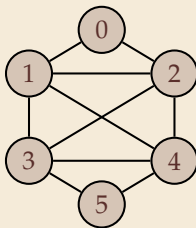
- ▶ default: unweighted, undirected, loop-free & simple graphs
- ▶ Graph $G = (V, E)$ with
 - ▶ V a finite of *vertices*
 - ▶ $E \subseteq [V]^2$ a set of *edges*, which are 2-subsets of V : $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

Example

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

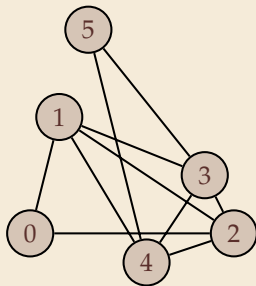
$$E = \{\{0, 1\}, \{1, 2\}, \{1, 4\}, \{1, 3\}, \{0, 2\}, \\ \{2, 4\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}.$$

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 set 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

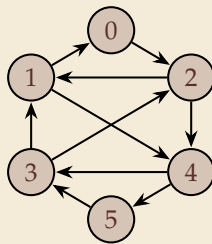
- ▶ default digraph: unweighted, loop-free & simple
- ▶ *Digraph (directed graph)* $G = (V, E)$ with
 - ▶ V a finite of *vertices*
 - ▶ $E \subseteq V^2 \setminus \{(v, v) : v \in V\}$ a set of (*directed*) *edges*,
 $V^2 = V \times V = \{(x, y) : x \in V \wedge y \in V\}$ 2-tuples / ordered pairs over 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 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* $d(v) = |E(v)|$
- ▶ 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., $c[0] = c[n]$

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* $d(v) = |E(v)|$
- ▶ *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

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?

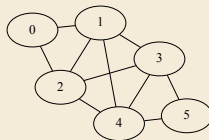
↖ can vary a lot, despite superficial similarity of problems

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
- ▶ *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

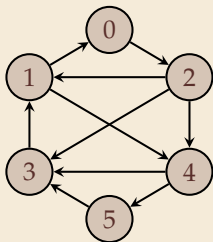
Conventions:

- ▶ (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 $A[u, v] = [uv \in E]$
 - ▶ 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:

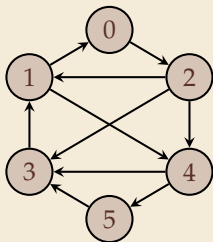


$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

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



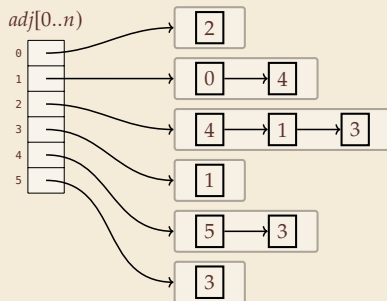
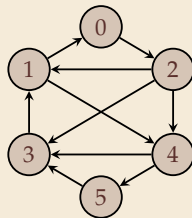
$O(n^2)$ (bits of) space wasteful for sparse graphs



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

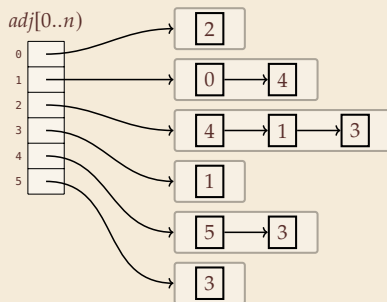
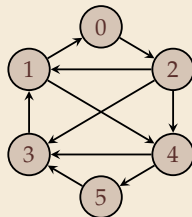
Adjacency List Representation

- 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

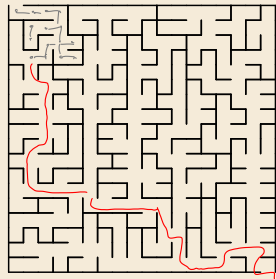
- ▶ Note that adj matrix and lists for undirected graphs effectively are representation of directed graph with directed edges both ways
 - ▶ conceptually still important to distinguish!
- ▶ multigraphs, loops, edge weights all naturally supported in adj lists
 - ▶ good if we allow and use them
 - ▶ but 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
 - ▶ 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
 - ▶ ...

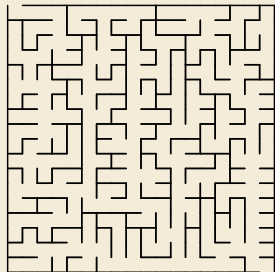
visiting all nodes & edges



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

- ▶ 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 $\underline{S}$.
 - ▶ **Green: done vertices** (a.k.a. visited vertices)
Visited vertices with all their edges explored.
- ▶ maintain edge status: either **unused** or **used**

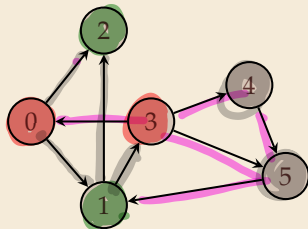
Tricolor Graph Traversal

- ▶ 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*)



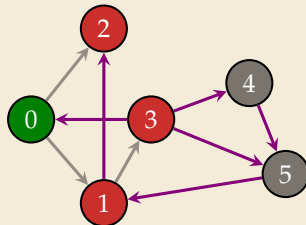
Tricolor Graph Traversal

- ▶ 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.
- ▶ 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*)



Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:

$v \in V$ is reachable from S iff $v \in \text{done}$.

$$\exists \text{ path } P[0..l] \quad P[0] \in S \wedge P[l] = v$$

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:

$v \in V$ is reachable from S iff $v \in \text{done}$.

Proof:

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

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

Generic Reachability – Correctness

Theorem 9.1 (Generic Reachability)

When Tricolor Graph Search terminates, the following holds:

$v \in V$ is reachable from S iff $v \in \text{done}$.

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:

$\Leftarrow$ ▶ $v \in \text{done} \rightsquigarrow \exists \text{ path from } S \text{ to } v \rightsquigarrow \text{reachable from } S$

$\Rightarrow$ ▶ Let v be reachable from S , i. e., $\exists \text{ path } p[0..l] \text{ from } p[0] \in S \text{ to } p[l] = v \text{ of length } l \leq n$.

Assume towards a contradiction $v \notin \text{done} \rightsquigarrow v \in \text{unseen}$ (*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 $v \in \text{done}$.

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:

- ▶ $v \in \text{done} \rightsquigarrow \exists$ 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 $v \notin \text{done}$. $\rightsquigarrow v \in \text{unseen}$ (*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 $v \in \text{done}$.

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:

- ▶ $v \in \text{done} \rightsquigarrow \exists$ 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 $v \notin \text{done}$. $\rightsquigarrow v \in \text{unseen}$ (*active* = $\emptyset$ upon termination).

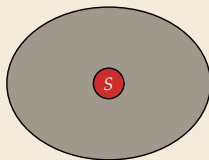
Let v be such a vertex with *minimal distance* ℓ from S . $\rightsquigarrow p[\ell - 1] = w \in \text{done}$.

But then $wv \in E$, $v \in \text{unseen}$, and yet w was marked *done* ⚡ (T2).

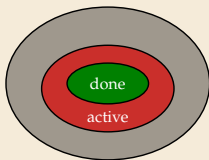
$$\begin{array}{c} \uparrow \\ p[\ell-1] \quad p[\ell] \end{array}$$

Generic Reachability – Invariant

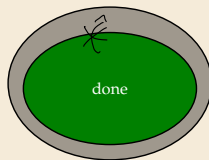
- **Invariant:** For every *done* or *active* vertex v , there exists a path from S to v .



initial state



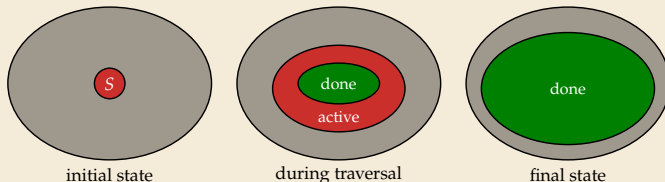
during traversal



final state

Generic Reachability – Invariant

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



Proof:

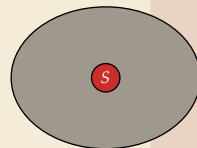
By induction over the number of executed steps of tricolor traversal.

- **IB:** (1) no *done* vertices yet. (2) $\checkmark$ 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 $v \in \text{active}$
 $\exists$ 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.

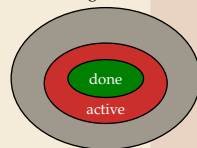
Generic Tricolor Graph Traversal – Code

```
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         $\text{unusedEdges.remove}(vw)$   
15    end if  
16  end while
```

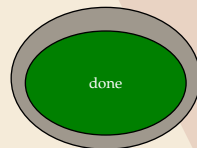
initial state



during traversal

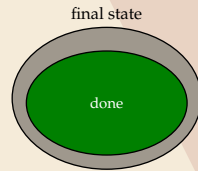
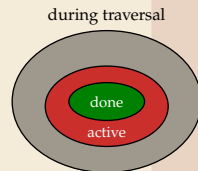
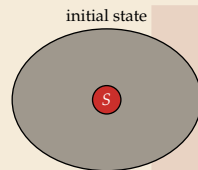


final state



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         $\text{unusedEdges.remove}(vw)$   
15    end if  
16  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)`
 - ▶ `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
 - $\rightsquigarrow$ `unusedEdges.remove(vw)` doesn't need to do anything

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
 - $\rightsquigarrow$ `unusedEdges.remove(vw)` doesn't need to do anything
 - ▶ 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

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:**

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

$$\text{dist}_G(S, t) = \min_{s \in S} \text{dist}_G(s, t)$$

- 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:**

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

$$\text{dist}_G(S, t) = \min_{s \in S} \text{dist}_G(s, t)$$

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

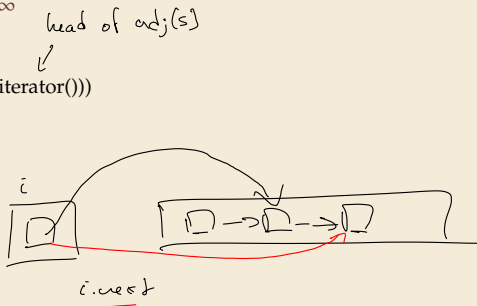
Theorem 9.2 (BFS Correctness)

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

```
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
```

- ▶ 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
```

- ▶ 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 $\Theta(n)$

Breadth-First Search – Correctness

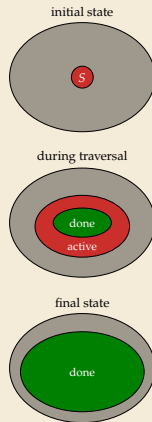
- ▶ BFS correctness directly follows from the following invariant.
- ▶ **BFS Invariant:**
 1. All *done* or *active* vertices were reached via a **shortest path** from S
 2. Vertices enter and leave *frontier* in order of increasing distance from S

fewest edges

Proof:

By induction over number of steps. Abbreviate $\delta(v) := \text{dist}_G(S, v)$

- ▶ **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.

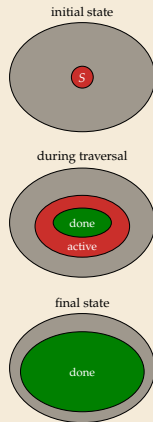
Breadth-First Search – Correctness

- ▶ BFS correctness directly follows from the following invariant.
- ▶ **BFS Invariant:**
 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 $\delta(v) := \text{dist}_G(S, v)$

- ▶ **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 $v \in \text{active}$
 - ▶ *D step:* v moved from *active* to *done* $\rightsquigarrow$ (1) unchanged. ($\mathbb{I} \nVdash$)
By IH, v entered *frontier* at correct time, queue keeps order $\rightsquigarrow$ (2) ✓



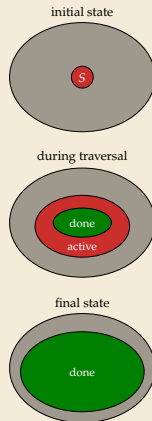
Breadth-First Search – Correctness

- ▶ BFS correctness directly follows from the following invariant.
- ▶ **BFS Invariant:**
 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 $\delta(v) := \text{dist}_G(S, v)$

- ▶ **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 $v \in \text{active}$
 - ▶ 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): $vw \in \text{unusedEdges}$ leads to $w \in \text{active} \cup \text{done}$
no changes, (1) and (2) ✓



Breadth-First Search – Correctness [2]

Proof (cont.):

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



Breadth-First Search – Correctness [2]

Proof (cont.):

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

By IH, we reached v by shortest path of $\underline{\delta(v)}$ 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): $vw \in \text{unusedEdges}$ leads to $w \in \text{unseen}$
 $\rightsquigarrow$ 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) ✓



Breadth-First Search – Correctness [2]

Proof (cont.):

- A step, (ii): $vw \in \text{unusedEdges}$ leads to $w \in \text{unseen}$
 $\rightsquigarrow$ 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 .

Breadth-First Search – Correctness [2]

Proof (cont.):

- A step, (ii): $vw \in \text{unusedEdges}$ leads to $w \in \text{unseen}$
 $\rightsquigarrow$ 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 . $S \rightsquigarrow u \rightarrow x$

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**.

$\rightsquigarrow \Rightarrow (2) \text{ is } \checkmark$

9.5 Depth-First Search

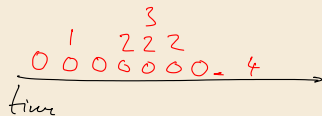
Depth-First Search

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

Depth-First Search

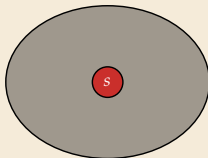
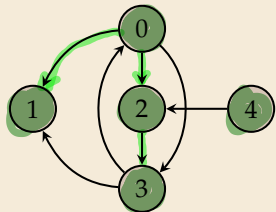
- Maintain *frontier* in a **stack** (LIFO: last in, first out)

- only consider $S = \{s\}$
- usual mode of operation: call $\text{dfs}(v)$ for all *unseen* v , for $v = 0, \dots, n-1$

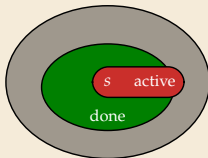


- **DFS Invariant:**

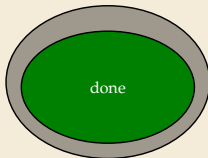
1. All *done* or *active* vertices are reached via a path from s
2. The *active* vertices form a **single path** from s



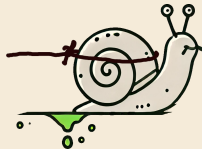
initial state



during traversal



final state



Depth-First Search – Code

```
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    $CC[0..n) := \text{NONE}$ 
5    $id := 0$ 
6   for  $v := 0, \dots, n - 1$ 
7     if  $C[v] == \text{unseen}$ 
8       dfs(G, v)
9        $id := id + 1$ 
10  return CC
11
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 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

↪ Detailed discussion in Unit 11

9.6 **Advanced Uses of DFS I**

Properties of DFS

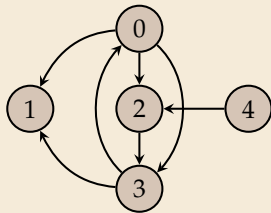
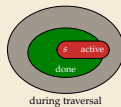
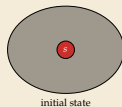
► Recall DFS Invariant (2):

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

input graph G

DFS forest

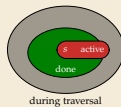
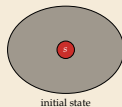
stack over time



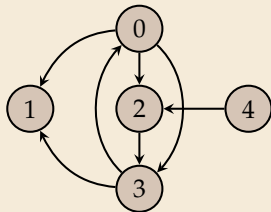
Properties of DFS

► Recall DFS Invariant (2):

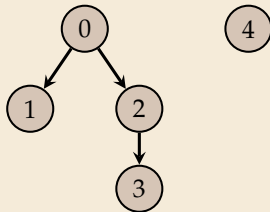
The *active* vertices form a **single path** from s



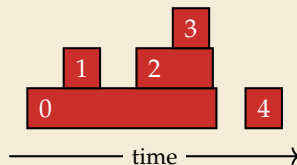
input graph G



DFS forest



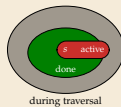
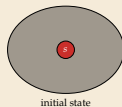
stack over time



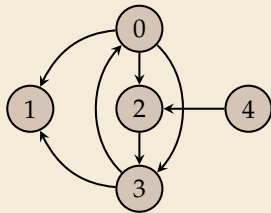
Properties of DFS

► Recall DFS Invariant (2):

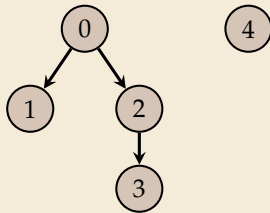
The **active** vertices form a **single path** from s



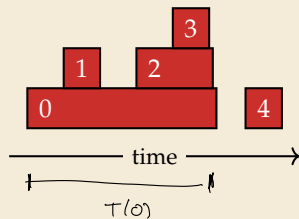
input graph G



DFS forest



stack over time

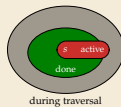
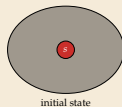


↪ Each vertex v spends *time interval* $T(v)$ as **active** vertex

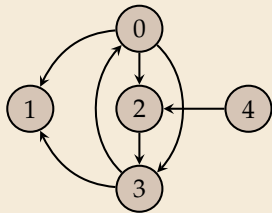
Properties of DFS

► Recall DFS Invariant (2):

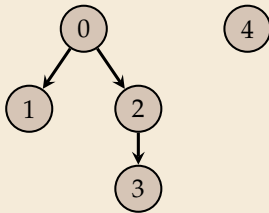
The **active** vertices form a **single path** from s



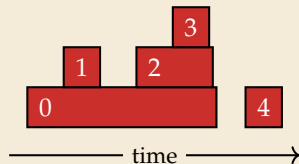
input graph G



DFS forest



stack over time



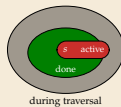
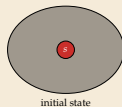
$\rightsquigarrow$ Each vertex v spends *time interval* $T(v)$ as **active** vertex

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 $T(v) \supseteq T(w)$

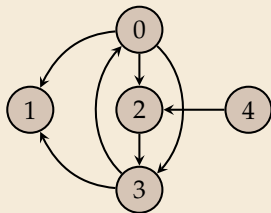
Properties of DFS

► Recall DFS Invariant (2):

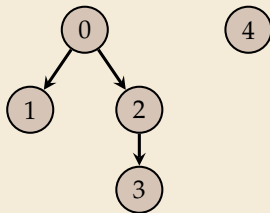
The **active** vertices form a **single path** from s



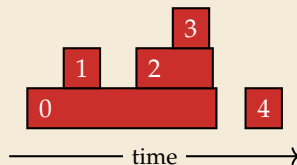
input graph G



DFS forest



stack over time



↪ 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**: $T(v) \supseteq T(w)$ **iff** v is ancestor of w in DFS tree

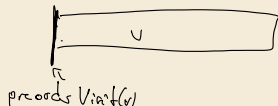
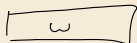
‘ $\Rightarrow$ ’ during $T(v)$, all discovered vertices become descendants of v

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



Properties of DFS – Unseen-Path Theorem

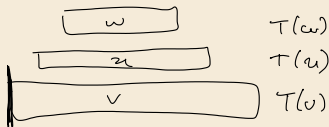
- **Unseen-Path Theorem:** In a DFS forest of a (di)graph G , w is a descendant of v iff at the time of $\text{preorderVisit}(v)$, there is a path from v to w using only *unseen* vertices.



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.

' $\Rightarrow$ ' If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem.
Hence the path from v to w in the DFS tree consists (at time of $\text{preorderVisit}(v)$) of solely *unseen* vertices.



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.
 - ' $\Rightarrow$ ' If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem. Hence the path from v to w in the DFS tree consists (at time of `preorderVisit( $v$ )`) of solely *unseen* vertices.
 - ' $\Leftarrow$ ' Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v .

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.
- ‘ $\Rightarrow$ ’ If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem. Hence the path from v to w in the DFS tree consists (at time of $\text{preorderVisit}(v)$) of solely *unseen* vertices.
- ‘ $\Leftarrow$ ’ Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v . W.l.o.g. let w be a first such vertex, i. e., $p[0], \dots, p[\ell - 1] = u$ are descendants of v .



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.
- ‘ $\Rightarrow$ ’ If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem. Hence the path from v to w in the DFS tree consists (at time of `preorderVisit( $v$ )`) of solely *unseen* vertices.
- ‘ $\Leftarrow$ ’ Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v . W.l.o.g. let w be a first such vertex, i. e., $p[0], \dots, p[\ell - 1] = u$ are descendants of v . So $T(u) \subset T(v)$. (1)

Properties of DFS – Unseen-Path Theorem

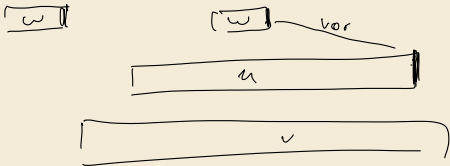
- **Unseen-Path Theorem:** In a DFS forest of a (di)graph G , w is a descendant of v iff at the time of $\text{preorderVisit}(v)$, there is a path from v to w using only *unseen* vertices.

‘ $\Rightarrow$ ’ If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem.

Hence the path from v to w in the DFS tree consists (at time of $\text{preorderVisit}(v)$) of solely *unseen* vertices.

‘ $\Leftarrow$ ’ Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v . W.l.o.g. let w be a first such vertex, i. e., $p[0], \dots, p[\ell - 1] = u$ are descendants of v . So $T(u) \subset T(v)$. (1)

Since w was *unseen* at the time of $\text{preorderVisit}(v)$, $\min T(v) \leq \min T(w)$. (2)



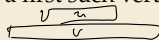
Properties of DFS – Unseen-Path Theorem

- **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.

‘ $\Rightarrow$ ’ If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem.

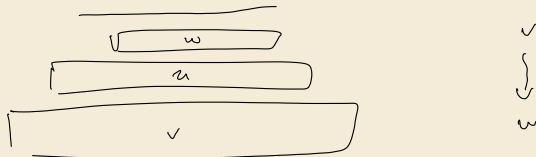
Hence the path from v to w in the DFS tree consists (at time of $\text{preorderVisit}(v)$) of solely *unseen* vertices.

‘ $\Leftarrow$ ’ Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v . W.l.o.g. let w be a first such vertex, i. e., $p[0], \dots, p[\ell - 1] = u$ are descendants of v . So $T(u) \subset T(v)$. (1)



Since w was *unseen* at the time of $\text{preorderVisit}(v)$, $\min T(v) \leq \min T(w)$. (2)

Upon processing u , we will discover edge uw , so whether or not w is already *done* at this point, w will be marked *done* before u . Hence $\max T(w) \leq \max T(u)$. (3)



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.

‘ $\Rightarrow$ ’ If w is a descendant of v , $T(w) \subseteq T(v)$ by the Parenthesis Theorem.

Hence the path from v to w in the DFS tree consists (at time of `preorderVisit( $v$ )`) of solely *unseen* vertices.

‘ $\Leftarrow$ ’ Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v . W.l.o.g. let w be a first such vertex, i. e., $p[0], \dots, p[\ell - 1] = u$ are descendants of v . So $T(u) \subset T(v)$. (1)

Since w was *unseen* at the time of `preorderVisit( $v$ )`, $\min T(v) \leq \min T(w)$. (2)

Upon processing u , we will discover edge uw , so whether or not w is already *done* at this point, w will be marked *done* before u . Hence $\max T(w) \leq \max T(u)$. (3)

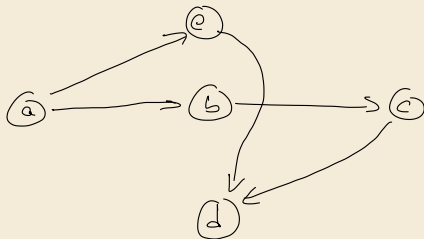
By (1), (2), (3) and laminarity, $T(w) \subset T(u) \subset T(v)$ and w is a descendant of v ⚡.

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



a e b c d
a b c e d
a b e c d

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
$$\forall (u, v) \in E : R[u] < R[v] \quad \text{think: } R[v] = \text{time slot for task } v$$
- ▶ **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
$$\forall (u, v) \in E : R[u] < R[v] \quad \text{think: } R[v] = \text{time slot for task } v$$
- ▶ **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!

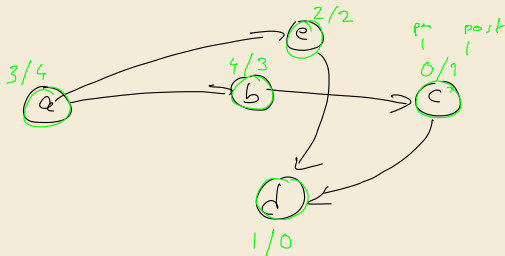
DFS Postorder & Topological Sort

- **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.

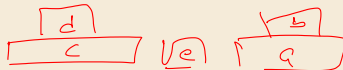
DFS Postorder & Topological Sort

- **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 $v \in \underline{\text{done}}$ and $(v, w) \in E$ then $w \in \underline{\text{done}}$ and $R[v] < R[w]$.
 - initially true ($\text{done} = \emptyset$) $\overline{\text{time stamp}}$
 - upon `postorderVisit(v)`, all outgoing edges vw lead to $w \in \underline{\text{done}}$ (Parenthesis Theorem)

v	a	b	c	d	e
R	0	1	3	4	2

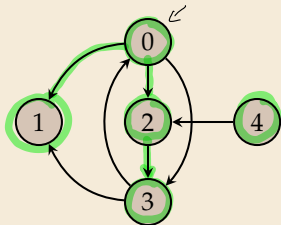


$d \quad c \quad e \quad b \quad a$
 $\longleftrightarrow$
 $a \quad b \quad e \quad c \quad d$



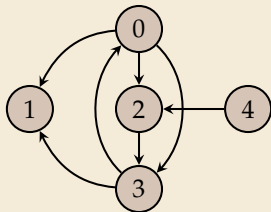
DFS Edge Types

input digraph G

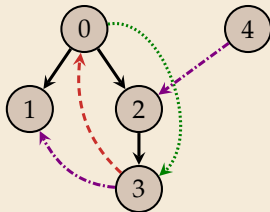


DFS Edge Types

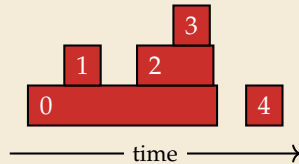
input digraph G



DFS forest

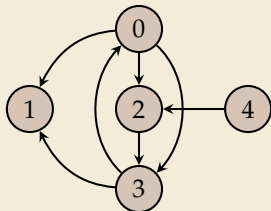


stack over time

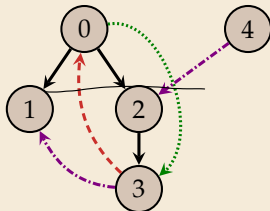


DFS Edge Types

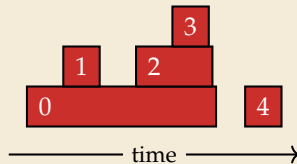
input digraph G



DFS forest



stack over time



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

1. **tree edge:** $\longrightarrow w \in \text{unseen} \rightsquigarrow vw$ 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:

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

$(3, 0)$

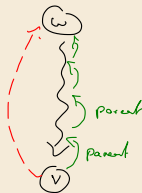
$(0, 3)$

$(3, 1), (4, 2)$

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 $v \rightarrow w$.

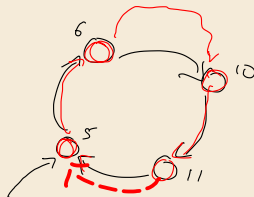


Cycle Detection

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

unseen path flow.



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
```

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  $cycle \neq \text{NULL}$   
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  $R$ 
```

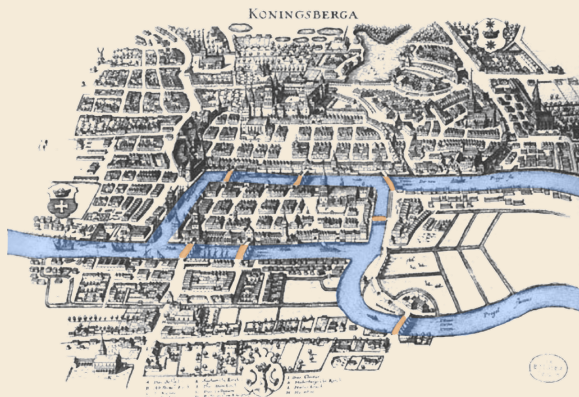
```
1 procedure topologicalSort( $P$ ):  
2   ( $P[0..n], cycle$ ) := dfsPostorder( $G$ )  
3   if  $c \neq \text{NULL}$   
4     return NOT_A_DAG  
5    $S[0..n] := \text{NONE}$   
6   for  $v := 0, \dots, n - 1$   
7      $S[n - 1 - P[v]] := v$   
8   return  $S$ 
```

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

9.7 Advanced Uses of DFS II

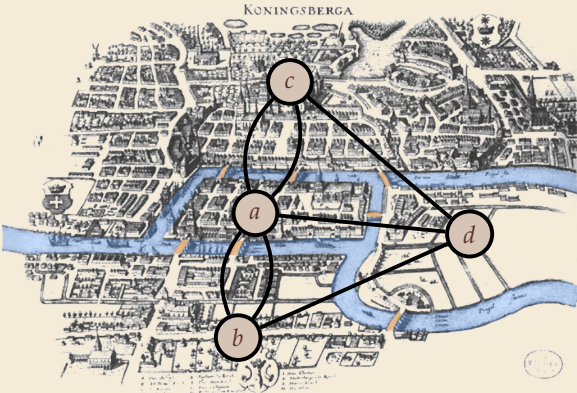
Euler Cycles

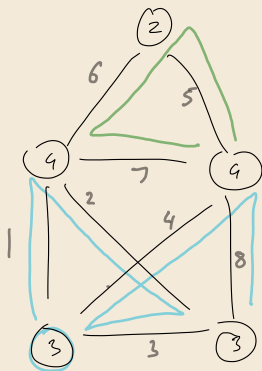
Euler Walk: Walk using every edge in $G = (V, E)$ exactly once.



Euler Cycles

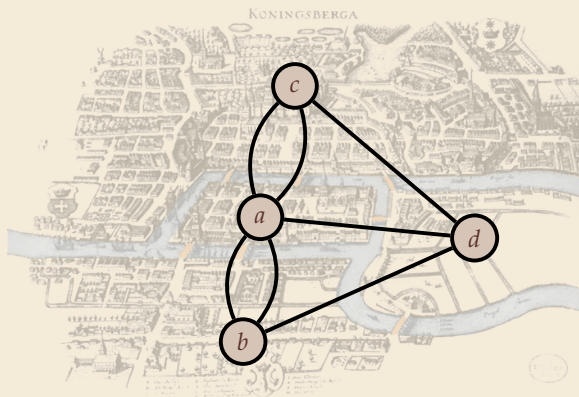
Euler Walk: Walk using every edge in $G = (V, E)$ exactly once.





Euler Cycles

Euler Walk: Walk using every edge in $G = (V, E)$ exactly once.



Euler's Theorem:

Euler walk exists iff G connected and 0 or 2 vertices have odd degree.

' $\Rightarrow$ ' trivial (need to enter and exit intermediate vertices equally often)

' $\Leftarrow$ ' Following algorithm *constructs* Euler walk under this assumption



Euler Cycles – Hierholzer's Algorithm

► use an *edge-centric DFS*

- We mark *edges* (not vertices)

↪ stack = **edge-simple walk**

- We remember iterator i globally per v to resume traversal

```
1 procedure eulerWalk(G):
2   // Assume  $G = (V, E)$  is connected (multi)graph
3    $V_{\text{odd}} := \{v \in V : d(v) \text{ odd}\}$ 
4   if  $|V_{\text{odd}}| \notin \{0, 2\}$  return NOT_EULERIAN
5   if  $V_{\text{odd}} = \{x, y\}$  then  $s := x$  else  $s := 0$ 
6    $euler[0..m] := \text{NONE}$ ;  $j := m - 1$ 
7    $visited[0..n, 0..n] := \text{false}$  // mark edges as visited
8   for  $v := 0, \dots, n - 1$ 
9     // globally remember next unexplored edge
10     $nextEdge[v] := G.adj[v].iterator()$ 
11  edgeDFS(s)
12  return euler
```

```
1 procedure edgeDFS(s):
2   frontier := new Stack;
3   frontier.push(s)
4   while  $\neg \text{frontier.isEmpty}()$ 
5      $v := \text{frontier.top}()$ ;  $i := nextEdge[v]$ 
6     if  $\neg i.hasNext()$  //  $v$  has no unused edge
7       frontier.pop()
8       if  $\neg \text{frontier.isEmpty}()$ 
9         // assign edge leading here largest free index
10         $euler[j] := (\text{frontier.top}(), v)$ ;  $j := j - 1$ 
11      end if
12    else
13       $w := i.next()$ 
14      if  $\neg visited[v, w]$ 
15         $visited[v, w] := \text{true}$ 
16         $visited[w, v] := \text{true}$ 
17        frontier.push( $w$ )
18      end if
19    end if
20  end while
```

Euler Cycles – Further Results

- ▶ Edge-centric DFS takes $O(n + m)$ time
- ▶ also $\Theta(m)$ space in the worst case for the stack
 - ▶ can be improved to $\Theta(n)$ with variant of the algorithm



Ismaili Alaoui, Plump, Wild: *Space-Efficient Hierholzer: Eulerian Cycles in $O(m)$ Time and $O(n)$ Space*, SOSA 2026

EXCERPT

Euler Cycles – Further Results

- ▶ Edge-centric DFS takes $O(n + m)$ time
- ▶ also $\Theta(m)$ space in the worst case for the stack
 - ▶ can be improved to $\Theta(n)$ with variant of variant of the algorithm



Ismaili Alaoui, Plump, Wild: *Space-Efficient Hierholzer: Eulerian Cycles in $O(m)$ Time and $O(n)$ Space*, SOSA 2026

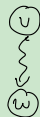
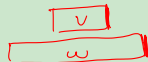
- ▶ Same approach can be made to work for *directed* graphs
- ▶ Euler paths can be used to assemble genomes from fragments
→ *Algorithms of Bioinformatics*

Clicker Question

Mark all correct statements about a `dfsTraversal` (Slide 24) of a **DAG** G :



- ☐ **A** Listing vertices in the order they are marked *done* is a topological sorting of G .
- ☐ **B** Listing vertices in the reverse order they are marked *done* is a topological sorting of G .
- ☐ **C** If v is marked *done* before vertex w , there is a path $v \rightsquigarrow w$.
- ☐ **D** If v is marked *done* before vertex w , there is a path $w \rightsquigarrow v$.
- ☐ **E** If v is marked *done* before vertex w , there cannot be a path $v \rightsquigarrow w$.
- ☐ **F** If v is marked *done* before vertex w , there cannot be a path $w \rightsquigarrow v$.



→ sli.do/cs566

Clicker Question

Mark all correct statements about a `dfsTraversal` (Slide 24) of a **DAG** G :



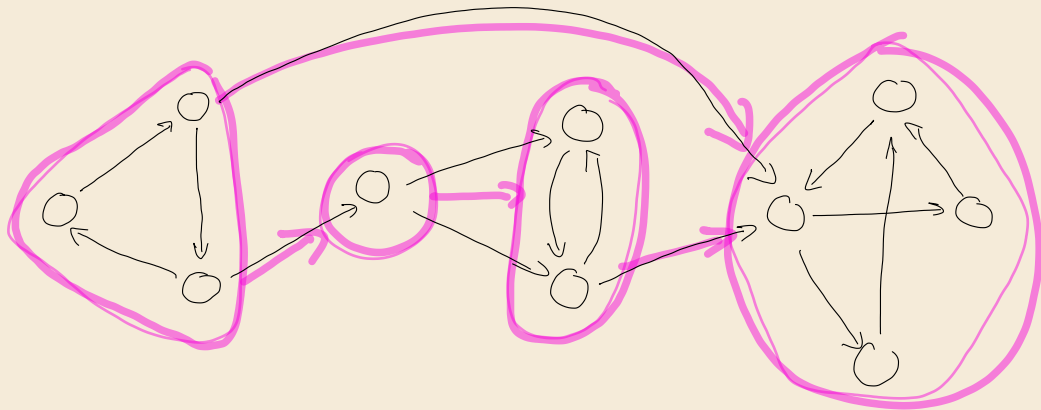
- ☐ **A** ~~Listing vertices in the order they are marked *done* is a topological sorting of G .~~
- ☐ **B** Listing vertices in the reverse order they are marked *done* is a topological sorting of G . ✓
- ☐ **C** ~~If v is marked *done* before vertex w , there is a path $v \rightsquigarrow w$.~~
- ☐ **D** ~~If v is marked *done* before vertex w , there is a path $w \rightsquigarrow v$.~~
- ☐ **E** If v is marked *done* before vertex w , there cannot be a path $v \rightsquigarrow w$. ✓
- ☐ **F** ~~If v is marked *done* before vertex w , there cannot be a path $w \rightsquigarrow v$.~~



→ sli.do/cs566

Strong Components

- **Given:** digraph $G = (V, E)$
- **Goal:** component ids $SCC[0..n)$, s.t. $SCC[v] = SCC[u]$ iff $\exists$ directed path from v to u



Strong Components

- ▶ **Given:** digraph $G = (V, E)$
 - ▶ **Goal:** component ids $SCC[0..n)$, s.t. $SCC[v] = SCC[u]$ iff $\exists$ directed path from v to u
strongly connected component
 - ▶ **Component DAG** G^{SCC} : contract SCCs into single vertices
 $V(G^{SCC}) = \{C_1, \dots, C_k\}$ with $C_1 \dot{\cup} \dots \dot{\cup} C_k = V$;
name by smallest vertex s.t. $i \leq j$ iff $\min C_i \leq \min C_j$
 - ▶ can't have cycles (⚡ maximality of SCC)
- $\rightsquigarrow$ component DAG has a topological order $R^{SCC}[1..k]$

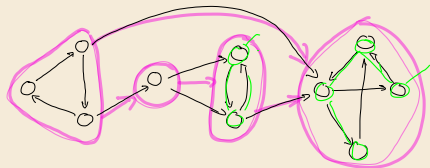
Strong Components

- ▶ **Given:** digraph $G = (V, E)$
- ▶ **Goal:** component ids $SCC[0..n)$, s.t. $SCC[v] = SCC[u]$ iff $\exists$ directed path from v to u
strongly connected component
- ▶ **Component DAG** G^{SCC} : contract SCCs into single vertices
 $V(G^{SCC}) = \{C_1, \dots, C_k\}$ with $C_1 \dot{\cup} \dots \dot{\cup} C_k = V$;
name by smallest vertex s.t. $i \leq j$ iff $\min C_i \leq \min C_j$
 - ▶ can't have cycles (⚡ maximality of SCC)
 - $\rightsquigarrow$ component DAG has a topological order $R^{SCC}[1..k]$



If we call dfs on any v in the **last** SCC C , it will discover all vertices in C , and only those!
(any edges between components lead *into* C by topological order)

And we can iterate this backwards through any topological order to get all SCCs!



Strong Components

- ▶ **Given:** digraph $G = (V, E)$
 - ▶ **Goal:** component ids $SCC[0..n)$, s.t. $SCC[v] = SCC[u]$ iff $\exists$ directed path from v to u
strongly connected component
 - ▶ **Component DAG G^{SCC} :** contract SCCs into single vertices
 $V(G^{SCC}) = \{C_1, \dots, C_k\}$ with $C_1 \dot{\cup} \dots \dot{\cup} C_k = V$;
name by smallest vertex s.t. $i \leq j$ iff $\min C_i \leq \min C_j$
 - ▶ can't have cycles (⚡ maximality of SCC)
- $\leadsto$ component DAG has a topological order $R^{SCC}[1..k]$



If we call dfs on any v in the **last** SCC C , it will discover all vertices in C , and only those!
(any edges between components lead *into* C by topological order)

And we can iterate this backwards through any topological order to get all SCCs!



Can we efficiently find the topological order of G^{SCC} ?
Without knowing the components to start with??

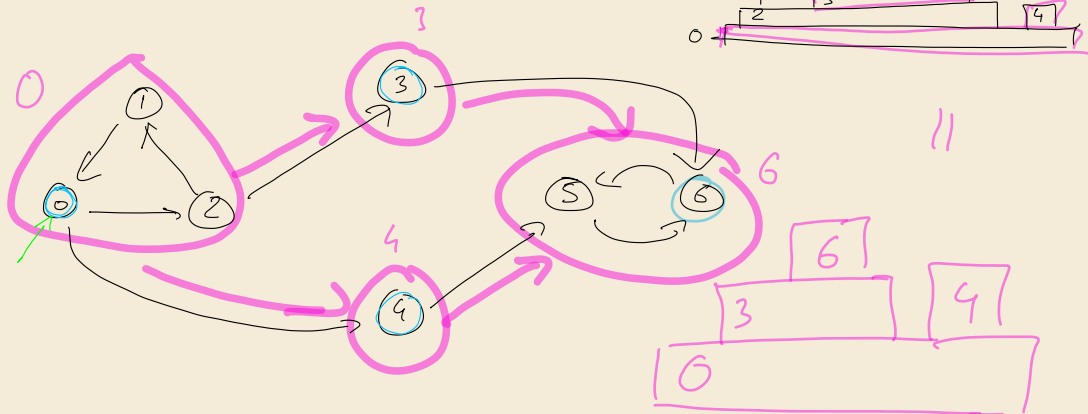
Amazingly, yes.

Component Graph DFS

- Suppose we run `dfsTraversal` on G .

↪ We can extend time intervals to SCCs: $T(C_i) := \bigcup_{v \in C_i} T(v)$

↪ $T(C_i) = T(v_i)$ for $v_i \in C_i$ the first vertex to be explored in a DFS on G
(by Unseen Path & Parenthesis Thms)



Component Graph DFS

► Suppose we run `dfsTraversal` on G .

↪ We can extend time intervals to SCCs: $T(C_i) := \bigcup_{v \in C_i} T(v)$

↪ $T(C_i) = T(v_i)$ for $v_i \in C_i$ the first vertex to be explored in a DFS on G
(by Unseen Path & Parenthesis Thms)

↪ DFS on G produces same $T(C_i)$ (up to time scaling) as DFS on G^{SCC} !

↪ reverse DFS postorder on G gives same relative order to $v_1, \dots, v_k$ as
reverse DFS postorder on G^{SCC} gives as relative order to $C_1, \dots, C_k$

Component Graph DFS

- ▶ Suppose we run `dfsTraversal` on G .

↪ We can extend time intervals to SCCs: $T(C_i) := \bigcup_{v \in C_i} T(v)$

↪ $T(C_i) = T(v_i)$ for $v_i \in C_i$ the first vertex to be explored in a DFS on G
(by Unseen Path & Parenthesis Thms)

↪ DFS on G produces same $T(C_i)$ (up to time scaling) as DFS on G^{SCC} !

↪ reverse DFS postorder on G gives same relative order to $v_1, \dots, v_k$ as
reverse DFS postorder on G^{SCC} gives as relative order to $C_1, \dots, C_k$



We need **reverse** topological order on G^{SCC} , e. g., *reversed* reverse DFS postorder

- ▶ If we had the actual reverse DFS postorder on G^{SCC} , could just reverse again!
- ▶ But we only have reverse DFS postorder $S[0..n)$ on G !
- ⚡ Reversing here would change v_i , i. e., which vertices of an SCC we see first

Kosaraju-Sharir's Algorithm

- **Recall:** Want $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$

Kosaraju-Sharir's Algorithm

- ▶ **Recall:** Want $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$
- ▶ **Transpose/Reverse Graph of $G = (V, E)$:** $G^T = (V, E^T)$ where $E^T = \{wv : vw \in E\}$
Note: A adj matrix of $G \rightsquigarrow A^T$ adj matrix of G^T
- ▶ For any DAG, we obtain a reverse topological order from reversing all edges:
 $\text{topologicalSort}(G^T)$  If we reverse *iteration order* in `dfsTraversal`, we get $\text{reverse}(\text{topologicalSort}(G)) = \text{topologicalSort}(G^T)$

Kosaraju-Sharir's Algorithm

- ▶ **Recall:** Want $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$
- ▶ **Transpose/Reverse Graph of $G = (V, E)$:** $G^T = (V, E^T)$ where $E^T = \{wv : vw \in E\}$
Note: A adj matrix of $G \rightsquigarrow A^T$ adj matrix of G^T
- ▶ For any DAG, we obtain a reverse topological order from reversing all edges:
 $\text{topologicalSort}(G^T)$  If we reverse iteration order in `dfsTraversal`, we get $\text{reverse}(\text{topologicalSort}(G)) = \text{topologicalSort}(G^T)$
- ▶ **Observation:** $(G^T)^{\text{SCC}} = (G^{\text{SCC}})^T$
 - ▶ strong components not affected by edge reversals
- ▶ **Want:** $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$ (any ranking works, need not be reverse DFS postorder)

Kosaraju-Sharir's Algorithm

- ▶ **Recall:** Want $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$
- ▶ **Transpose/Reverse Graph of $G = (V, E)$:** $G^T = (V, E^T)$ where $E^T = \{wv : vw \in E\}$
Note: A adj matrix of $G \rightsquigarrow A^T$ adj matrix of G^T
- ▶ For any DAG, we obtain a reverse topological order from reversing all edges:
 $\text{topologicalSort}(G^T)$  If we reverse iteration order in dfsTraversal , we get $\text{reverse}(\text{topologicalSort}(G)) = \text{topologicalSort}(G^T)$
- ▶ **Observation:** $(G^T)^{\text{SCC}} = (G^{\text{SCC}})^T$
 - ▶ strong components not affected by edge reversals
- ▶ **Want:** $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$ (any ranking works, need not be reverse DFS postorder)
- $\rightsquigarrow$ Get it from: $\text{topologicalRanking}((G^{\text{SCC}})^T) = \text{topologicalRanking}((G^T)^{\text{SCC}})$

Kosaraju-Sharir's Algorithm

- ▶ **Recall:** Want $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$
- ▶ **Transpose/Reverse Graph of $G = (V, E)$:** $G^T = (V, E^T)$ where $E^T = \{wv : vw \in E\}$
Note: A adj matrix of $G \rightsquigarrow A^T$ adj matrix of G^T
- ▶ For any DAG, we obtain a reverse topological order from reversing all edges:
 $\text{topologicalSort}(G^T)$  If we reverse iteration order in dfsTraversal , we get $\text{reverse}(\text{topologicalSort}(G)) = \text{topologicalSort}(G^T)$
- ▶ **Observation:** $(G^T)^{\text{SCC}} = (G^{\text{SCC}})^T$
 - ▶ strong components not affected by edge reversals
- ▶ **Want:** $\text{reverse}(\text{topologicalRanking}(G^{\text{SCC}}))$ (any ranking works, need not be reverse DFS postorder)
- $\rightsquigarrow$ Get it from: $\text{topologicalRanking}((G^{\text{SCC}})^T) = \text{topologicalRanking}((G^T)^{\text{SCC}})$
- $\rightsquigarrow$ Get that as induced ranking on $v_1, \dots, v_k$ from $\text{reverse dfsPostorder}(G^T)$

Kosaraju-Sharir's Algorithm – Code

```
1 procedure strongComponents(G):
2   // directed graph  $G = (V, E)$  with  $V = [0..n)$ 
3    $G^T = (V, \{wv : vw \in E\})$ 
4    $P[0..n) := \text{dfsPostorder}(G^T)$  // postorder numbers
5   for  $v \in V$  do  $S[P[v]] := v$  end for // postorder sequence
6   // Rest like connectedComponents (with permuted vertices)
7    $C[0..n) := \text{unseen}$ 
8    $\text{SCC}[0..n) := \text{NONE}$ 
9    $id := 0$ 
10  for  $j := n - 1, \dots, 0$  // reverse postorder seq
11     $v := S[j]$ 
12    if  $C[v] == \text{unseen}$ 
13       $\text{dfs}(G, v)$ 
14       $id := id + 1$ 
15  return  $\text{SCC}$ 
16
17 procedure preorderVisit( $v$ ):
18    $\text{SCC}[v] := id$ 
```

Kosaraju-Sharir's Algorithm – Code

```
1 procedure strongComponents(G):
2   // directed graph  $G = (V, E)$  with  $V = [0..n)$ 
3    $G^T = (V, \{wv : vw \in E\})$ 
4    $P[0..n) := \text{dfsPostorder}(G^T)$  // postorder numbers
5   for  $v \in V$  do  $S[P[v]] := v$  end for // postorder sequence
6   // Rest like connectedComponents (with permuted vertices)
7    $C[0..n) := \text{unseen}$ 
8    $\text{SCC}[0..n) := \text{NONE}$ 
9    $id := 0$ 
10  for  $j := n - 1, \dots, 0$  // reverse postorder seq
11     $v := S[j]$ 
12    if  $C[v] == \text{unseen}$ 
13       $\text{dfs}(G, v)$ 
14       $id := id + 1$ 
15  return  $\text{SCC}$ 
16
17 procedure preorderVisit( $v$ ):
18    $\text{SCC}[v] := id$ 
```

- correctness follows from our discussion

Kosaraju-Sharir's Algorithm – Code

```
1 procedure strongComponents( $G$ ):
2   // directed graph  $G = (V, E)$  with  $V = [0..n)$ 
3    $G^T = (V, \{wv : vw \in E\})$ 
4    $P[0..n) := \text{dfsPostorder}(G^T)$  // postorder numbers
5   for  $v \in V$  do  $S[P[v]] := v$  end for // postorder sequence
6   // Rest like connectedComponents (with permuted vertices)
7    $C[0..n) := \text{unseen}$ 
8    $\text{SCC}[0..n) := \text{NONE}$ 
9    $id := 0$ 
10  for  $j := n - 1, \dots, 0$  // reverse postorder seq
11     $v := S[j]$ 
12    if  $C[v] == \text{unseen}$ 
13       $\text{dfs}(G, v)$ 
14       $id := id + 1$ 
15  return  $\text{SCC}$ 
16
17 procedure preorderVisit( $v$ ):
18    $\text{SCC}[v] := id$ 
```

- ▶ correctness follows from our discussion
- ▶ ordering of SCCs follows reverse topological sort of G^{SCC}
 - ▶ some implementations reverse G for 2nd DFS, not 1st
 - $\rightsquigarrow$ output in (forward) topological order
 - ▶ but derivation more natural this way?

Kosaraju-Sharir's Algorithm – Code

```
1 procedure strongComponents(G):
2   // directed graph  $G = (V, E)$  with  $V = [0..n)$ 
3    $G^T = (V, \{wv : vw \in E\})$ 
4    $P[0..n) := \text{dfsPostorder}(G^T)$  // postorder numbers
5   for  $v \in V$  do  $S[P[v]] := v$  end for // postorder sequence
6   // Rest like connectedComponents (with permuted vertices)
7    $C[0..n) := \text{unseen}$ 
8    $\text{SCC}[0..n) := \text{NONE}$ 
9    $id := 0$ 
10  for  $j := n - 1, \dots, 0$  // reverse postorder seq
11     $v := S[j]$ 
12    if  $C[v] == \text{unseen}$ 
13      dfs( $G, v$ )
14       $id := id + 1$ 
15  return SCC
16
17 procedure preorderVisit( $v$ ):
18    $\text{SCC}[v] := id$ 
```

- ▶ correctness follows from our discussion
- ▶ ordering of SCCs follows reverse topological sort of G^{SCC}
 - ▶ some implementations reverse G for 2nd DFS, not 1st
 - $\rightsquigarrow$ output in (forward) topological order
 - ▶ but derivation more natural this way?
- ▶ as all our traversals:
 - $\Theta(n + m)$ time,
 - $\Theta(n)$ extra space

9.8 Network flows

Clicker Question



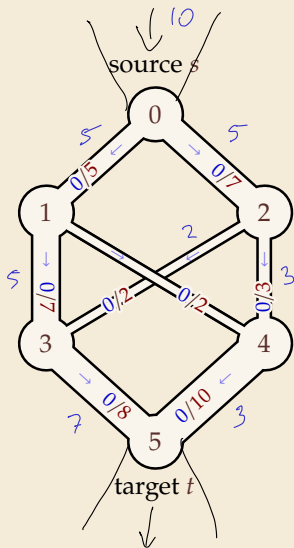
Prior knowledge from linear optimization; check all apply.

- ☐ **A** I've seen LPs in lectures before.
- ☐ **B** I could model an application problem as (I)LP.
- ☐ **C** I know algorithms for solving LPs.
- ☐ **D** I know what weak and strong duality in LPs are.
- ☐ **E** I could dualize an LP given to me.
- ☐ **F** I know about the complexity of LPs and ILPs.
- ☐ **G** LPs for me only mean music on vinyl.



→ *sli.do/cs566*

Networks and Flows – Informal

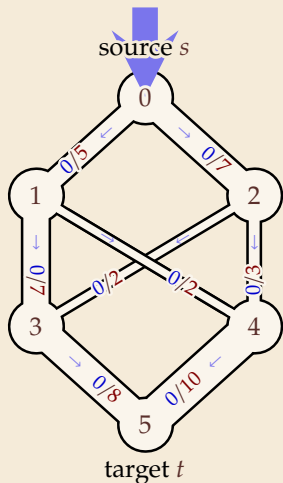


Informally, imagine a network of water pipes.

- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

Networks and Flows – Informal

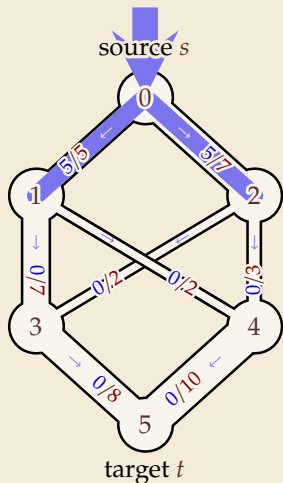


Informally, imagine a network of water pipes.

- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

Networks and Flows – Informal

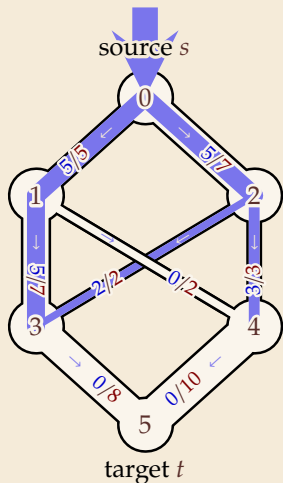


Informally, imagine a network of water pipes.

- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

Networks and Flows – Informal

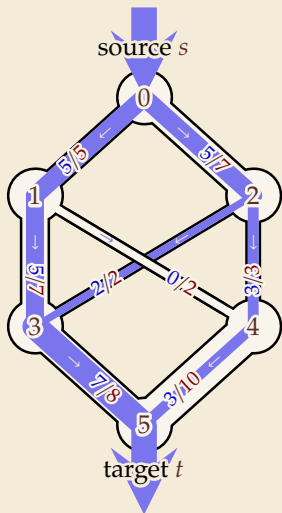


Informally, imagine a network of water pipes.

- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

Networks and Flows – Informal

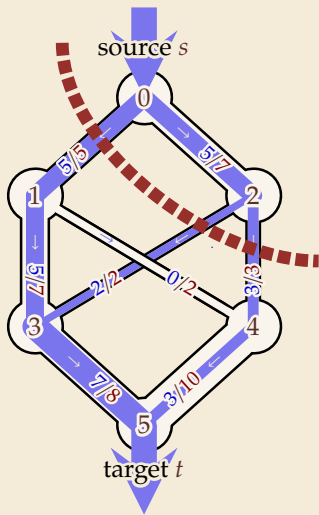


Informally, imagine a network of water pipes.

- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

Networks and Flows – Informal



Informally, imagine a network of water pipes.

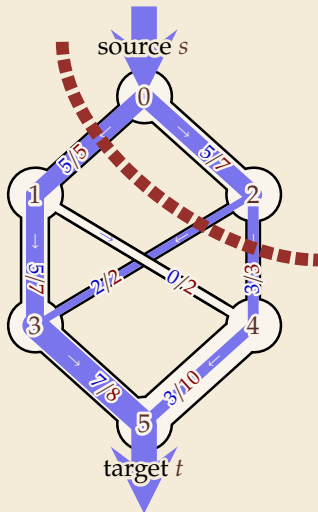
- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

In this example:

- ▶ not more than $5 + 2 + 3 = 10$ units of flow out of $\{0, 2\}$ possible
- ~> not more than 10 units out of s possible

Networks and Flows – Informal



Informally, imagine a network of water pipes.

- ▶ Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- ▶ There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- ▶ At all other junctions, inflow = outflow (no leakage)

~> How much water can flow through the network?

In this example:

- ▶ not more than $5 + 2 + 3 = 10$ units of flow out of $\{0, 2\}$ possible
- ~> not more than 10 units out of s possible
- ~> shown flow is maximal

Remainder of this unit: general version of above (+ efficient algorithms)

Networks and Flows – Definitions

► *s-t-(flow) network:*

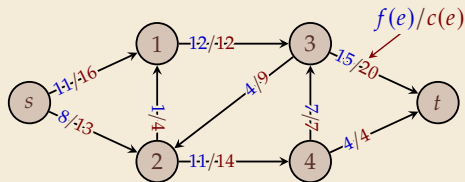
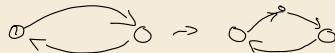
► **simple, directed, connected** graph $G = (V, E)$,

► *edge capacities* $c : E \rightarrow \mathbb{R}_{\geq 0}^{\neq 0}$

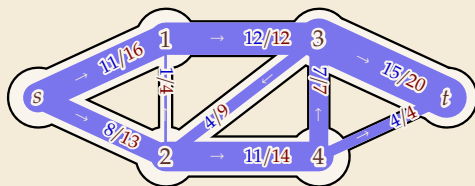
► distinguished vertices: *source* $s \in V$, target/*sink* $t \in V$

for notational convenience only

no antiparallel edges ($vw \in E \rightsquigarrow wv \notin E$)



=



Networks and Flows – Definitions

► *s-t-(flow) network*:

► **simple, directed, connected** graph $G = (V, E)$,

for notational convenience only

no antiparallel edges ($vw \in E \rightsquigarrow wv \notin E$)

► *edge capacities* $c : E \rightarrow \mathbb{R}_{\geq 0}$

► distinguished vertices: *source* $s \in V$, *target/sink* $t \in V$

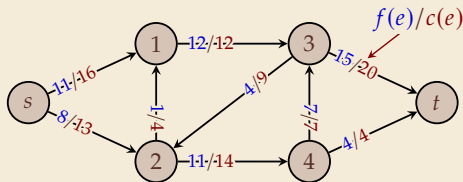
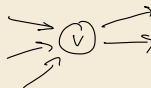
► (*network*) *flow* (in G): $f : E \rightarrow \mathbb{R}_{\geq 0}$

► flow f is *feasible* if it satisfies

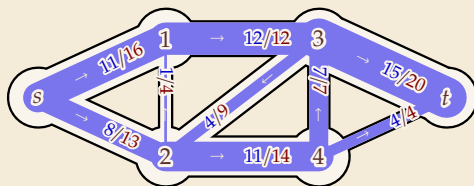
notational convenience: set $f(vw) = c(vw) = 0$ for $vw \notin E$

► *capacity constraints*: $\forall v, w \in V : 0 \leq f(vw) \leq c(vw)$

► *flow conservation*: $\forall v \in V \setminus \{s, t\} : \sum_{w \in V} f(w, v) = \sum_{w \in V} f(v, w)$



=



Networks and Flows – Definitions

► *s-t-(flow) network:*

► **simple, directed, connected** graph $G = (V, E)$,

for notational convenience only

no antiparallel edges ($vw \in E \rightsquigarrow wv \notin E$)

► *edge capacities* $c : E \rightarrow \mathbb{R}_{\geq 0}$

► distinguished vertices: *source* $s \in V$, *target/sink* $t \in V$

► (*network*) *flow* (in G): $f : E \rightarrow \mathbb{R}_{\geq 0}$

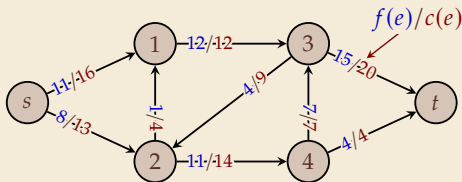
► flow f is *feasible* if it satisfies

notational convenience: set $f(vw) = c(vw) = 0$ for $vw \notin E$

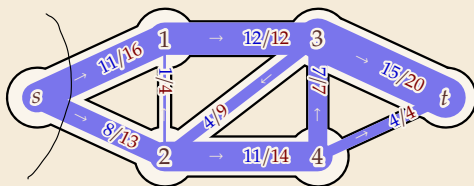
► *capacity constraints*: $\forall v, w \in V : 0 \leq f(vw) \leq c(vw)$

► *flow conservation*: $\forall v \in V \setminus \{s, t\} : \sum_{w \in V} f(w, v) = \sum_{w \in V} f(v, w)$

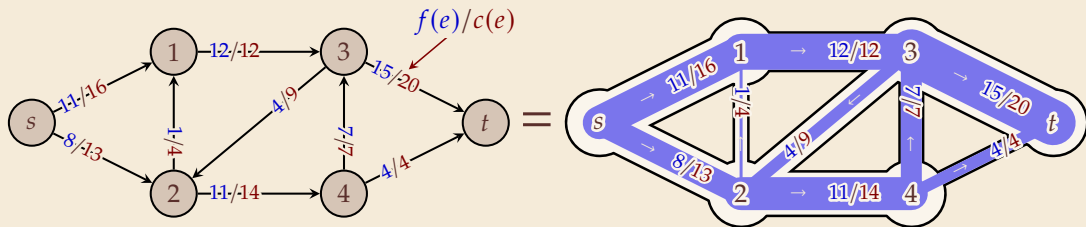
► *value* $|f|$ of flow f : $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$



=



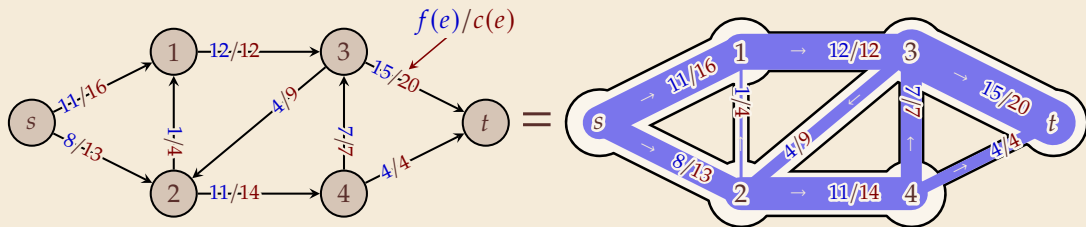
Max-Flow Problem



► Maximum-Flow Problem:

- **Given:** s - t -flow network
- **Goal:** Find feasible flow f^* with maximum $|f^*|$ among all feasible flows

Max-Flow Problem



► Maximum-Flow Problem:

- **Given:** s - t -flow network
- **Goal:** Find feasible flow f^* with maximum $|f^*|$ among all feasible flows

► $\mathbb{N}$ vs $\mathbb{R}$

- We focus on integral capacities here as we will see can restrict ourselves to integral flows
- but: ideally want algorithms that work with arbitrary real numbers, too

Multiple Sources & Sinks, Antiparallel Edges

- ▶ Some of the restrictions can be generalized easily.
 - ▶ We forbid **loops** and **antiparallel** edges.
 - ▶ The presented algorithms actually work fine with both!
 - ▶ but proofs are cleaner to write without them
 - ▶ also: can always remove loops and (anti)parallel edges by adding a new vertex in the middle of the edge
- ~→ same maximum $|f|$

Multiple Sources & Sinks, Antiparallel Edges

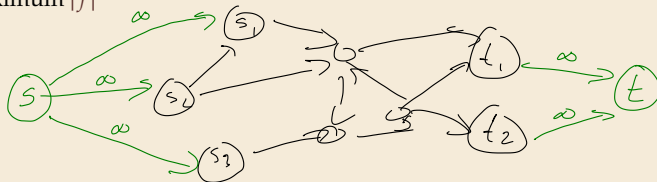
- ▶ Some of the restrictions can be generalized easily.
- ▶ We forbid **loops** and **antiparallel** edges.
 - ▶ The presented algorithms actually work fine with both!
 - ▶ but proofs are cleaner to write without them
 - ▶ also: can always remove loops and (anti)parallel edges by adding a new vertex in the middle of the edge

~> same maximum $|f|$

- ▶ We only allow a **single source** and a **single sink**

- ▶ can add a “**supersource**” and “**supersink**” with capacity- ∞ edges to all sources resp. sinks

~> same maximum $|f|$



Reductions

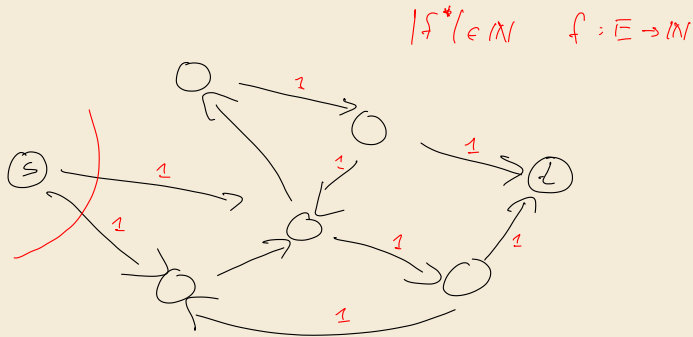
- ▶ Apart from directly modeling (data, traffic, etc.) flow, a key reason to study network flows are **reductions** of other problems

Reductions

- ▶ Apart from directly modeling (data, traffic, etc.) flow, a key reason to study network flows are **reductions** of other problems

1. Disjoint Paths

- ▶ **Given:** Unweighted (di)graph $G = (V, E)$, vertices $s, t \in V$
- ▶ **Goal:** How many edge-disjoint paths are there from s to t ?



Reductions

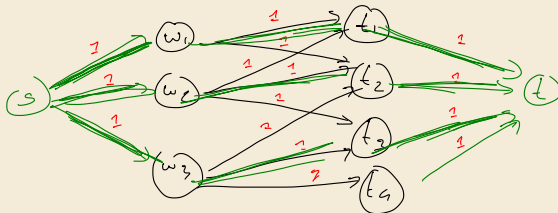
- ▶ Apart from directly modeling (data, traffic, etc.) flow, a key reason to study network flows are **reductions** of other problems

1. Disjoint Paths

- ▶ **Given:** Unweighted (di)graph $G = (V, E)$, vertices $s, t \in V$
- ▶ **Goal:** How many edge-disjoint paths are there from s to t ?

2. Assignment Problem, Maximum Bipartite Matching

- ▶ **Given:** workers $W = \{w_1, \dots, w_k\}$ tasks $T = \{t_1, \dots, t_\ell\}$, *qualified-for* relation $Q \subseteq W \times T$
- ▶ **Goal:** Assignment $a : W \rightarrow T \cup \{\perp\}$ of workers to tasks such that
 - ▶ workers are qualified: $\forall w \in W : a(w) \neq \perp \implies (w, a(w)) \in Q$
 - ▶ $|a(W)|$, the number of tasks assigned, is maximized



Reductions

- ▶ Apart from directly modeling (data, traffic, etc.) flow, a key reason to study network flows are **reductions** of other problems

1. Disjoint Paths

- ▶ **Given:** Unweighted (di)graph $G = (V, E)$, vertices $s, t \in V$
- ▶ **Goal:** How many edge-disjoint paths are there from s to t ?

2. Assignment Problem, Maximum Bipartite Matching

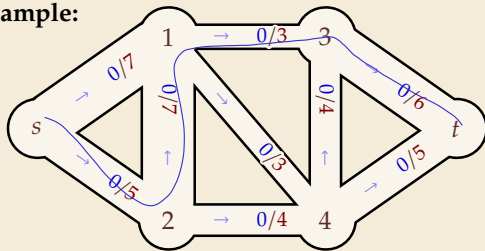
- ▶ **Given:** workers $W = \{w_1, \dots, w_k\}$ tasks $T = \{t_1, \dots, t_\ell\}$, *qualified-for* relation $Q \subseteq W \times T$
- ▶ **Goal:** Assignment $a : W \rightarrow T \cup \{\perp\}$ of workers to tasks such that
 - ▶ workers are qualified: $\forall w \in W : a(w) \neq \perp \implies (w, a(w)) \in Q$
 - ▶ $|a(W)|$, the number of tasks assigned, is maximized
- ▶ Both problems can be solved by (in both cases, 1. and 3. are very efficient)
 1. constructing a specific flow network from their input data
 2. computing a maximum flow in that network
 3. “reading off” a solution for the original problem from the max flow

9.9 The Ford-Fulkerson Method

Push Push Push!

- **Simple Idea:** Iteratively find a path from s to t that we can push more flow over.

Example:



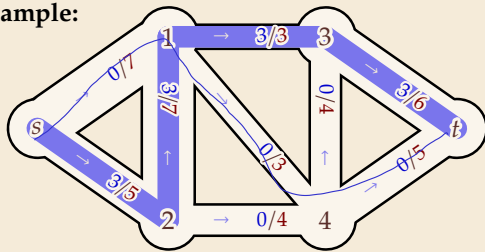
1. Push 3 units of flow over
 $s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t$

Problem: Cannot undo mistakes.
Here: shouldn't have put so
much flow on $(1, 2) \dots$

Push Push Push!

- **Simple Idea:** Iteratively find a path from s to t that we can push more flow over.

Example:



1. Push 3 units of flow over

$s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t$

2. Push 3 units of flow over

$s \rightarrow 1 \rightarrow 4 \rightarrow t$

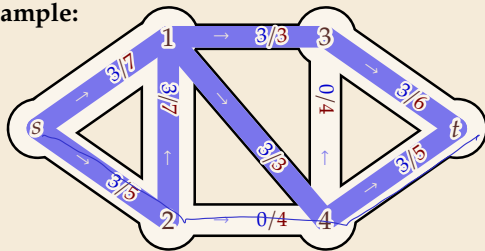
Problem: Cannot undo mistakes.

Here: shouldn't have put so much flow on $(1, 2) \dots$

Push Push Push!

- **Simple Idea:** Iteratively find a path from s to t that we can push more flow over.

Example:



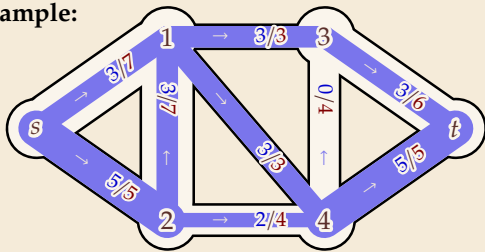
1. Push 3 units of flow over $s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t$
2. Push 3 units of flow over $s \rightarrow 1 \rightarrow 4 \rightarrow t$
3. Push 2 units of flow over $s \rightarrow 2 \rightarrow 4 \rightarrow t$

Problem: Cannot undo mistakes.
Here: shouldn't have put so much flow on $(1, 2) \dots$

Push Push Push!

- **Simple Idea:** Iteratively find a path from s to t that we can push more flow over.

Example:



1. Push 3 units of flow over
 $s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t$

2. Push 3 units of flow over
 $s \rightarrow 1 \rightarrow 4 \rightarrow t$

3. Push 2 units of flow over
 $s \rightarrow 2 \rightarrow 4 \rightarrow t$

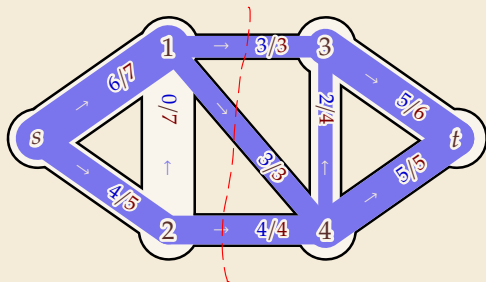
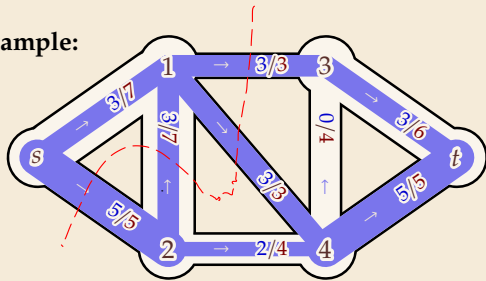
~> Every s - t path now has a saturated edge.

Problem: Cannot undo mistakes.
Here: shouldn't have put so
much flow on $(1, 2) \dots$

Push Push Push!?

- **Simple Idea:** Iteratively find a path from s to t that we can push more flow over.

Example:



1. Push 3 units of flow over
 $s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t$

2. Push 3 units of flow over
 $s \rightarrow 1 \rightarrow 4 \rightarrow t$

3. Push 2 units of flow over
 $s \rightarrow 2 \rightarrow 4 \rightarrow t$

↪ Every s - t path now has a saturated edge.

⚠ **But:** resulting flow is **not** optimal!

Problem: Cannot undo mistakes.
Here: shouldn't have put so much flow on $(1, 2) \dots$

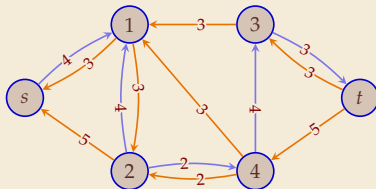
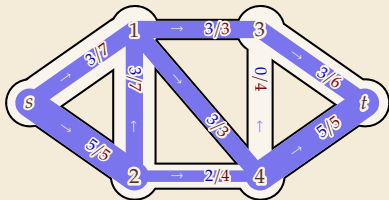
Residual Networks

- ▶ Goal: Allow undoing flow (without backtracking)

Residual Networks

- Goal: Allow undoing flow (without backtracking)
- *Residual network* G_f : given network $G = (V, E)$ and feasible flow f

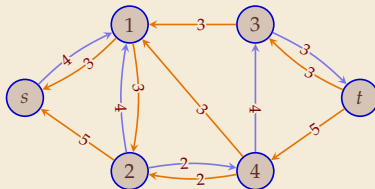
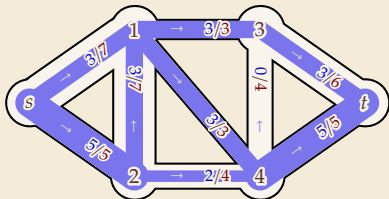
► $G_f = (V, E_f)$ with capacities $c_f(vw) = \begin{cases} c(vw) - f(vw) & vw \in E \quad // \text{add flow} \\ f(wv) & wv \in E \quad // \text{revert flow} \\ 0 & \text{else} \end{cases}$
 $E_f = \{vw : c_f(vw) > 0\}$



Residual Networks

- Goal: Allow undoing flow (without backtracking)
- *Residual network* G_f : given network $G = (V, E)$ and feasible flow f

- $G_f = (V, E_f)$ with capacities $c_f(vw) = \begin{cases} c(vw) - f(vw) & vw \in E \quad // \text{add flow} \\ f(wv) & wv \in E \quad // \text{revert flow} \\ 0 & \text{else} \end{cases}$
 $E_f = \{vw : c_f(vw) > 0\}$



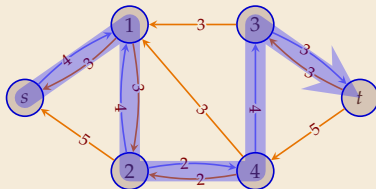
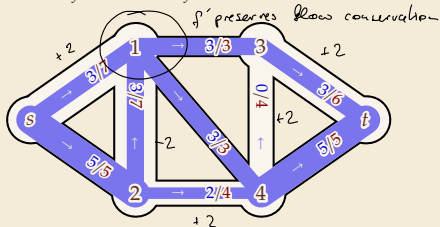
- *residual flow* f' : feasible flow in G_f
 $\rightsquigarrow$ for any f and residual flow f' in G_f , flow $f + f'$ is a feasible flow in G

$$(f + f')(vw) = f(vw) + f'(vw) - f'(wv)$$

Residual Networks

- Goal: Allow undoing flow (without backtracking)
- *Residual network* G_f : given network $G = (V, E)$ and feasible flow f

- $G_f = (V, E_f)$ with capacities $c_f(vw) = \begin{cases} c(vw) - f(vw) & vw \in E \quad // \text{add flow} \\ f(wv) & wv \in E \quad // \text{revert flow} \\ 0 & \text{else} \end{cases}$
 $E_f = \{vw : c_f(vw) > 0\}$



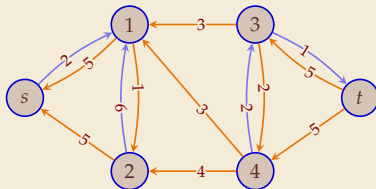
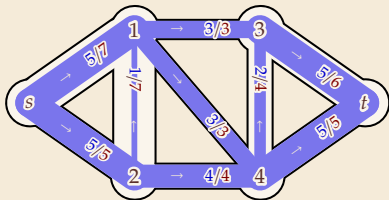
- *residual flow* f' : feasible flow in G_f
 $\rightsquigarrow$ for any f and residual flow f' in G_f , flow $f + f'$ is a feasible flow in G
- *augmenting path* p : s - t -path G_f particularly simple f' !

$$(f + f')(vw) = f(vw) + f'(vw) - f'(wv)$$

Residual Networks

- Goal: Allow undoing flow (without backtracking)
- *Residual network* G_f : given network $G = (V, E)$ and feasible flow f

- $G_f = (V, E_f)$ with capacities $c_f(vw) = \begin{cases} c(vw) - f(vw) & vw \in E \quad // \text{add flow} \\ f(wv) & wv \in E \quad // \text{revert flow} \\ 0 & \text{else} \end{cases}$
 $E_f = \{vw : c_f(vw) > 0\}$

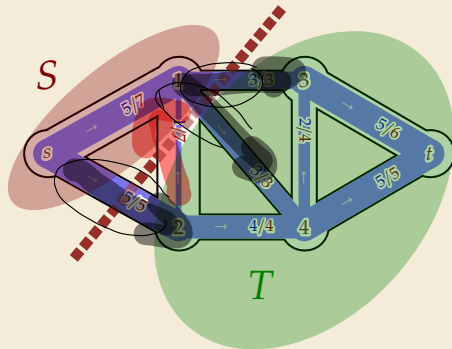


- *residual flow* f' : feasible flow in G_f
 $\rightsquigarrow$ for any f and residual flow f' in G_f , flow $f + f'$ is a feasible flow in G
- *augmenting path* p : s - t -path G_f particularly simple f' !

$$(f + f')(vw) = f(vw) + f'(vw) - f'(wv)$$

Cuts

- **Goal:** Certificate for maximum flows
- *s-t-cut* (S, T) : partition $S \dot{\cup} T = V, s \in S, t \in T$
 - *net flow* across cut:
$$f(S, T) = \sum_{v \in S} \sum_{w \in T} (f(vw) - f(wv))$$
 - *capacity* of cut:
$$c(S, T) = \sum_{v \in S} \sum_{w \in T} c(vw)$$



Cuts

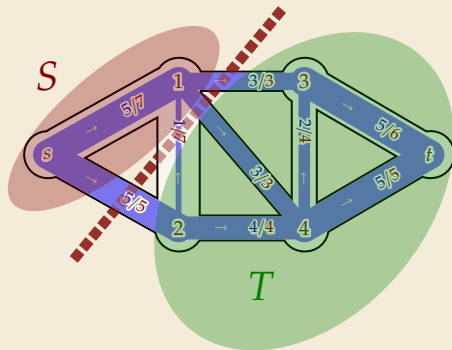
- **Goal:** Certificate for maximum flows
- *s-t-cut* (S, T) : partition $S \dot{\cup} T = V$, $s \in S$, $t \in T$

- *net flow* across cut:

$$f(S, T) = \sum_{v \in S} \sum_{w \in T} (f(vw) - \underline{f(wv)})$$

- *capacity* of cut:

$$c(S, T) = \sum_{v \in S} \sum_{w \in T} c(vw)$$



- $f(S, T) = 5 + 3 + 3 - 1 = 10$

- $c(S, T) = 5 + 3 + 3 = 11$

Cuts

- **Goal:** Certificate for maximum flows
- s - t -cut (S, T) : partition $S \dot{\cup} T = V, s \in S, t \in T$

- *net flow* across cut:

$$f(S, T) = \sum_{v \in S} \sum_{w \in T} (f(vw) - f(wv))$$

- *capacity* of cut:

$$c(S, T) = \sum_{v \in S} \sum_{w \in T} c(vw)$$

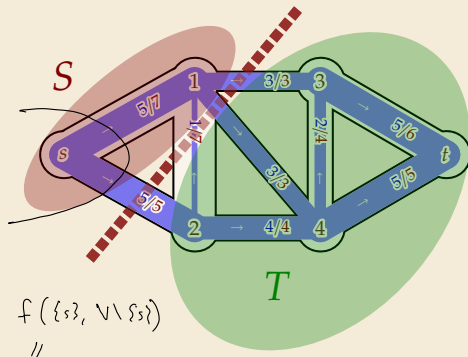
- **Lemma:** For any cut (S, T) , we have $f(S, T) = |f|$.
(flow conservation!)

Proof: Induction on $|S|$

IA. $|S| = 1 \quad s \in S \quad \checkmark$

IS $|S| = k+1$

flow conservation at $\odot v$

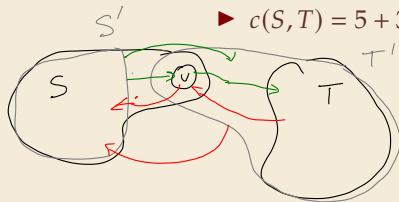


$$f(\{s\}, V \setminus \{s\})$$

//

$$\triangleright f(S, T) = 5 + 3 + 3 - 1 = 10$$

$$\triangleright c(S, T) = 5 + 3 + 3 = 11$$



Cuts

- **Goal:** Certificate for maximum flows
- *s-t-cut* (S, T) : partition $S \dot{\cup} T = V$, $s \in S$, $t \in T$

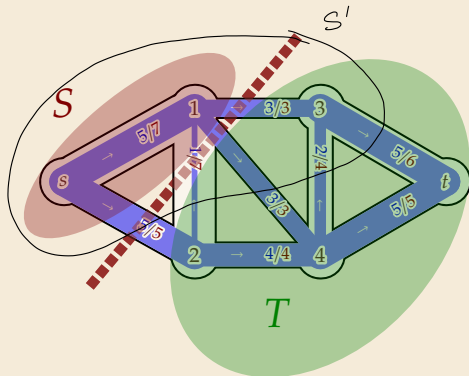
- *net flow* across cut:

$$f(S, T) = \sum_{v \in S} \sum_{w \in T} (f(vw) - f(wv))$$

- *capacity* of cut:

$$c(S, T) = \sum_{v \in S} \sum_{w \in T} c(vw)$$

- **Lemma:** For any cut (S, T) , we have $f(S, T) = |f|$.
(flow conservation!)
- **Corollary:** $|f| \leq c(S, T)$ for any *s-t-cut* (S, T)



$$f(S, T) = 5 + 3 + 3 - 1 = 10$$

$$c(S, T) = 5 + 3 + 3 = 11$$

$$f(S', T') = 10$$

$$c(S', T') = 14$$

The Max-Flow Min-Cut Theorem

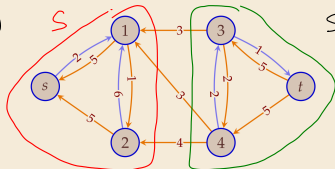
► Max-Flow Min-Cut Theorem:

Let f be a feasible flow in s - t -network $G = (V, E)$. Then the following conditions are equivalent:

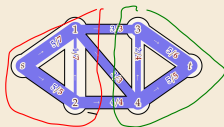
1. $|f| = c(S, T)$ for some cut (S, T) of G .
2. f is a maximum flow in G
3. The residual network G_f has no augmenting path. (path from s to t in G_f)

Proof:

- (1) $\Rightarrow$ (2) $|f| \leq c(S, T)$ for any s - t -cut $(S, T) \Rightarrow$ any f' has $|f'| \leq c(S, T) = |f|$
- (2) $\Rightarrow$ (3) contraposition $\neg(3) \Rightarrow \neg(2)$
 $\Rightarrow G_f$ has path from s to t w/ min. capacity $\varepsilon > 0 \Rightarrow f' \text{ in } G_f \text{ } |f'| = \varepsilon$
 $\Rightarrow f + f'$ is valid flow in G with $|f + f'| = |f| + \varepsilon > |f| \Rightarrow \neg(2)$
- (3) $\Rightarrow$ (1)



$S = \{v : v \text{ reachable from } s\}$



$$c(S, T) = 10 \\ = |f|$$

□

Generic Ford-Fulkerson Method

```
1 procedure genericFordFulkerson( $G = (V, E), s, t, c$ ):  
2   //  $G$  is a flow network with source  $s \in V$ , sink  $t \in V$  and capacities  $c : E \rightarrow \mathbb{R}_{\geq 0}$   
3   for  $vw \in E$  do  $f(vw) := 0$  end for  
4   while  $\exists$  path  $p$  from  $s$  to  $t$  in  $G_f$  // Freedom: Which augmenting path?  
5      $\Delta := \min\{c_f(e) : e \in p\}$  // bottleneck capacity  
6     for  $e \in p$   
7       if  $e \in E$  // forward edge  
8          $f(e) := f(e) + \Delta$   
9       else // backward edge  
10         $f(e) := f(e) - \Delta$   
11   return  $f$ 
```

- Returned flow is a maximum flow f^* (Max-Flow Min-Cut Theorem)

Generic Ford-Fulkerson Method

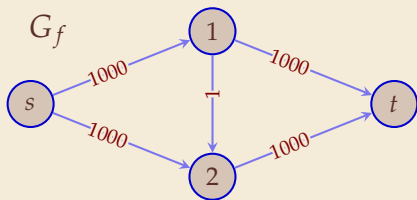
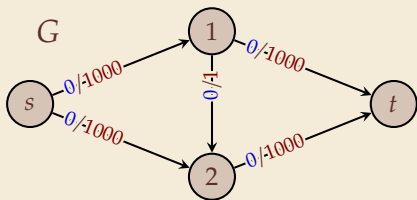
```
1 procedure genericFordFulkerson( $G = (V, E), s, t, c$ ):  
2   //  $G$  is a flow network with source  $s \in V$ , sink  $t \in V$  and capacities  $c : E \rightarrow \mathbb{R}_{\geq 0}$   
3   for  $vw \in E$  do  $f(vw) := 0$  end for  
4   while  $\exists$  path  $p$  from  $s$  to  $t$  in  $G_f$  // Freedom: Which augmenting path?  
5      $\Delta := \min\{c_f(e) : e \in p\}$  // bottleneck capacity  
6     for  $e \in p$   
7       if  $e \in E$  // forward edge  
8          $f(e) := f(e) + \Delta$   
9       else // backward edge  
10         $f(e) := f(e) - \Delta$   
11   return  $f$ 
```

- ▶ Returned flow is a maximum flow f^* (Max-Flow Min-Cut Theorem)
- ▶ If $c : E \rightarrow \mathbb{N}_0$, also $f : E \rightarrow \mathbb{N}_0$: For all $v, w \in V$ holds:
 - ▶ initially $f(vw) = 0 \in \mathbb{N}_0$
 - ▶ $c_f(vw)$ is difference of $c(vw) \in \mathbb{N}_0$ and $f(vw) \in \mathbb{N}_0$
 - ▶ Δ equal to some $c_f(v'w') \in \mathbb{N}_{\geq 1}$ (E_f contains only non-zero capacity edges!)
- $\rightsquigarrow$ new flow $f(vw) \pm \Delta \in \mathbb{N}_0$

$\rightsquigarrow$ For integral capacities, always terminate after $\leq |f^*|$ iterations

Bad Example

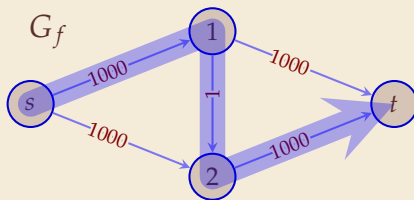
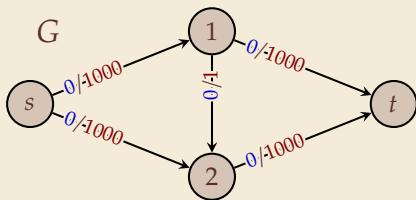
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

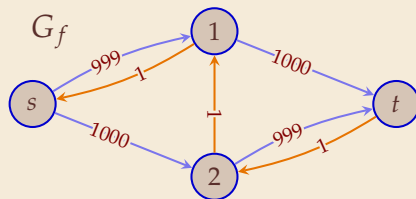
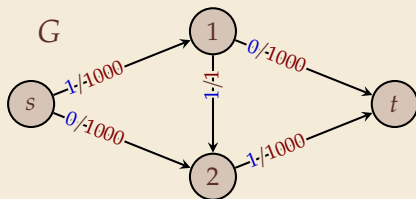
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

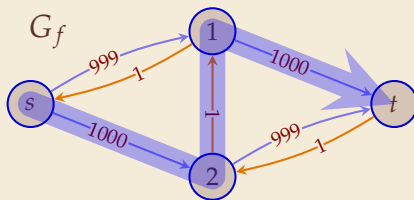
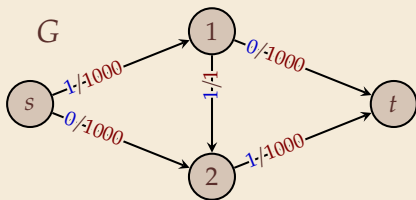
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

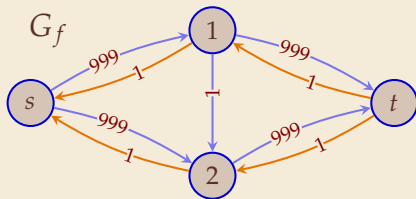
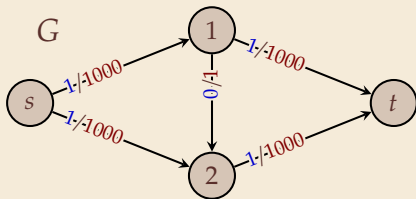
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

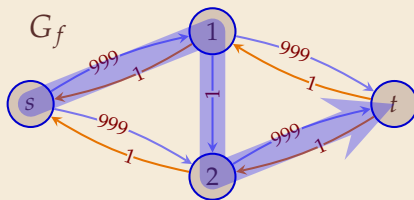
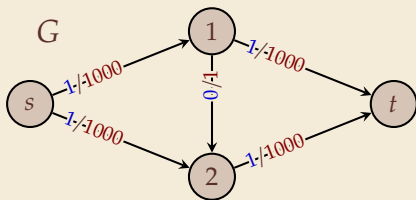
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

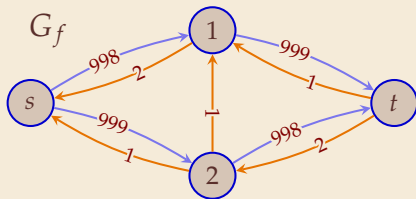
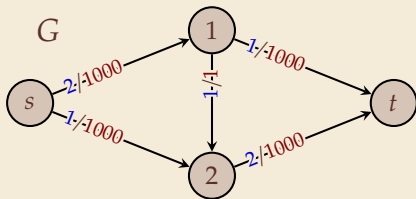
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

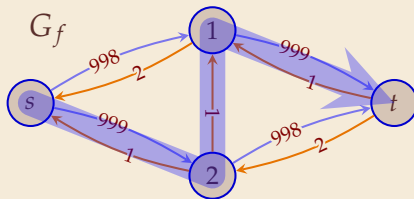
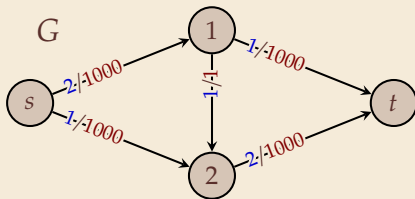
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

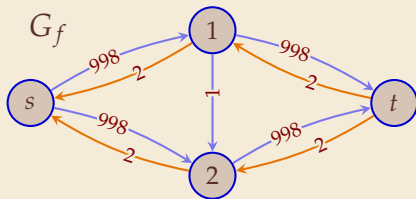
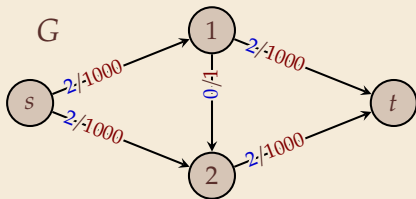
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

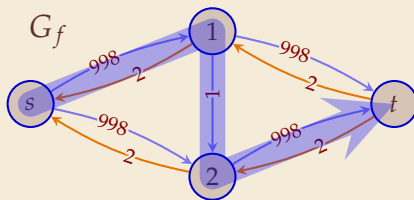
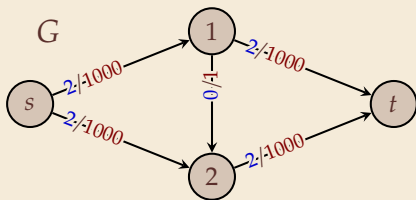
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

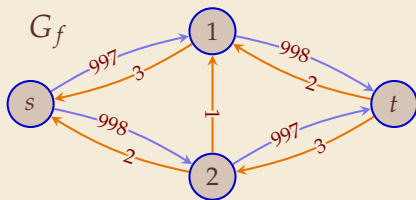
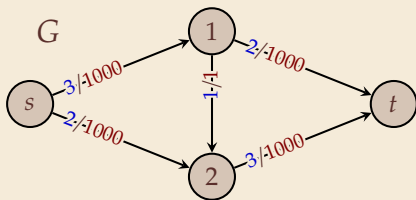
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

Bad Example

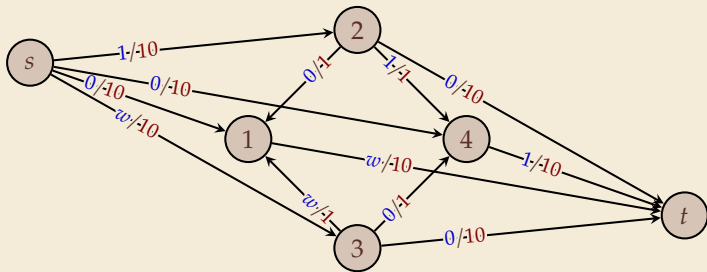
- Unfortunately, we might also take $|f^*|$ iterations!



- (2 iterations with smarter augmenting paths would have sufficed here)

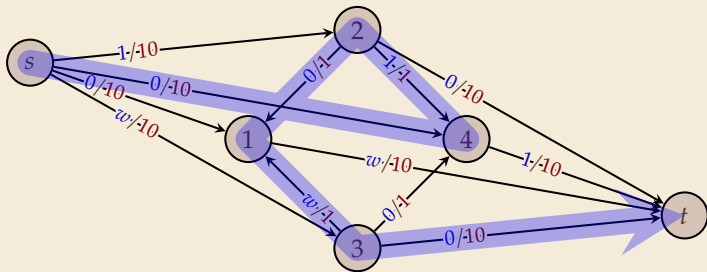
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



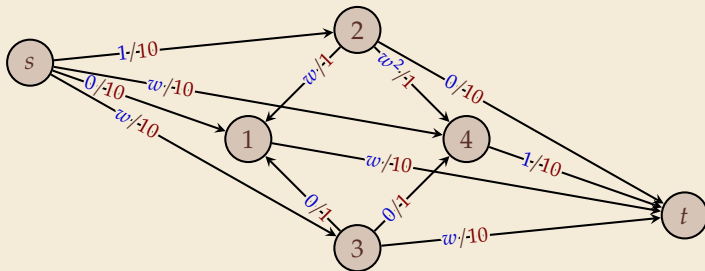
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



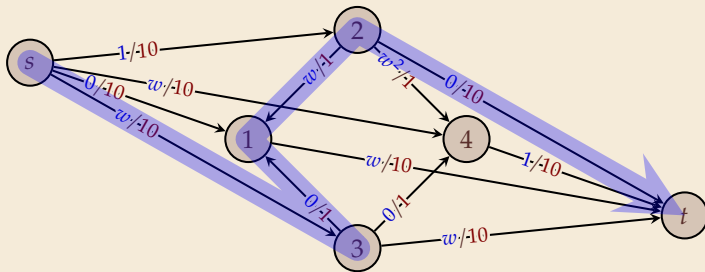
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



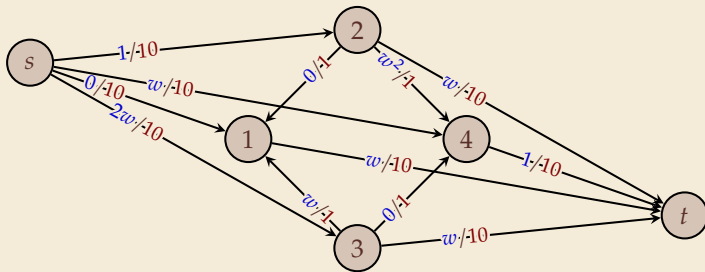
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



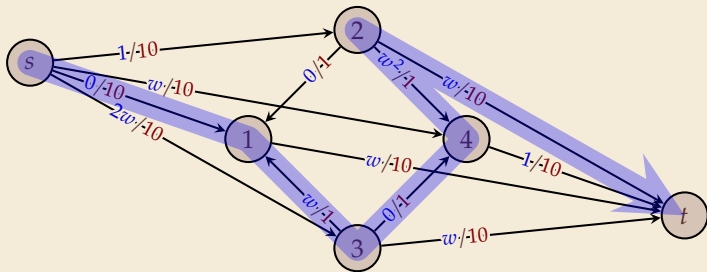
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



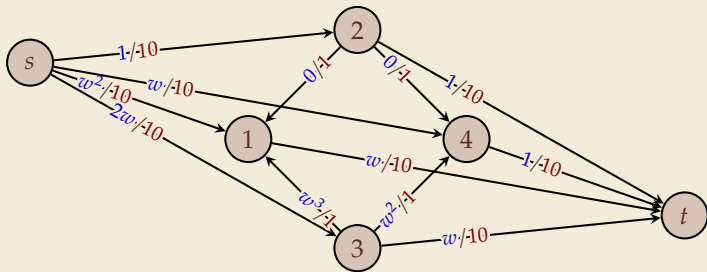
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



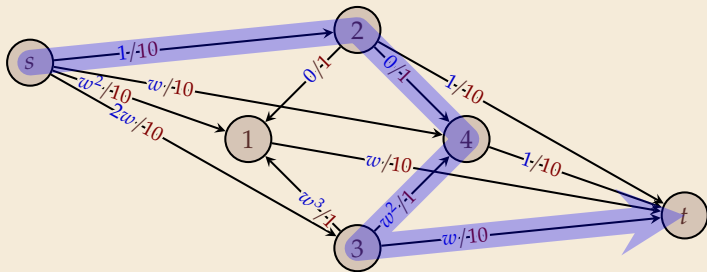
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



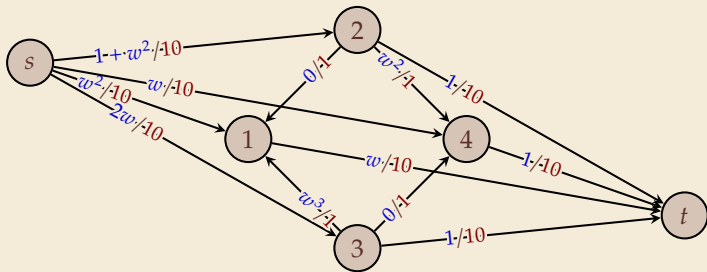
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



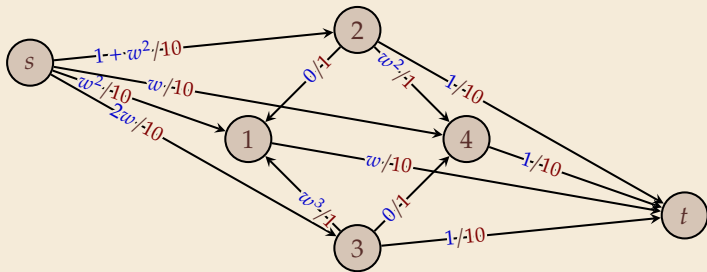
A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



A Very Bad Example

- ▶ for irrational flows, might not even terminate
- ▶ example network with irrational initial flow
- ▶ $w = \varphi - 1 = (\sqrt{5} - 1)/2 \approx 0.618 \rightsquigarrow 1 - w = w^2 \approx 0.382$



- ▶ after 2 paths, situation in 1-2-3-4 restored (rotated), but flows multiplied by w
- $\rightsquigarrow$ augmenting paths have capacities $w, w, w^2, w^2, w^3, w^3 \dots$
- $\rightsquigarrow$ never terminate, never exceed $|f| \geq 5$

9.10 The Edmonds-Karp Algorithm

Edmonds-Karp

- It turns out, many ways to choose augmenting paths systematically work fine
- Edmonds & Karp: take a shortest path (in #edges)

Java code Sedgewick & Wayne

```
1 procedure EdmondsKarp( $G = (V, E), s, t, c$ ):  
2   //  $G$  is a flow network with source  $s \in V$ , sink  $t \in V$  and capacities  $c : E \rightarrow \mathbb{R}_{\geq 0}$   
3   for  $vw \in E$  do  $f(vw) := 0$  end for  
4   while true  
5     bfs( $G_f, \{s\}$ )  
6     if  $\text{distFrom}[t] == \infty$  return  $f$   
7     else  $p := \text{pathTo}(t)$   
8      $\Delta := \min\{c_f(e) : e \in p\}$  // bottleneck capacity  
9     for  $e \in p$   
10      if  $e \in E$  // forward edge  
11         $f(e) := f(e) + \Delta$   
12      else // backward edge  
13         $f(e) := f(e) - \Delta$   
14    end while
```

new

FF

Edmonds-Karp – Analysis

- **Theorem:** The Edmonds-Karp algorithm terminates after $O(nm)$ iterations with a maximum flow. The total running time is in $O(nm^2)$.

Edmonds-Karp – Analysis

- ▶ **Theorem:** The Edmonds-Karp algorithm terminates after $O(nm)$ iterations with a maximum flow. The total running time is in $O(nm^2)$.

- ▶ *Proof Plan:*



- ▶ every augmenting path has a critical edge vw contributing the bottleneck capacity

- ▶ we will show:

- (1) distances of vertices from s in G_f weakly increase over time $S_i(v)$

- (2) before vw can be a *critical* edge *again*, v 's distance increases by at least 2

- ↪ each edge vw is critical for at most $n/2$ augmenting paths (v 's distance $\in [1..n-2]$)

- ↪ $O(nm)$ augmenting paths

- ▶ each iteration runs one BFS, which costs $O(n+m) = O(m)$ times since G is connected.

Edmonds-Karp – Analysis

- ▶ **Theorem:** The Edmonds-Karp algorithm terminates after $O(nm)$ iterations with a maximum flow. The total running time is in $O(nm^2)$.
- ▶ *Proof Plan:*
 - ▶ every augmenting path has a *critical* edge vw contributing the bottleneck capacity
 - ▶ we will show:
 - (1) distances of vertices from s in G_f weakly increase over time
 - (2) before vw can be a *critical* edge *again*, v 's distance increases by at least 2
 - ↪ each edge vw is critical for at most $n/2$ augmenting paths (v 's distance $\in [1..n-2]$)
 - ↪ $O(nm)$ augmenting paths
 - ▶ each iteration runs one BFS, which costs $O(n+m) = O(m)$ times since G is connected.
- ▶ **Notation:**
 - ▶ Write $f_0, f_1, \dots$ for values of f during iterations of while loop
 - ↪ G_{f_i} residual network after i th augmentation
 - ▶ Write $\delta_i(v)$ for shortest-path distance from s to v in G_{f_i}

Edmonds-Karp – Analysis [2]

(1) ► **EK Monotonicity Lemma:** For all i and $v \in V$, we have $\delta_{i+1}(v) \geq \delta_i(v)$.

► f_i : flow after i th augmentation

► $\delta_i(v)$ distance from s to v in G_{f_i}

Edmonds-Karp – Analysis [2]

- ▶ **EK Monotonicity Lemma:** For all i and $v \in V$, we have $\delta_{i+1}(v) \geq \delta_i(v)$.

Proof:

- ▶ by induction over k , the value of $\delta_i(v)$
 - ▶ f_i : flow after i th augmentation
 - ▶ $\delta_i(v)$ distance from s to v in G_{f_i}
- ▶ IB: $k = 0$: only $v = s$ possible; $\delta_{i+1}(s) = 0 \geq 0 = \delta_i(s)$ ✓
- ▶ IH: Assume the claim is true for all shortest paths up to length k

Edmonds-Karp – Analysis [2]

- ▶ **EK Monotonicity Lemma:** For all i and $v \in V$, we have $\delta_{i+1}(v) \geq \delta_i(v)$.

Proof:

▶ f_i : flow after i th augmentation

▶ $\delta_i(v)$ distance from s to v in G_{f_i}

- ▶ by induction over k , the value of $\delta_i(v)$
- ▶ IB: $k = 0$: only $v = s$ possible; $\delta_{i+1}(s) = 0 \geq 0 = \delta_i(s)$ ✓
- ▶ IH: Assume the claim is true for all shortest paths up to length k
- ▶ IS: Suppose $\delta_{i+1}(v) = k + 1$.

$\rightsquigarrow \exists$ shortest path $p[0..k+1]$ in $G_{f_{i+1}}$ with $p[0] = s$ and $p[k+1] = v$.

$\rightsquigarrow$ For $w = p[k]$, $p[0..k]$ is a shortest path from s to w $\rightsquigarrow k = \delta_{i+1}(w) \underset{\text{IH}}{\geq} \delta_i(w)$

$G_{f_{i+1}}$



Edmonds-Karp – Analysis [2]

- **EK Monotonicity Lemma:** For all i and $v \in V$, we have $\delta_{i+1}(v) \geq \delta_i(v)$.

Proof:

- by induction over k , the value of $\delta_i(v)$
- IB: $k = 0$: only $v = s$ possible; $\delta_{i+1}(s) = 0 \geq 0 = \delta_i(s)$ ✓
- IH: Assume the claim is true for all shortest paths up to length k
- IS: Suppose $\delta_{i+1}(v) = k + 1$.

↪ $\exists$ shortest path $p[0..k+1]$ in $G_{f_{i+1}}$ with $p[0] = s$ and $p[k+1] = v$.

↪ For $w = p[k]$, $p[0..k]$ is a shortest path from s to w ↪ $k = \delta_{i+1}(w) \geq \delta_i(w)$
IH

► Case 1: $wv \in E_{f_i}$ ↪ $\delta_i(v) \leq \delta_i(w) + 1$

► Case 2: $wv \notin E_{f_i}$ ↪ reverse edge vw in i th augmenting path, a shortest s - t -path
↪ $\delta_i(v) = \delta_i(w) - 1 \leq \delta_i(w) + 1$

► in both cases: $\delta_{i+1}(v) = \delta_{i+1}(w) + 1 \geq \delta_i(w) + 1 \geq \delta_i(v)$

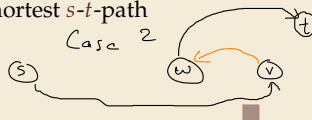
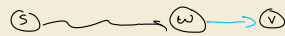
► f_i : flow after i th augmentation

► $\delta_i(v)$ distance from s to v in G_{f_i}

$G_{f_{i+1}}$



G_{f_i} : Case 1



Edmonds-Karp – Analysis [3]

- ▶ **Critical Distance Lemma:** When critical edge vw becomes a critical again, $\delta(v)$ has increased by at least 2.

Edmonds-Karp – Analysis [3]

- ▶ **Critical Distance Lemma:** When critical edge vw becomes a critical again, $\delta(v)$ has increased by at least 2.

Proof:

- ▶ Suppose vw is critical in i th iteration $\rightsquigarrow$ lies on shortest path

$$\rightsquigarrow \delta_i(w) = \delta_i(v) + 1$$

- ▶ before vw reappears in G_f , need to have had wv in augmenting path;
say this first happens in iteration $j > i$ $\rightsquigarrow \delta_j(v) = \delta_j(w) + 1$

Edmonds-Karp – Analysis [3]

- ▶ **Critical Distance Lemma:** When critical edge vw becomes a critical again, $\delta(v)$ has increased by at least 2.

Proof:

- ▶ Suppose vw is critical in i th iteration $\rightsquigarrow$ lies on shortest path

$$\rightsquigarrow \delta_i(w) = \delta_i(v) + 1$$

- ▶ before vw reappears in G_f , need to have had wv in augmenting path;
say this first happens in iteration $j > i$ $\rightsquigarrow \delta_j(v) = \delta_j(w) + 1$

- ▶ by EK Monotonicity Lemma:

$$\delta_j(v) = \delta_j(w) + 1 \geq \delta_i(w) + 1 = \delta_i(v) + 2$$

This concludes the proof of the theorem.

Maximum Flow – Discussion

 Edmonds-Karp is a robust choice ($O(nm^2)$ time independent of numbers)

 easy to implement (see Sedgewick Wayne for an elegant Java version!)

 worst-case time $O(n^5)$ for dense graphs quickly prohibitive

- ▶ but: worst-case results typically overly pessimistic
- ▶ other choices of augmenting flows possible
- ▶ in practice: push-relabel methods often faster

Maximum Flow – Discussion



Edmonds-Karp is a robust choice ($O(nm^2)$ time independent of numbers)



easy to implement (see Sedgewick Wayne for an elegant Java version!)



worst-case time $O(n^5)$ for dense graphs quickly prohibitive

- ▶ but: worst-case results typically overly pessimistic
 - ▶ other choices of augmenting flows possible
 - ▶ in practice: push-relabel methods often faster
-
- ▶ 2022 theory breakthrough: almost linear(!) $O(m^{1+o(1)})$ time max flow algorithm
Chen, Kyng, Liu, Peng, Gutenberg & Sachdeva, FOCS 2022

Maximum Flow – Discussion



Edmonds-Karp is a robust choice ($O(nm^2)$ time independent of numbers)



easy to implement (see Sedgewick Wayne for an elegant Java version!)



worst-case time $O(n^5)$ for dense graphs quickly prohibitive

- ▶ but: worst-case results typically overly pessimistic
- ▶ other choices of augmenting flows possible
- ▶ in practice: push-relabel methods often faster

▶ 2022 theory breakthrough: almost linear(!) $O(m^{1+o(1)})$ time max flow algorithm
Chen, Kyng, Liu, Peng, Gutenberg & Sachdeva, FOCS 2022

▶ max-flow min-cut theorem is a special case of LP duality

▶ can also solve generalization of min-cost flows

- ▶ each edge vw has a cost $a(vw)$
- ▶ cost of a flow f : $\sum a(vw) \cdot f(vw)$
- ▶ demand d at sink becomes part of constraints: $|f| \geq d$