9 Graph Algorithms

9 December 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-12-16 09:28`

Learning Outcomes

Unit 9: ***Graph Algorithms***

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

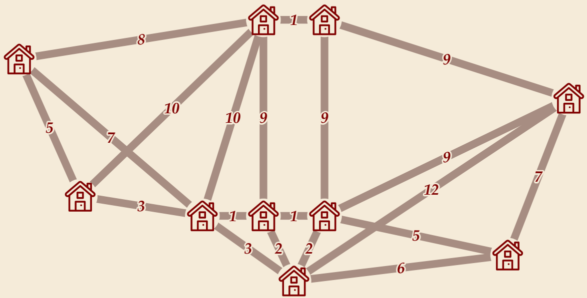
Outline

9 Graph Algorithms

9.1 Introduction & Definitions

Graphs in real life

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



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

`https://www.coursera.org/learn/algorithms-part2`

Flavors of Graphs

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

Property	Yes	No
edges are one-way	*directed* graph (*digraph*)	*undirected* graph
≤ 1 edge between u and v	*simple* graph	*multigraph* / with *parallel* edges
edges can lead from v to v	with *loops* (Schlinge, Schleife)	(loop-free)
edges have weights	*(edge-) weighted* graph	*unweighted* graph

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

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

Graph Theory

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

$e = \{u, v\}$

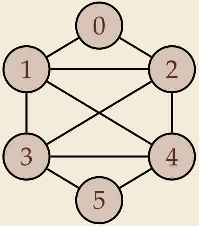
Graph Theory

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

Example

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

Graphical representation



like so ...

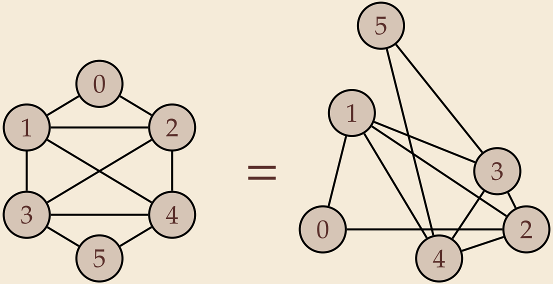
Graph Theory

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

Example

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

Graphical representation

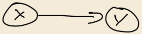


like so . . .   . . . or so

(same graph)

Digraphs

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



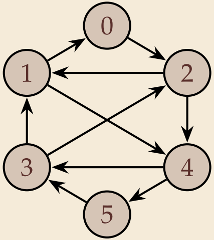
Digraphs

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

Example

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

Graphical representation



Graph Terminology

Undirected Graphs

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

Directed Graphs (where different)

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

Graph Terminology

Undirected Graphs

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

Directed Graphs (where different)

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

Kantenzug

- *walk* w of length n: sequence of vertices $w[0..n]$ with $\forall i \in [0..n) : w[i]w[i+1] \in E$
- *path* p is a (vertex-) simple walk: without duplicate vertices except possibly its endpoints
- *edge-simple* walk: no edge used twice
- *cycle* c is a closed path, i. e., $c[0] = c[n]$ (geschlossener Weg, Zykel, Kreis, Zyklus)

Graph Terminology

Undirected Graphs

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

Directed Graphs (where different)

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

- *walk* w of length n: sequence of vertices $w[0..n]$ with $\forall i \in [0..n) : \underline{w[i]w[i+1] \in E}$
- *path* p is a (vertex-) simple walk: without duplicate vertices except possibly its endpoints
- *edge-simple* walk: no edge used twice
- *cycle* c is a closed path, i. e., $c[0] = c[n]$

- G is *connected* iff for all $u \neq v \in V$ there is a path from u to v
- G is *acyclic* iff $\nexists$ cycle (of length $n \geq 1$) in G

- *strongly connected* for digraphs (*weakly connected* = connected ignoring directions)

Typical graph-processing problems

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

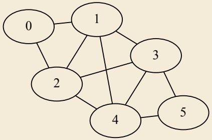
can vary a lot, despite superficial similarity of problems

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

Tools to work with graphs

- Convenient GUI to edit & draw graphs: *yEd live*
  `yworks.com/yed-live`
- *graphviz* cmdline utility to draw graphs
  - Simple text format for graphs: `DOT`

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



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

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

9.2 Graph Representations

Graphs in Computer Memory

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

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

  - which is better depends on the required operations

Graphs in Computer Memory

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

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

- which is better depends on the required operations

Key Operations:

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

Graphs in Computer Memory

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

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

  - which is better depends on the required operations

Key Operations:

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

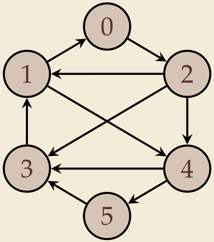
Conventions:

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

Adjacency Matrix Representation

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

Example:

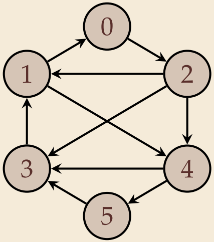


$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 &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

Adjacency Matrix Representation

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

Example:



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

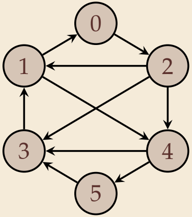
`isAdjacent` in $O(1)$ time

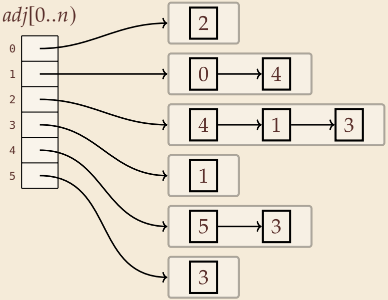
$O(n^2)$ (bits of) space wasteful for sparse graphs

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

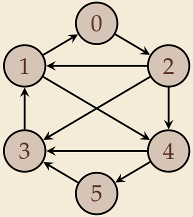
Adjacency List Representation

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

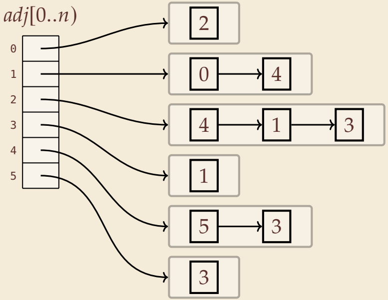




Adjacency List Representation



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



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

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

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

$\rightsquigarrow$ de-facto standard for graph algorithms

Graph Types and Representations

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

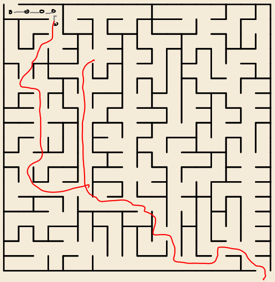
9.3 Graph Traversal

Generic Graph Traversal

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

  visiting all nodes & edges

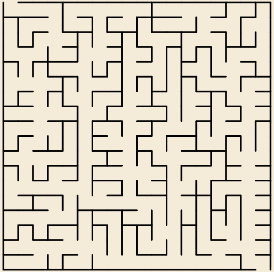
  - depth-first search, breadth-first search
  - connected components
  - detecting cycles
  - topological sorting
  - Hierholzer's algorithm for Euler walks
  - strong components
  - testing bipartiteness
  - Dijkstra's algorithm
  - Prim's algorithm
  - Lex-BFS for perfect elimination orders of chordal graphs
  - ...



Generic Graph Traversal

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



$\leadsto$ Formulate generic traversal algorithm
  - first in abstract terms to argue about correctness
  - then again for concrete instance with efficient data structures

Tricolor Graph Traversal

Tricolor Graph Search:

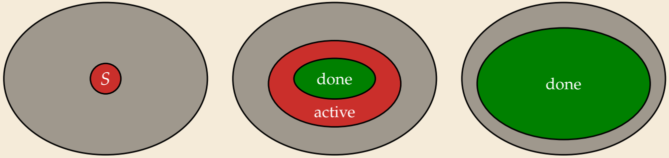
- maintain vertices in 3 (dynamic) sets
  - **Gray: unseen vertices**
    The traversal has not reached these vertices so far.
  - **Green: done vertices** (a.k.a. visited vertices)
    These vertices have been visited and all their edges have been explored already.
  - **Red: active vertices** (a.k.a.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

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

want to color green
⇝ to do that need neighbors have red

- (implicitly) maintain status of each edge
  - **not yet used**
  - **used edge**

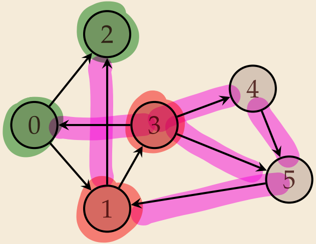
- Vertices “want” to turn green.



initial state | during traversal | final state

Generic Tricolor Graph Traversal – Code

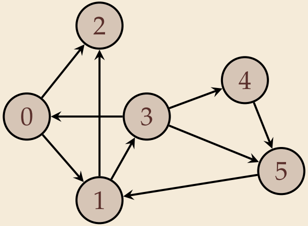
```
procedure genericGraphTraversal(G, S)
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // Color array, all cells initialized to unseen
    for s ∈ S do C[s] := active end for
    unusedEdges := E
    while ∃v : C[v] == active
        v := nextActiveVertex()  // Freedom 1: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



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

Generic Tricolor Graph Traversal – Code

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

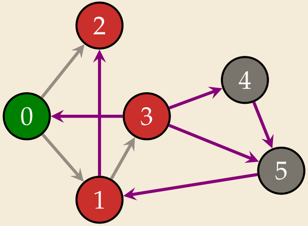


Invariant:
No edges from ***done*** to ***unseen*** vertices

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

Generic Tricolor Graph Traversal – Code

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



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

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

Generic Reachability

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

Generic Reachability

- Any choices nextActiveVertex() and nextUnusedEdge(v) suffice to find exactly the vertices reachable from S in ***done***
- **Invariant:**
  1. No edges from ***done*** to ***unseen*** vertices
  2. For every ***done*** or ***active*** vertex v, there exists a path from $s \in S$ to v.

IB: (1) ✓ (2) ✓

IS: (i) line 9 color v green

no edges to unseen from v

(1) ✓

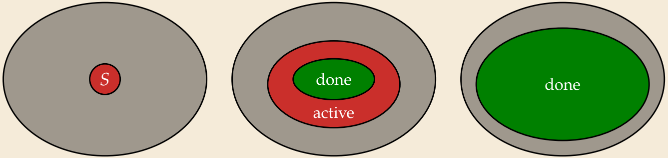
(2) by IH ✓

(ii) line 11 color w red

(1) ✓

(2) by IH $s \rightsquigarrow v \to w$

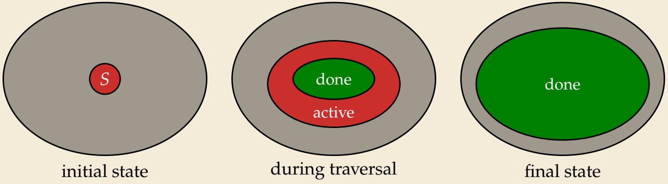
hence $s \rightsquigarrow w$ ✓



initial state — during traversal — final state

Generic Reachability

- Any choices nextActiveVertex() and nextUnusedEdge(v) suffice to find exactly the vertices reachable from S in ***done***
- **Invariant:**
  1. No edges from ***done*** to ***unseen*** vertices
  2. For every ***done*** or ***active*** vertex v, there exists a path from $s \in S$ to v.



$\rightsquigarrow$ in final state:

- $v \in$ ***done*** $\rightsquigarrow$ path from S $\rightsquigarrow$ reachable from S
- $v \in$ ***unseen*** $\rightsquigarrow$ not reachable from ***done*** $\supseteq S$ $\rightsquigarrow$ not reachable from S

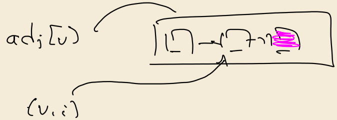
(assume not. then $s \rightsquigarrow v$ first vertex on path that is unseen has done neighbor ↯ (1)

Data Structures for Frontier

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

Data Structures for Frontier

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



- Typical solution maintains **bag** "*frontier*" of *pairs* (v, i) where $v \in V$ and i is an **iterator** in `adj`$[v]$
  - *unusedEdges* represented implicitly: edge used iff previously returned by i

    $\rightsquigarrow$ don't need *unusedEdges*.remove(vw)

Data Structures for Frontier

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

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

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

9.4 BFS and DFS

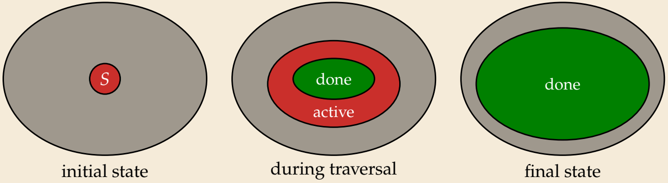
Breadth-First Search

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

Breadth-First Search

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

- **Invariant:**
  1. No edges from done to unseen vertices
  2. All *done* & active or *active* vertices are reached via a **shortest path** from S fewest edges
  3. Vertices enter and leave *frontier* in order of increasing distance from S



$\rightsquigarrow$ in final state, we reach all reachable vertices via shortest paths

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

= #edges = length of shortest path from s to v

Proof of BFS Invariant. (corrected version)

(1) follows from the generic proof for tricolor traversal

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

IB: (2) only $s \in S$ are active or done.

we reach s via path $[s]$ (length 0) ✓

(3) only $s \in S$ have entered frontier so far. these are minimal wrt. δ

IH: Invariant is fulfilled up to now.

IS: consider next active vertex v.

(i) v has no more unused edges. $\Rightarrow$ mark v as done

(2) v only changes color $\Rightarrow$ still holds by IH

(3) by IH, v entered frontier at correct time & queue keeps order $\Rightarrow$ (3) still holds

(ii) v has another edge vw. We further distinguish:

(ii-a) w is active or done $\Rightarrow$ no changes

$\Rightarrow$ invariant still holds

(ii-b) w is unseen $\Rightarrow$ enqueue w and make it active

by IH, we reached v by shortest path of $\delta(v)$ edges

and any node u with $\delta(u) < \delta(v)$ is done (since before v)

$\Rightarrow$ $\delta(w) = \delta(v) + 1$ and we reach w via a shortest path $\Rightarrow$ (2) ✓

It remains to show that any x with $\delta(x) = \delta(v)$ is active or done by now;

then adding w to frontier now is according to sorted order by δ.

Suppose the path to x goes via u : $s \rightsquigarrow u \to x$ then $\delta(u) < \delta(v)$ and u is done

$\Rightarrow$ all edges $u \to x$ have been used and thus x is active or done

(We cannot have active or done y with $\delta(y) > \delta(w)$ in frontier by IH;

their addition before w would have violated the invariant at that time.)

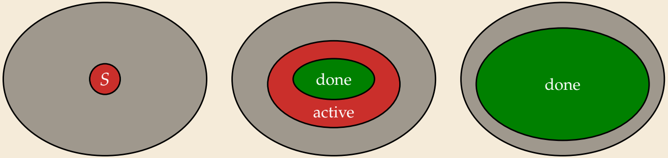
$\Rightarrow$ (3) ✓

$\square$.

Breadth-First Search

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

- **Invariant:**
  1. No edges from done to unseen vertices
  2. All ***done*** or ***active*** vertices are reached via a **shortest path** from S fewest edges
  3. Vertices enter and leave *frontier* in order of increasing distance from S



initial state — during traversal — final state

$\leadsto$ in final state, we reach all reachable vertices via shortest paths

- To preserve that knowledge, we collect extra information during traversal
  - *parent*[v] stores predecessor on path from S via which v was reached time when v was colored red
  - *distFromS*[v] stores the length of this path

Breadth-First Search – Code

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

Breadth-First Search – Code

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

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

Breadth-First Search – Code

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

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

Depth-First Search

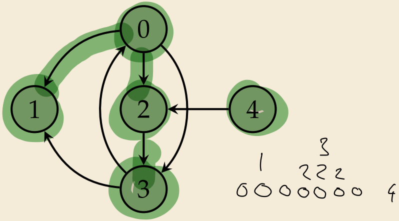
- Maintain *frontier* in a **stack** (LIFO: last in, first out)
  - only consider $S = \{s\}$
  - usual mode of operation: call `dfs`(v) for all ***unseen*** v, for $v = 0, \ldots, n-1$

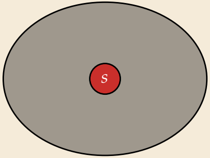
Depth-First Search

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

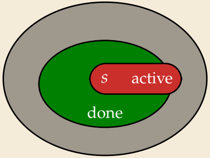
- **Invariant:**

  1. No edges from done to unseen vertices
  2. All ***done*** & active or ***active*** vertices are reached via a path from s
  3. The ***active*** vertices form a single **path** from s

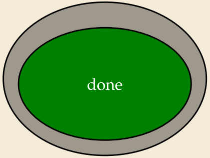




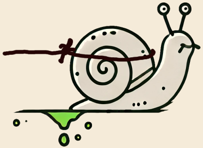
initial state



during traversal



final state



Depth-First Search – Code

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

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

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

Depth-First Search – Code

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

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

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

Simple DFS Application: Connected Components

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

Simple DFS Application: Connected Components

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

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

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

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

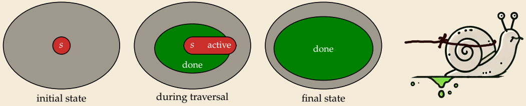
Dijkstra's Algorithm & Prim's Algorithm

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

$\rightsquigarrow$ Detailed discussion in Unit 11

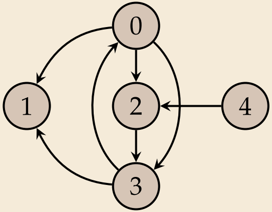
9.5 Advanced Uses of DFS

Properties of DFS



- **Recall DFS Invariant 3:**
  The *active* vertices form a single **path** from s

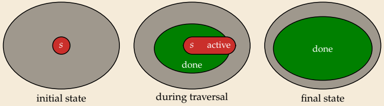
input graph G

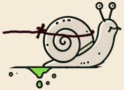


DFS forest

stack over time

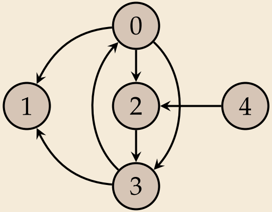
Properties of DFS



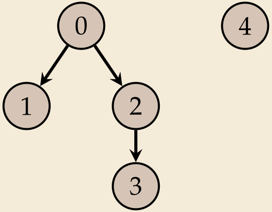


- **Recall DFS Invariant 3:**
  The *active* vertices form a single **path** from s

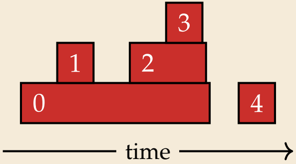
input graph G



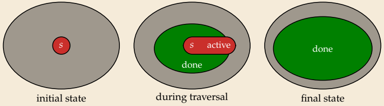
DFS forest

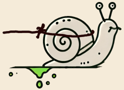


stack over time



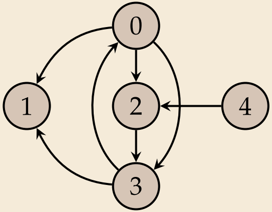
Properties of DFS



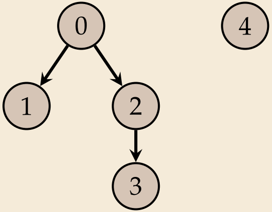


- **Recall DFS Invariant 3:**
  The *active* vertices form a single **path** from s

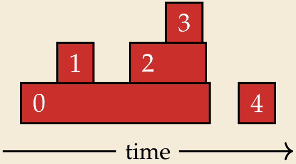
input graph G



DFS forest

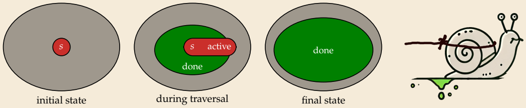


stack over time



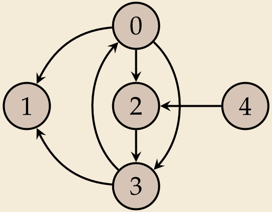
$\rightsquigarrow$ Each vertex v spends *time interval* $T(v)$ as *active* vertex

Properties of DFS

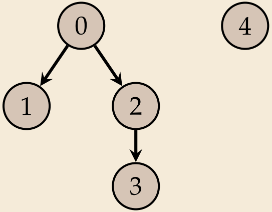


- **Recall DFS Invariant 3:**
  The *active* vertices form a single **path** from s

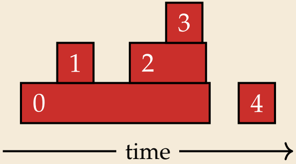
input graph G



DFS forest



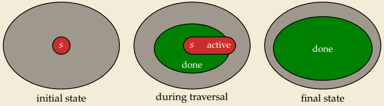
stack over time

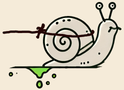


$\rightsquigarrow$ Each vertex v spends *time interval* $T(v)$ as *active* vertex

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

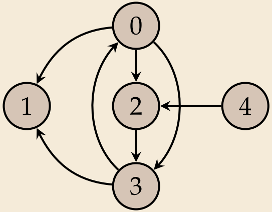
Properties of DFS



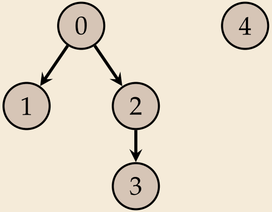


- **Recall DFS Invariant 3:**
  The *active* vertices form a single **path** from s

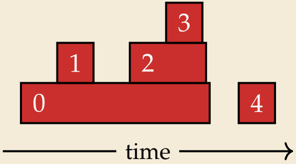
input graph G



DFS forest



stack over time



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

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

2. **Parenthesis Theorem:** $T(v) \supseteq T(w)$ **iff** v is ancestor of w in DFS tree
   '$\Rightarrow$' during $T(v)$, all discovered vertices become descendants of v
   '$\Leftarrow$' $T(v)$ covers v's entire subtree, which contains w's subtree

Properties of DFS – Unseen-Path Theorem

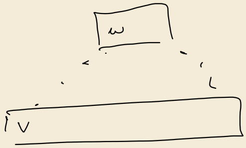
- **Unseen-Path Theorem:** In a DFS forest of a (di)graph G, w is a descendant of v **iff** at the time of `preorderVisit`(v), there is a path from v to w using only *unseen* vertices.

at the time v was made red

Properties of DFS – Unseen-Path Theorem

- **Unseen-Path Theorem:** In a DFS forest of a (di)graph G, $\overset{w}{\cancel{w}}$ is a descendant of $\overset{v}{\cancel{v}}$ **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.



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.

'⇒' 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.

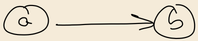
'⇐' Suppose towards a contradiction that there was a w with an *unseen* path $p[0..\ell]$ with $p[0] = v$ and $p[\ell] = w$, but w is not a descendant of v. W.l.o.g. let w be a first such vertex, i. e., $p[0], \ldots, p[\ell - 1] = u$ are descendants of v.
So $T(u) \subset T(v) \quad (*)$.
Upon processing 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 $\max T(w) \le \max T(u)$.
With $(*)$, we obtain $\min T(v) \le \min T(u) \le \max T(w) \le \max T(u)$, so by 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



Topological Sorting & Cycle Detection

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

  ⇝ Model as directed graph!
  - tasks are the vertices V
  - add an edge (a, b) when a must be done before b

- **Definition:** $R[0..n)$ is a *topological (order) ranking* of digraph $G = (V, E)$ if $\forall (u, v) \in E : R[u] < R[v]$

  think: R[v] time slot for v

- **Lemma DAG iff topo:**
  A directed graph G has a topological ranking **iff** it does not contain a directed cycle.

Topological Sorting & Cycle Detection

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

  $\rightsquigarrow$ Model as directed graph!
  - tasks are the vertices V
  - add an edge (a, b) when a must be done before b

- **Definition:** $R[0..n)$ is a *topological (order) ranking* of digraph $G = (V, E)$ if $\forall(u, v) \in E : R[u] < R[v]$

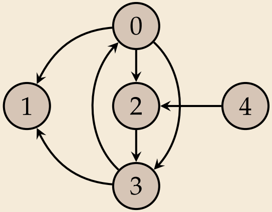
- **Lemma DAG iff topo:**
  A directed graph G has a topological ranking **iff** it does not contain a directed cycle.

- **Topological Sorting**
  - **Given:** simple digraph $G = (V, E)$
  - **Goal:** Compute topological ranking of vertices $R[0..n)$ or output a directed cycle in G.

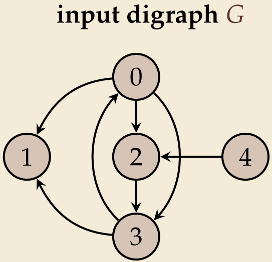
- Amazingly, can do all with one pass of DFS!

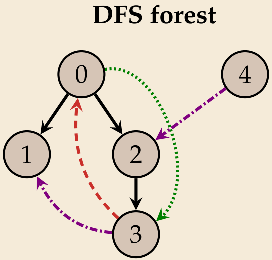
DFS Edge Types

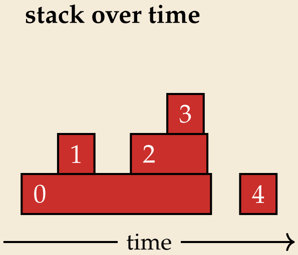
input digraph G



DFS Edge Types

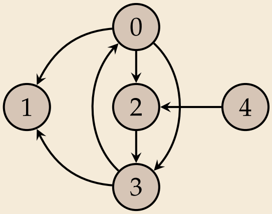




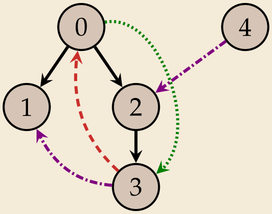


DFS Edge Types

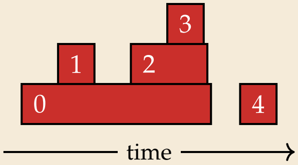
input digraph G



DFS forest



stack over time



- During DFS traversal, an edge vw has one of these 4 types: **example:**

1. **tree edge:** $\longrightarrow$ $w \in$ *unseen* $\rightsquigarrow$ vw part of DFS forest. $(0,1), (0,2), (2,3)$
2. **back edges:** $\dashrightarrow$ $w \in$ *active*; $\rightsquigarrow$ w points to ancestor of v. $(3,0)$
3. **forward edges*:** $\cdots\rightarrow$ $w \in$ *done* $\wedge$ w is descendant of v in DFS tree. $(0,3)$
4. **cross edges*:** $\dashrightarrow$ $w \in$ *done* $\wedge$ w is not descendant of v. $(3,0)$

*only possible in *directed* graphs

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

DFS Postorder Implementation

```
procedure dfsPostorder(G):
    C[0..n) := unseen
    P[0..n) := NONE;  r := 0
    parent[0..n) := NONE
    cycle := NONE
    for v := 0, ..., n − 1
        if C[v] == unseen
            dfs(G, v)
    return (P, cycle)

procedure postorderVisit(v):
    P[v] := r;  r := r + 1

procedure visitEdge(vw):
    if C[w] == active
        if cycle ≠ NONE return
        while v ≠ w
            cycle.append(v)
            v := parent[v]
        cycle.append(v)
```

```
// dfs is as in CC but with parent
procedure dfs(G, s)
    frontier := new Stack;
    parent[s] := NONE;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop()
            postorderVisit(v)
        else
            w := i.next() // Advance i in adj[v]
            visitEdge(vw)
            if C[w] == unseen
                parent[w] := v;
                preorderVisit(w)
                C[w] := active;  frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

DFS Postorder & Topological Sort

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

DFS Postorder & Topological Sort

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

- **Lemma rev postorder:**
  directed acyclic graph
  Let G be a simple, connected DAG and $R[0..n)$ a *reverse DFS postorder* of G, i. e., $R[v] = n - 1 - P[v]$ for a DFS postorder $P[0..n)$. Then R is a topological ranking of G.

- **Invariant:** If $v \in$ ***done*** and $(v, w) \in E$ then $w \in$ ***done*** and $R[v] < R[w]$.
  - initially true (***done*** $= \emptyset$)
  - upon `postorderVisit`(v), all outgoing edges vw lead to $w \in$ ***done*** (Parenthesis Theorem)

Topological Sorting & Cycle Detection – Summary

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

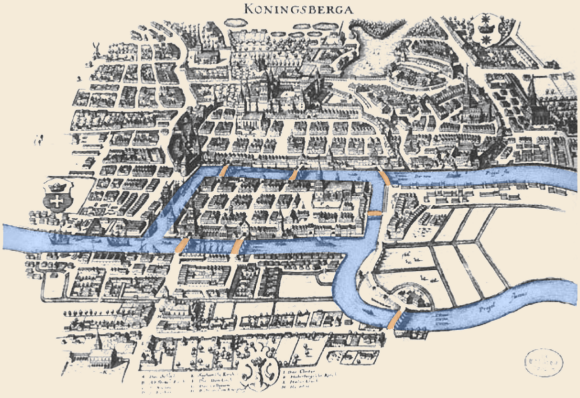
```
procedure topologicalRanking(P):
    (P[0..n), cycle) := dfsPostorder(G)
    if cycle ≠ NULL
        return NOT_A_DAG
    R[0..n) := NONE
    for v := 0, ..., n − 1
        R[v] = n − 1 − P[v]
    return R
```

```
procedure topologicalSort(P):
    (P[0..n), cycle) := dfsPostorder(G)
    if c ≠ NULL
        return NOT_A_DAG
    S[0..n) := NONE
    for v := 0, ..., n − 1
        S[n − 1 − P[v]] := v
    return S
```

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

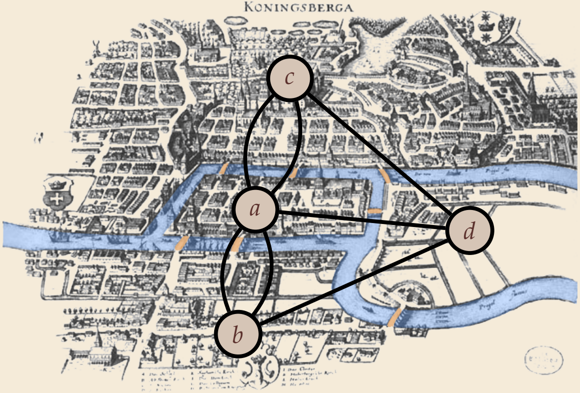
Euler Cycles

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



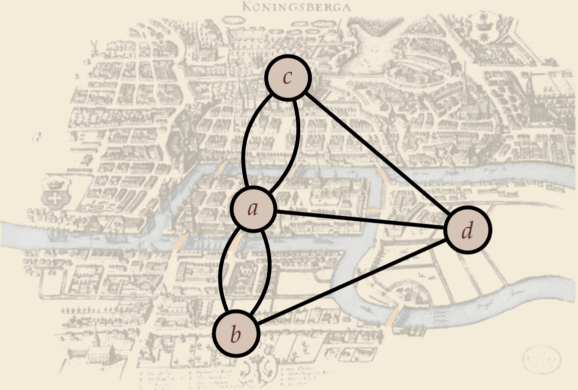
Euler Cycles

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.

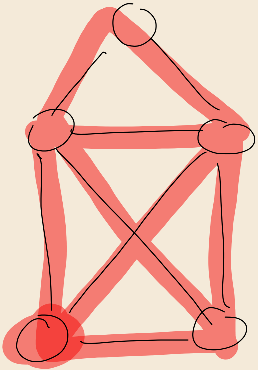


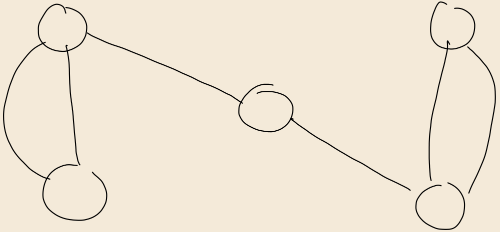
Euler's Theorem:

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

```
procedure eulerWalk(G):
    // Assume G = (V, E) is connected (multi)graph
    V_odd := {v ∈ V : d(v) odd}
    if |V_odd| ∉ {0, 2} return NOT_EULERIAN
    if V_odd = {x, y} then s := x else s := 0
    euler[0..m) := NONE;   j := m − 1
    visited[0..n, 0..n) := false // mark edges as visited
    for v := 0, . . . , n − 1
        // globally remember next unexplored edge
        nextEdge[v] := G.adj[w].iterator())
    edgeDFS(s)
    return euler
```

```
procedure edgeDFS(s):
    frontier := new Stack;
    frontier.push(s)
    while ¬frontier.isEmpty()
        v := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            frontier.pop()
            if ¬frontier.isEmpty()
                // assign edge leading here largest free index
                euler[j] := (frontier.top(), v);   j := j − 1
            end if
        else
            w := i.next()
            if ¬visited[v, w]
                visited[v, w] := true
                visited[w, v] := true
                frontier.push(w)
            end if
        end if
    end while
```

Clicker Question

Mark all correct statements about a `dfsTraversal` (Slide 21) 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

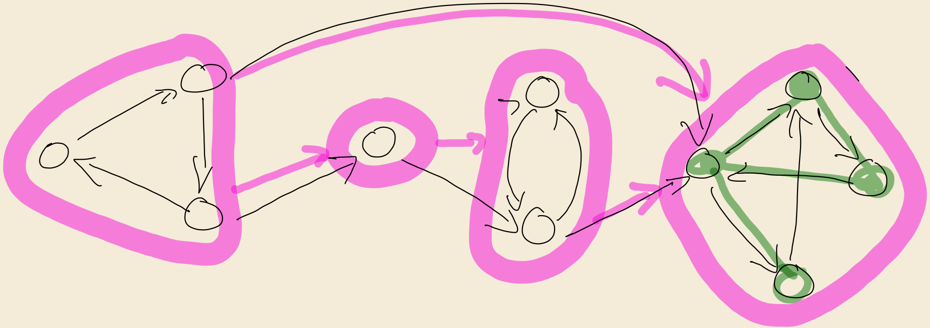
Mark all correct statements about a `dfsTraversal` (Slide 21) of a **DAG** G:

- 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 intro single vertices
  $V(G^{\text{SCC}}) = \{C_1, \ldots, C_k\}$ with $C_1 \dot\cup \cdots \dot\cup C_k = V$;
  name by smallest vertex s.t. $i \le j$ iff $\min C_i \le \min C_j$
  - can't have cycles (↯ maximality of SCC)
  - $\rightsquigarrow$ component DAG has a topological order $R^{\text{SCC}}[1..k]$

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 intro single vertices
  $V(G^{\mathrm{SCC}}) = \{C_1, \ldots, C_k\}$ with $C_1 \dot\cup \cdots \dot\cup C_k = V$;
  name by smallest vertex s.t. $i \le j$ iff $\min C_i \le \min C_j$
  - can't have cycles (↯ maximality of SCC)

  $\rightsquigarrow$ component DAG has a topological order $R^{\mathrm{SCC}}[1..k]$

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 intro single vertices
  $V(G^{\mathrm{SCC}}) = \{C_1, \ldots, C_k\}$ with $C_1 \dot\cup \cdots \dot\cup C_k = V$;
  name by smallest vertex s.t. $i \le j$ iff $\min C_i \le \min C_j$
  - can't have cycles (↯ maximality of SCC)

  $\rightsquigarrow$ component DAG has a topological order $R^{\mathrm{SCC}}[1..k]$

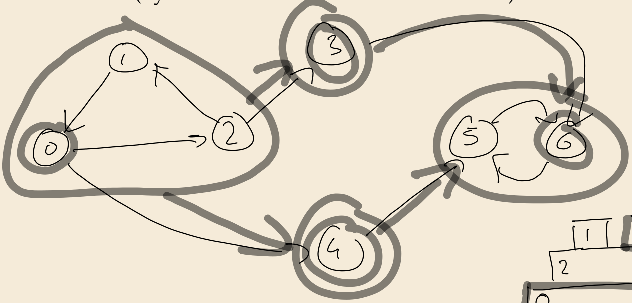
If we call `dfs` on any v in the **last** SCC C, it will discover all vertices in C, and only those!
(any edges between components lead *into* C by topological order)
And we can iterate this backwards through any topological order to get all SCCs!

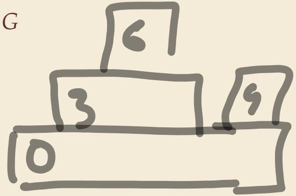
Can we efficiently find the topological order of G^{SCC}?
Without knowing the components to start with??

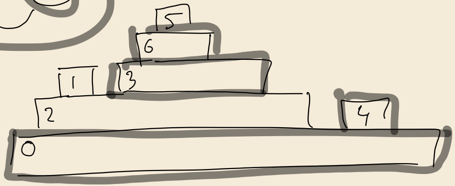
Amazingly, yes.

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

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

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!
- ϟ Reversing here would change v_i, i. e., which vertices of an SCC we see first

Kosaraju-Sharir's Algorithm

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

Kosaraju-Sharir's Algorithm

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

  If we reverse *iteration order* in dfsTraversal, we get `reverse`$($`topologicalSort`$(G))$ = `topologicalSort`(G^T)

Kosaraju-Sharir's Algorithm

- **Recall:** Want $\texttt{reverse}\big(\texttt{topologicalRanking}(G^{\mathrm{SCC}})\big)$
- **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: $\texttt{topologicalSort}(G^T)$

  If we reverse *iteration order* in dfsTraversal, we get $\texttt{reverse}(\texttt{topologicalSort}(G)) = \texttt{topologicalSort}(G^T)$
- Observation: $(G^T)^{\mathrm{SCC}} = (G^{\mathrm{SCC}})^T$
  - strong components not affected by edge reversals
- Want: $\texttt{reverse}\big(\texttt{topologicalRanking}(G^{\mathrm{SCC}})\big)$ (any ranking works, need not be reverse DFS postorder)

Kosaraju-Sharir's Algorithm

- **Recall:** Want $\texttt{reverse}\big(\texttt{topologicalRanking}(G^{\mathrm{SCC}})\big)$
- **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: $\texttt{topologicalSort}(G^T)$

  If we reverse *iteration order* in dfsTraversal, we get $\texttt{reverse}(\texttt{topologicalSort}(G)) = \texttt{topologicalSort}(G^T)$
- Observation: $\left(G^T\right)^{\mathrm{SCC}} = \left(G^{\mathrm{SCC}}\right)^T$
  - strong components not affected by edge reversals
- Want: $\texttt{reverse}\big(\texttt{topologicalRanking}(G^{\mathrm{SCC}})\big)$ (any ranking works, need not be reverse DFS postorder)

$\rightsquigarrow$ Get it from: $\texttt{topologicalRanking}\big((G^{\mathrm{SCC}})^T\big) = \texttt{topologicalRanking}\big((G^T)^{\mathrm{SCC}}\big)$

Kosaraju-Sharir's Algorithm

- **Recall:** Want $\texttt{reverse}\big(\texttt{topologicalRanking}(G^{\mathrm{SCC}})\big)$
- **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: $\texttt{topologicalSort}(G^T)$

  If we reverse *iteration order* in dfsTraversal, we get $\texttt{reverse}(\texttt{topologicalSort}(G)) = \texttt{topologicalSort}(G^T)$
- Observation: $\left(G^T\right)^{\mathrm{SCC}} = \left(G^{\mathrm{SCC}}\right)^T$
  - strong components not affected by edge reversals
- Want: $\texttt{reverse}\big(\texttt{topologicalRanking}(G^{\mathrm{SCC}})\big)$ (any ranking works, need not be reverse DFS postorder)

$\rightsquigarrow$ Get it from: $\texttt{topologicalRanking}\big((G^{\mathrm{SCC}})^T\big) = \texttt{topologicalRanking}\big((G^T)^{\mathrm{SCC}}\big)$

$\rightsquigarrow$ Get that as induced ranking on $v_1, \ldots, v_k$ from reverse $\texttt{dfsPostorder}(G^T)$

Kosaraju-Sharir's Algorithm – Code

```
procedure strongComponents(G):
    // directed graph G = (V, E) with V = [0..n)
    G^T = (V, {wv : vw ∈ E})
    P[0..n) := dfsPostorder(G^T) // postorder numbers
    for v ∈ V do S[P[v]] := v end for // postorder sequence
    // Rest like connectedComponents (with permuted vertices)
    C[0..n) := unseen
    SCC[0..n) := NONE
    id := 0
    for j := n − 1, . . . , 0 // reverse postorder seq
        v := S[j]
        if C[v] == unseen
            dfs(G, v)
            id := id + 1
    return SCC

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

Kosaraju-Sharir's Algorithm – Code

```
procedure strongComponents(G):
    // directed graph G = (V, E) with V = [0..n)
    G^T = (V, {wv : vw ∈ E})
    P[0..n) := dfsPostorder(G^T) // postorder numbers
    for v ∈ V do S[P[v]] := v end for // postorder sequence
    // Rest like connectedComponents (with permuted vertices)
    C[0..n) := unseen
    SCC[0..n) := NONE
    id := 0
    for j := n − 1, . . . , 0 // reverse postorder seq
        v := S[j]
        if C[v] == unseen
            dfs(G, v)
            id := id + 1
    return SCC

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

- correctness follows from our discussion

Kosaraju-Sharir's Algorithm – Code

```
procedure strongComponents(G):
    // directed graph G = (V, E) with V = [0..n)
    G^T = (V, {wv : vw ∈ E})
    P[0..n) := dfsPostorder(G^T) // postorder numbers
    for v ∈ V do S[P[v]] := v end for // postorder sequence
    // Rest like connectedComponents (with permuted vertices)
    C[0..n) := unseen
    SCC[0..n) := NONE
    id := 0
    for j := n − 1, . . . , 0 // reverse postorder seq
        v := S[j]
        if C[v] == unseen
            dfs(G, v)
            id := id + 1
    return SCC

procedure preorderVisit(v):
    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
  - ⇝ output in (forward) topological order
  - but derivation more natural this way?

Kosaraju-Sharir's Algorithm – Code

```
procedure strongComponents(G):
    // directed graph G = (V, E) with V = [0..n)
    G^T = (V, {wv : vw ∈ E})
    P[0..n) := dfsPostorder(G^T) // postorder numbers
    for v ∈ V do S[P[v]] := v end for // postorder sequence
    // Rest like connectedComponents (with permuted vertices)
    C[0..n) := unseen
    SCC[0..n) := NONE
    id := 0
    for j := n − 1, . . . , 0 // reverse postorder seq
        v := S[j]
        if C[v] == unseen
            dfs(G, v)
            id := id + 1
    return SCC

procedure preorderVisit(v):
    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
    - ⇝ output in (forward) topological order
    - but derivation more natural this way?
- as all our traversals: $\Theta(n + m)$ time, $\Theta(n)$ extra space

9.6 Network flows

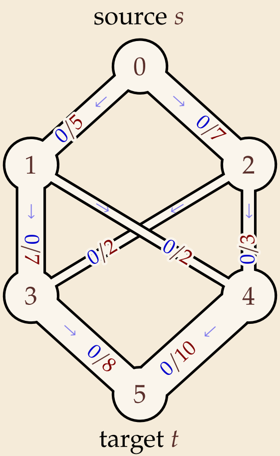
Clicker Question

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

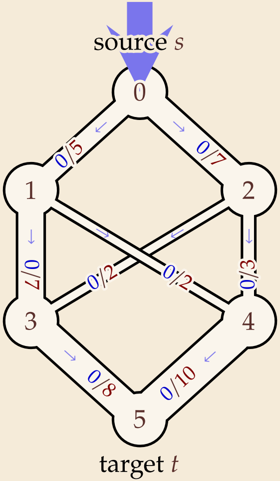


Informally, imagine a network of water pipes.

- 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

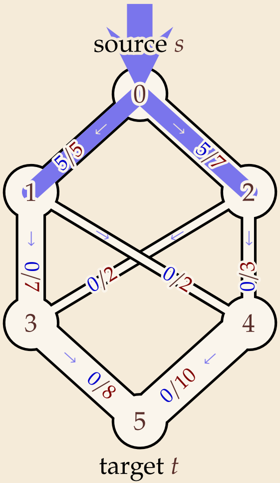


Informally, imagine a network of water pipes.

- 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

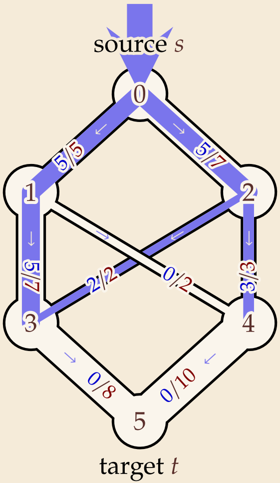


Informally, imagine a network of water pipes.

- 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

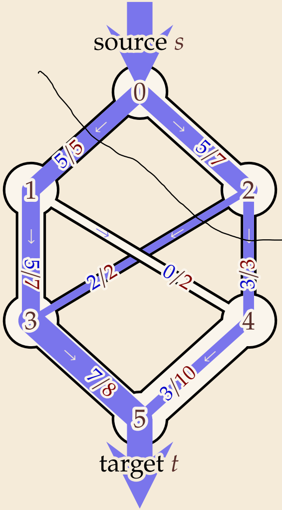


Informally, imagine a network of water pipes.

- 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)

$\rightsquigarrow$ How much water can flow through the network?

Networks and Flows – Informal

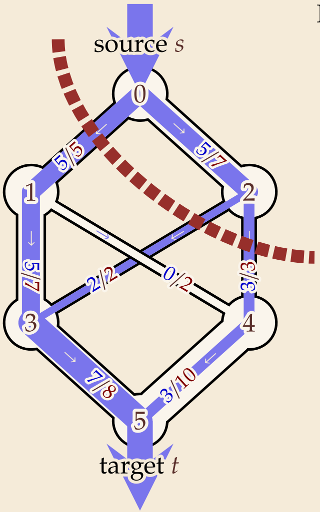


Informally, imagine a network of water pipes.

- 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)

$\rightsquigarrow$ How much water can flow through the network?

Networks and Flows – Informal



Informally, imagine a network of water pipes.

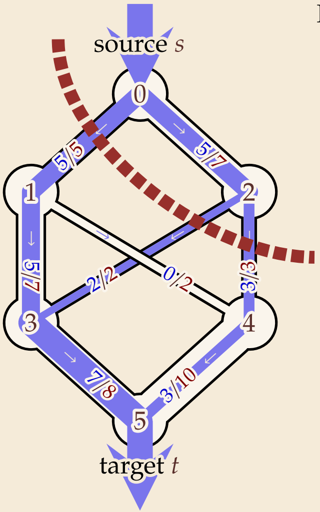
- Water can flow through the pipes up to a flow capacity limit (up to $c(e)$ liters per second, say).
- There's infinite water pressing into the source s and infinite drain capacity at the sink / target t
- At all other junctions, inflow = outflow (no leakage)

$\rightsquigarrow$ 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
- $\rightsquigarrow$ 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)

$\rightsquigarrow$ 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

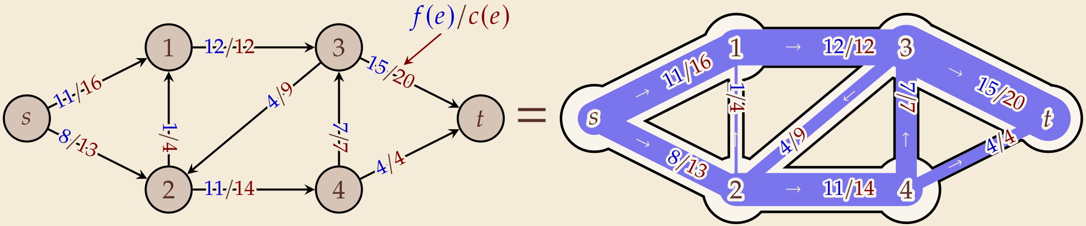
$\rightsquigarrow$ not more than 10 units out of s possible

$\rightsquigarrow$ 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 $G = (V, E)$, no antiparallel edges ($vw \in E \leadsto wv \notin E$)
    for notational convenience only
  - *edge capacities* $c : E \to \mathbb{R}_{\geq 0}$
  - distinguished vertices: *source* $s \in V$, target/*sink* $t \in V$

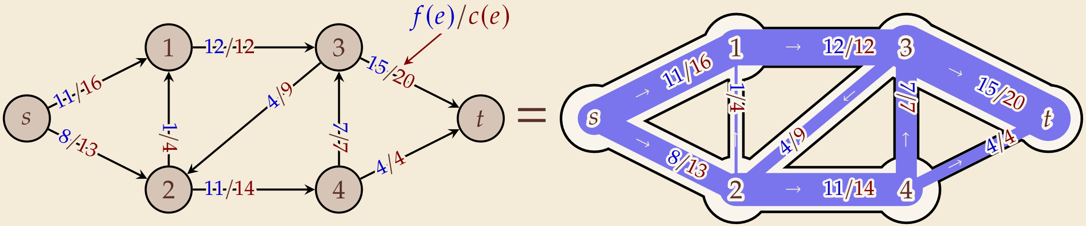


Networks and Flows – Definitions

- *s-t-(flow) network*:
  - **simple**, **directed**, **connected** graph $G = (V, E)$, no antiparallel edges ($vw \in E \rightsquigarrow wv \notin E$)

    for notational convenience only
  - *edge capacities* $c : E \to \mathbb{R}_{\geq 0}$
  - distinguished vertices: *source* $s \in V$, target/*sink* $t \in V$
- *(network) flow (in G)*: $f : E \to \mathbb{R}_{\geq 0}$
- 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:* $\forall v \in V \setminus \{s, t\} \ : \ \sum_{w \in V} f(w, v) = \sum_{w \in V} f(v, w)$

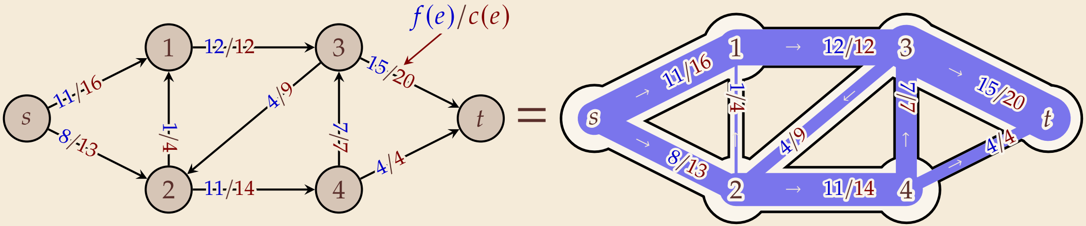


Networks and Flows – Definitions

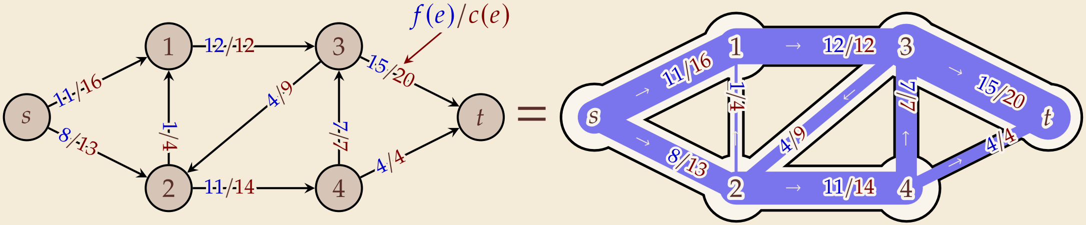
- *s-t-(flow) network*:
  - **simple**, **directed**, **connected** graph $G = (V, E)$, no antiparallel edges ($vw \in E \rightsquigarrow wv \notin E$)

    for notational convenience only
  - *edge capacities* $c : E \to \mathbb{R}_{\geq 0}$
  - distinguished vertices: *source* $s \in V$, target/*sink* $t \in V$
- *(network) flow (in G)*: $f : E \to \mathbb{R}_{\geq 0}$
- 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: $|f| = \underline{\sum_{v \in V} f(s, v)} - \sum_{v \in V} f(v, s)$

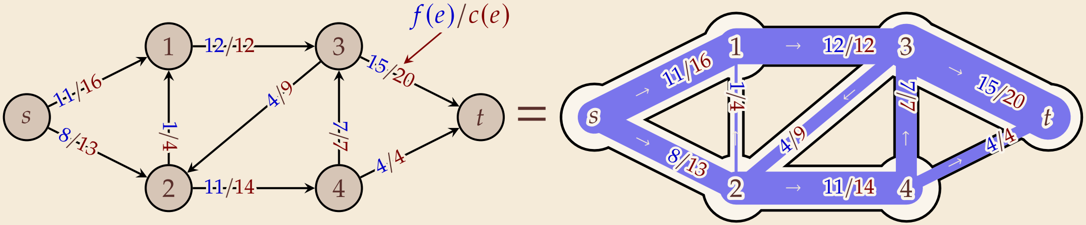


Max-Flow Problem



- **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 $\rightsquigarrow$ can restrict ourselves to integral flows as we will see
  - 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

  $\leadsto$ same maximum $|f|$

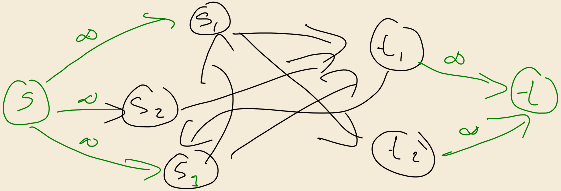
Multiple Sources & Sinks, Antiparallel Edges

- 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

  $\rightsquigarrow$ 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

  $\rightsquigarrow$ same maximum $|f|$



Reductions

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

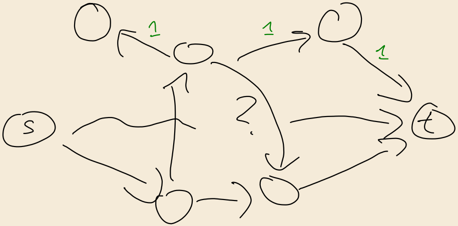
Reductions

- 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?

$c(e) = 1$

$|f^*| \in \mathbb{N}_0$

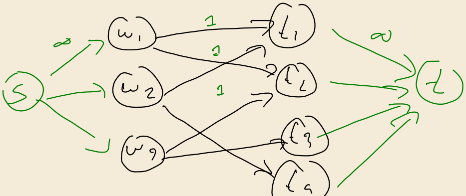


Reductions

- 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 $W = \{w_1, \ldots, w_k\}$ tasks $T = \{t_1, \ldots, t_\ell\}$, *qualified-for* relation $Q \subseteq W \times T$
   - **Goal:** Assignment $a : W \to T \cup \{\perp\}$ 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



$f^*(e) \in \mathbb{N}_0$

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

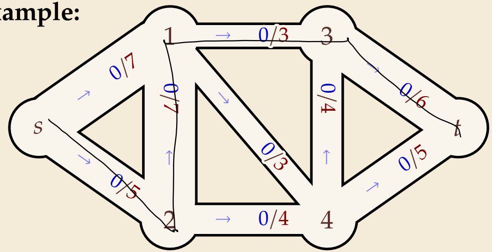
- Both problems can be solved by (in both cases, 1. and 3. are very efficient)
  1. constructing a specific flow network from their input data
  2. computing a maximum flow in that network
  3. "reading off" a solution for the orignal problem from the max flow

9.7 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:



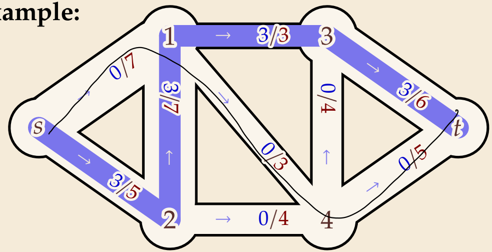
1. Push 3 units of flow over $s \to 2 \to 1 \to 3 \to t$

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

Push Push Push!

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

Example:



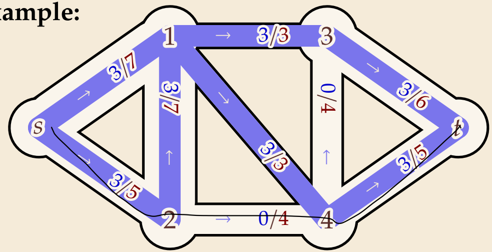
1. Push 3 units of flow over $s \to 2 \to 1 \to 3 \to t$
2. Push 3 units of flow over $s \to 1 \to 4 \to t$

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

Push Push Push!

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

Example:



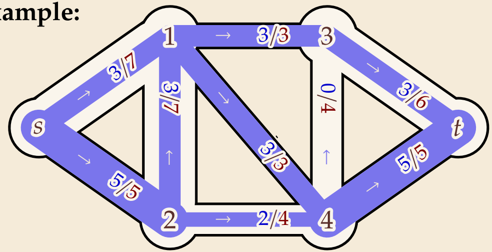
1. Push 3 units of flow over $s \to 2 \to 1 \to 3 \to t$
2. Push 3 units of flow over $s \to 1 \to 4 \to t$
3. Push 2 units of flow over $s \to 2 \to 4 \to t$

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

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

$\rightsquigarrow$ Every s-t path now has a saturated edge.

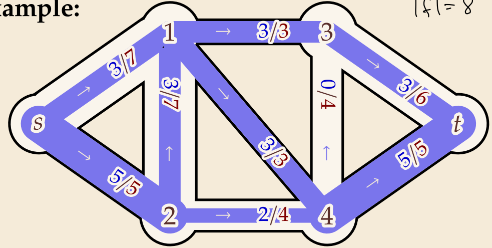
Problem: Cannot undo mistakes.
Here: shouldn't have put so much flow on $(1, 2)$ ...

Push Push Push!?

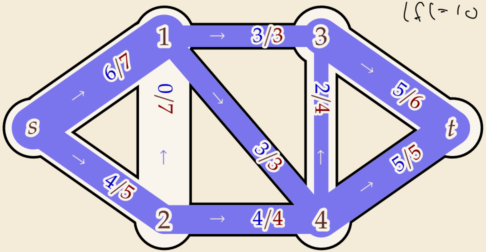
- **Simple Idea:** Iteratively find a path from s to t that we can push more flow over.

Example:

$|f| = 8$



$|f| = 10$



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

$\rightsquigarrow$ Every s-t path now has a saturated edge.

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

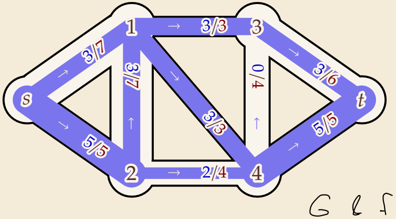
Problem: Cannot undo mistakes.
Here: shouldn't have put so much flow on $(1,2)$ …

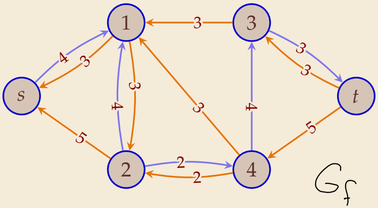
Residual Networks

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

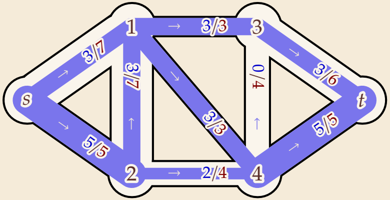


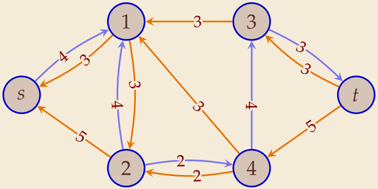


Residual Networks

- 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 $c_f(vw) = \begin{cases} c(vw) - f(vw) & vw \in E \quad \textit{// add flow} \\ f(wv) & wv \in E \quad \textit{// revert flow} \\ 0 & \text{else} \end{cases}$

    $E_f = \{vw : c_f(vw) > 0\}$





- *residual flow f':* feasible flow in G_f

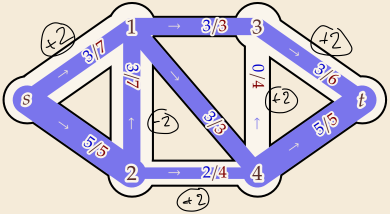
  $(f + f')(vw) = f(vw) + f'(vw) - f'(wv)$

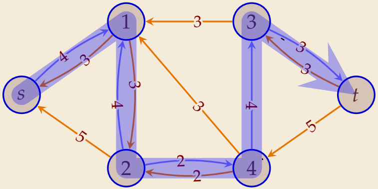
  $\rightsquigarrow$ for any f and residual flow f' in G_f, flow $f + f'$ is a feasible flow in G

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 $c_f(vw) = \begin{cases} c(vw) - f(vw) & vw \in E \quad \textit{// add flow} \\ f(wv) & wv \in E \quad \textit{// revert flow} \\ 0 & \text{else} \end{cases}$

    $E_f = \{vw : c_f(vw) > 0\}$





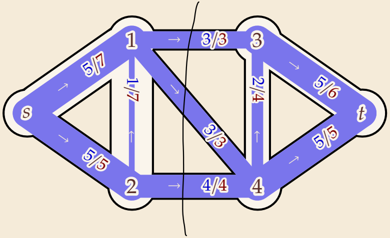
- *residual flow* f'*:* feasible flow in G_f

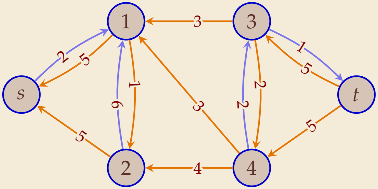
  $(f + f')(vw) = f(vw) + f'(vw) - f'(wv)$

  $\rightsquigarrow$ 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'!

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





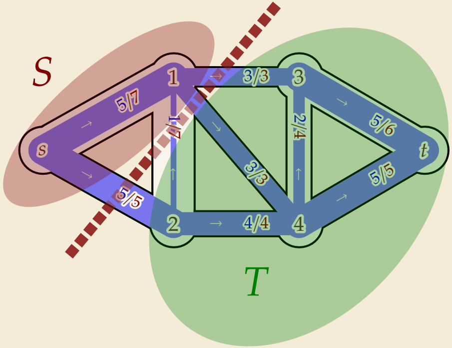
- *residual flow* f'*:* feasible flow in G_f
  $(f + f')(vw) = f(vw) + f'(vw) - f'(wv)$
  $\rightsquigarrow$ for any f and residual flow f' in G_f, flow $f + f'$ is a feasible flow in G
- *augmenting path* p*:* s-t-path G_f particularly simple f'!

Cuts

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

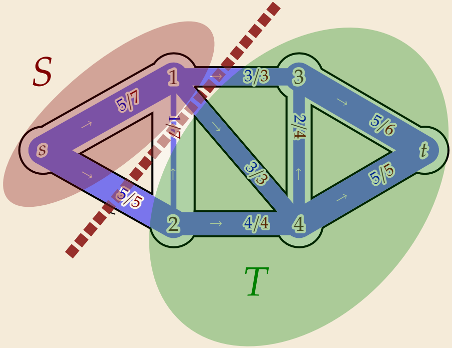
  disjoint union

  - *net flow* across cut:
    $$f(S,T) = \sum_{v \in S} \sum_{w \in T} \big(f(vw) - f(wv)\big)$$
  - *capacity* of cut:
    $$c(S,T) = \sum_{v \in S} \sum_{w \in T} f(vw)$$



Cuts

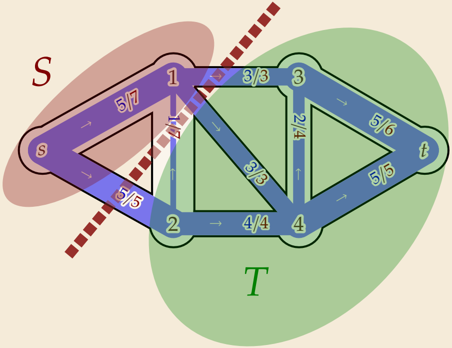
- **Goal:** Certificate for maximum flows
- *s-t-cut* (S,T): partition $S \mathbin{\dot\cup} T = V$, $s \in S$, $t \in T$
  - *net flow* across cut:
    $$f(S,T) = \sum_{v\in S}\sum_{w\in T}\bigl(f(vw) - f(wv)\bigr)$$
  - *capacity* of cut:
    $$c(S,T) = \sum_{v\in S}\sum_{w\in T} f(vw)$$



- $f(S,T) = 5 + 3 + 3 - 1 = 10$
- $c(S,T) = 5 + 3 + 3 = 11$

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:
    $$f(S,T) = \sum_{v \in S} \sum_{w \in T} \big(f(vw) - f(wv)\big)$$
  - *capacity* of cut:
    $$c(S,T) = \sum_{v \in S} \sum_{w \in T} f(vw)$$



- $f(S,T) = 5 + 3 + 3 - 1 = 10$
- $c(S,T) = 5 + 3 + 3 = 11$

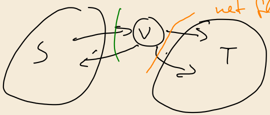
- **Lemma:** For any cut (S,T), we have $f(S,T) = |f|$.
  (flow conservation!)

Def. $|S| = f(\{s\}, V \setminus \{s\})$ induction on $|S|$

IS: add v to S

net flow from S

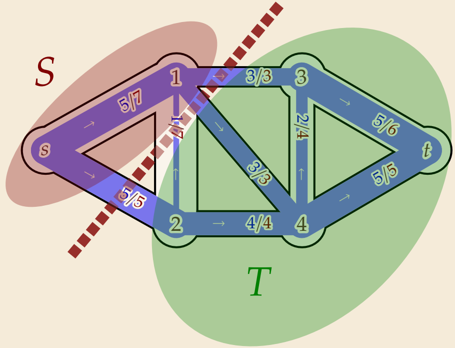
net flow to T



are equal by flow conservation

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:
    $$f(S,T) = \sum_{v\in S} \sum_{w\in T} \big(f(vw) - f(wv)\big)$$
  - *capacity* of cut:
    $$c(S,T) = \sum_{v\in S} \sum_{w\in T} f(vw)$$



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

  ∧|
- $c(S,T) = 5 + 3 + 3 = 11$

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

  (flow conservation!)
- **Corollary:** $|f| \le c(S,T)$ for any s-t-cut (S,T)

  ||

  f(S,T)