Graph Algorithms

Zbyněk Křivka
krivka@fit.vut.cz
Brno University of Technology
Faculty of Information Technology
Czech Republic

Outline

Introduction

References

Books

- Cormen, Leiserson, Rivest, Stein: *Introduction to algorithms.* The MIT Press and McGraw-Hill, 2001.
- Gibbons: *Algorithmic Graph Theory.* Cambridge University Press, 1985.

References

Books

- Cormen, Leiserson, Rivest, Stein: *Introduction to algorithms.* The MIT Press and McGraw-Hill, 2001.
- Gibbons: *Algorithmic Graph Theory.* Cambridge University Press, 1985.

Materials

- Lecture slides @ `https://www.fit.vutbr.cz/study/courses/GALe/public/`
- Text generated from lecture slides

Course Details

- lectures (2/3 + 0/1) – Zbyněk Křivka
- project (25 points) – Ľubica Genčúrová
- midterm test (15 points) – approx. middle of semester
- exam (60 points) — 3 terms, minimum 25 points
- consultations – `krivka@fit.vut.cz`, `igencurova@fit.vut.cz`

Course Details

- lectures (2/3 + 0/1) – Zbyněk Křivka
- project (25 points) – Ľubica Genčúrová
- midterm test (15 points) – approx. middle of semester
- exam (60 points) — 3 terms, minimum 25 points
- consultations – `krivka@fit.vut.cz`, `igencurova@fit.vut.cz`

About the Project

- individual
- implementation of two/more graph algorithms, experiments, comparison
- own assignment (suggestion of algorithms related to your thesis)
- presentation of your solutions during the last lecture
- implementation programming language - C/C++, Java, Python, Ruby (anything available at Merlin server or agreed by the teacher)

Algorithms and Complexity

Basic Notions

- Informally, algorithm is a well-defined procedure (sequence of computational steps) that transforms some input into the corresponding output.
- Data structure is a way of storage and organization of data optimized for access and/or modification.

Requirements on Algorithms

- Finiteness: Algorithm always ends for a valid (correct) input.
- Soundness, Correctness: The result is correct as well.

- Memory and time are limited!
- There is many solutions, we focus on the effective ones.

Algorithm Complexity

Time complexity of algorithm:

- Running time $T(n)$ – function giving the maximum number of "primitive" steps depending on the size of an input n, i.e. number of steps in the worst case.

Space complexity of algorithm:

- Memory consumption $S(n)$ – function giving the maximum number of used memory cells during the computation depending on the size of an input n. (including algorithm initialization or not?)

In general, n can be a vector (multidimensional).

Θ-notation

Let $g(n)$ be a function. Let $f(n)$ denote, for instance, $T(n)$ or $S(n)$.

- $\Theta(g(n)) = \{f(n):$ there exist $c_1, c_2, n_0 > 0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.
- $\Theta(g(n))$ is a family of functions that can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.
- Sometimes written as $f(n) = \Theta(g(n))$ instead $f(n) \in \Theta(g(n))$.
- We say that $g(n)$ is an asymptotically tight bound for $f(n)$.

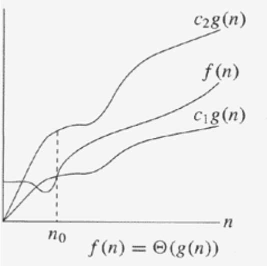


Figure: Θ-notation.

- $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ – verify its properties for $c_1 = \frac{1}{14}, c_2 = \frac{1}{2}, n_0 = 7$.

O-notation

Let $g(n)$ be a function.

- $O(g(n)) = \{f(n) :$ there exist $c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.
- $O(g(n))$ is a family of functions $f(n)$ such that $f(n)$'s value is on or below $cg(n)$ for all $n \geq n_0$.
- $f(n) = O(g(n))$ means some $cg(n)$ is an asymptotic upper bound on $f(n)$ (but not necessarily tight $\approx$ worst-case scenario).

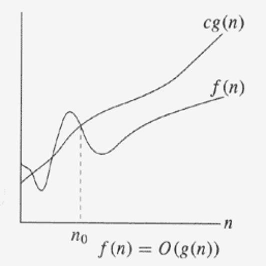


Figure: O-notation.

- $\Theta(g(n)) \subseteq O(g(n))$.
- $n = O(n^2)$, but $n \neq \Theta(n^2)$.

Ω-notation

Let $g(n)$ be a function.

- $\Omega(g(n)) = \{f(n) :$ there exist $c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.
- $\Omega(g(n))$ is a family of functions $f(n)$ such that $f(n)$'s value is on or above $cg(n)$ for all $n \geq n_0$.
- $f(n) = \Omega(g(n))$ means some $cg(n)$ is an asymptotic lower bound on $f(n)$ (but not necessarily tight $\approx$ best-case scenario).

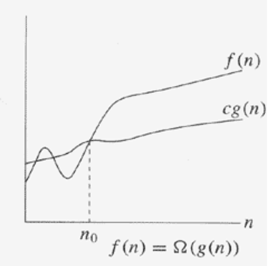


Figure: Ω-notation.

Theorem 1.
For any $f(n)$, $g(n)$, it holds $f(n) = \Theta(g(n))$ if and only if (iff) $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

o-notation and ω-notation

Let $g(n)$ be a function.

- $o(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.
  - upper bound that is NOT asymptotically tight

o-notation and ω-notation

Let $g(n)$ be a function.

- $o(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.
  - upper bound that is NOT asymptotically tight
- $\omega(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.
  - lower bound that is NOT asymptotically tight

o-notation and ω-notation

Let $g(n)$ be a function.

- $o(g(n)) = \{f(n) : \text{ for every } c > 0 \text{ there exist } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\}.$
  - upper bound that is NOT asymptotically tight
- $\omega(g(n)) = \{f(n) : \text{ for every } c > 0 \text{ there exist } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\}.$
  - lower bound that is NOT asymptotically tight
- $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$.

o-notation and ω-notation

Let $g(n)$ be a function.

- $o(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.
  - upper bound that is NOT asymptotically tight
- $\omega(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.
  - lower bound that is NOT asymptotically tight
- $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$.

- $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.
- $f(n) = o(g(n))$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

o-notation and ω-notation

Let $g(n)$ be a function.

- $o(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.
  - upper bound that is NOT asymptotically tight
- $\omega(g(n)) = \{f(n) :$ for every $c > 0$ there exist $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.
  - lower bound that is NOT asymptotically tight
- $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$.

- $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.
- $f(n) = o(g(n))$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

- $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$.
- $f(n) = \omega(g(n))$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Properties

Let $f(n)$, $g(n)$, and $h(n)$ be (asymptotically positive) functions.

- Transitivity
  $f(n) = X(g(n))$ and $g(n) = X(h(n))$ imply $f(n) = X(h(n))$, for $X \in \{\Theta, O, \Omega, o, \omega\}$.

Properties

Let $f(n)$, $g(n)$, and $h(n)$ be (asymptotically positive) functions.

- Transitivity
  $f(n) = X(g(n))$ and $g(n) = X(h(n))$ imply $f(n) = X(h(n))$, for $X \in \{\Theta, O, \Omega, o, \omega\}$.
- Reflexivity
  $f(n) = X(f(n))$, for $X \in \{\Theta, O, \Omega\}$.

Properties

Let $f(n)$, $g(n)$, and $h(n)$ be (asymptotically positive) functions.

- Transitivity
  $f(n) = X(g(n))$ and $g(n) = X(h(n))$ imply $f(n) = X(h(n))$, for $X \in \{\Theta, O, \Omega, o, \omega\}$.
- Reflexivity
  $f(n) = X(f(n))$, for $X \in \{\Theta, O, \Omega\}$.
- Symmetry
  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

Properties

Let $f(n)$, $g(n)$, and $h(n)$ be (asymptotically positive) functions.

- Transitivity
  $f(n) = X(g(n))$ and $g(n) = X(h(n))$ imply $f(n) = X(h(n))$,
  for $X \in \{\Theta, O, \Omega, o, \omega\}$.
- Reflexivity
  $f(n) = X(f(n))$, for $X \in \{\Theta, O, \Omega\}$.
- Symmetry
  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.
- Transpose symmetry
  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
  $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.

Properties

Let $f(n)$, $g(n)$, and $h(n)$ be (asymptotically positive) functions.

- Transitivity
  $f(n) = X(g(n))$ and $g(n) = X(h(n))$ imply $f(n) = X(h(n))$, for $X \in \{\Theta, O, \Omega, o, \omega\}$.
- Reflexivity
  $f(n) = X(f(n))$, for $X \in \{\Theta, O, \Omega\}$.
- Symmetry
  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.
- Transpose symmetry
  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
  $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.
- Not always comparable
  n and $n^{1+\sin(n)}$ are incomparable.

Graphs

Graph Theory: The Beginning

- Leonhard Euler, *The Königsberg bridges problem*, 1736.
- Problem: Is it possible to cross all bridges, but everyone just once?
- `https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg`

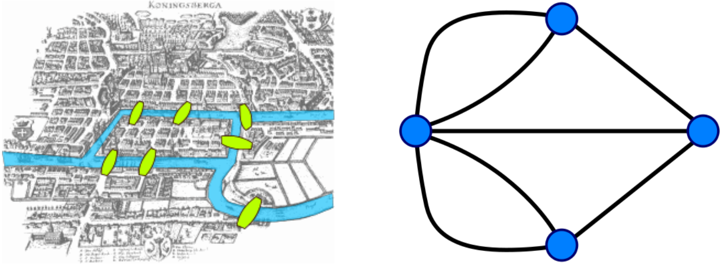


Figure: Map of bridges and its logical representation.

Definitions

Directed graph (digraph) G is a pair

$$G = (V, E),$$

where

- V is a finite set of vertices (nodes) and
- $E \subseteq V^2$ is a set of edges (arrows, arcs).

An edge (u, u) is called a self-loop.
If (u, v) is an edge, we say that (u, v) is incident from u and incident to v, that is v is adjacent to u.

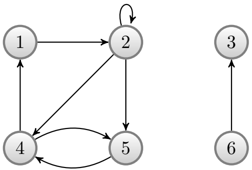


Figure: Digraph

A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$, if

- $V' \subseteq V$ and $E' \subseteq E$.

Let $V'' \subseteq V$. Subgraph induced by V'' is graph $G'' = (V'', E'')$, where

- $E'' = \{(u, v) \in E : u, v \in V''\}$.

Let $E''' \subseteq E$. Factor subgraph of G is graph $G''' = (V, E''')$.

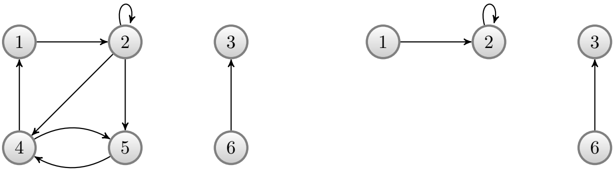


Figure: A graph and its subgraph induced by $\{1, 2, 3, 6\}$.

Definitions

Undirected graph G is a pair

$$G = (V, E),$$

where

- V is a finite set of vertices and
- $E \subseteq \binom{V}{2}$ is a set of edges.

Note

An edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. Self-loops are forbidden.

Convention: $\{u, v\}$, (u, v), and (v, u) denote the same edge.

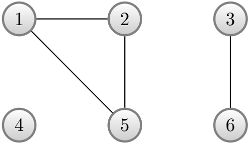


Figure: Undirected Graph

- Degree of vertex u in an undirected graph is the number of adjacent vertices, denoted by $d(u)$.
- $d(1) = d(2) = d(5) = 2$, $d(3) = d(6) = 1$, $d(4) = 0$.
- If $d(u) = 0$, u is called isolated vertex.

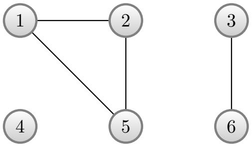


Figure: Undirected graph

- Out-degree of vertex u is the number of outcoming edges, denoted as $deg_-(u)$.
- In-degree of vertex u is the number of incoming edges, denoted as $deg_+(u)$.
- Degree of vertex u is the sum of its in-degree and out-degree, denoted as $deg(u)$.
- $deg_-(2) = 3$, $deg_+(2) = 2$, $deg(2) = 5$.

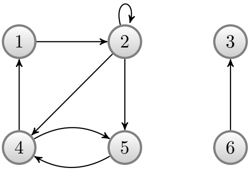


Figure: Digraph

Definitions

- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.

Definitions

- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.
- The length of p equals to the number of edges in p.

Definitions

- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.
- The length of p equals to the number of edges in p.
- If the length is 0, we consider a trivial path from u to u by following no edge (for every vertex u).

Definitions

- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.
- The length of p equals to the number of edges in p.
- If the length is 0, we consider a trivial path from u to u by following no edge (for every vertex u).
- If there is p from u to u', we say that u' is reachable from u by p, denoted as $u \overset{p}{\leadsto} u'$.

Definitions

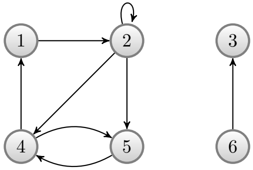
- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.
- The length of p equals to the number of edges in p.
- If the length is 0, we consider a trivial path from u to u by following no edge (for every vertex u).
- If there is p from u to u', we say that u' is reachable from u by p, denoted as $u \overset{p}{\leadsto} u'$.
- A path is tour if all edges in the path are distinct.

Definitions

- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.
- The length of p equals to the number of edges in p.
- If the length is 0, we consider a trivial path from u to u by following no edge (for every vertex u).
- If there is p from u to u', we say that u' is reachable from u by p, denoted as $u \overset{p}{\rightsquigarrow} u'$.
- A path is tour if all edges in the path are distinct.
- A path is simple if all vertices in the path are distinct.

Definitions

- A path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a connected sequence of vertices where $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$.
- The length of p equals to the number of edges in p.
- If the length is 0, we consider a trivial path from u to u by following no edge (for every vertex u).
- If there is p from u to u', we say that u' is reachable from u by p, denoted as $u \overset{p}{\leadsto} u'$.
- A path is tour if all edges in the path are distinct.
- A path is simple if all vertices in the path are distinct.



- Give some examples of a path and simple path.
- Give an example of unconnected sequence.

Definitions

- A subpath s of $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a contiguous subsequence, $s = \langle v_i, v_{i+1}, v_{i+2}, \ldots, v_j \rangle$, for $0 \leq i \leq j \leq k$.

Definitions

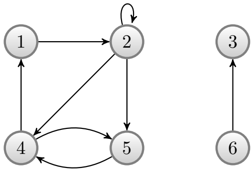
- A subpath s of $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a contiguous subsequence, $s = \langle v_i, v_{i+1}, v_{i+2}, \ldots, v_j \rangle$, for $0 \leq i \leq j \leq k$.

- A path $c = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a cycle (closed path), if $k \geq 1$ and $v_0 = v_k$.
- For undirected graph, let $k \geq 3$.

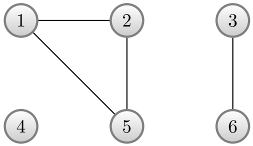
Definitions

- A subpath s of $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a contiguous subsequence, $s = \langle v_i, v_{i+1}, v_{i+2}, \ldots, v_j \rangle$, for $0 \leq i \leq j \leq k$.

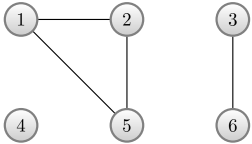
- A path $c = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a cycle (closed path), if $k \geq 1$ and $v_0 = v_k$.
- For undirected graph, let $k \geq 3$.
- Closed simple path is called simple cycle.



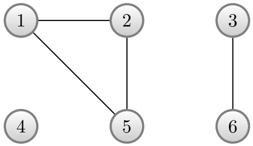
- What is $\langle 1, 2, 4, 5, 4, 1 \rangle$?
- What is $\langle 1, 2, 4, 1 \rangle$?
- What is $\langle 2, 2 \rangle$?



- $\langle 1, 2, 5, 1 \rangle$ is an undirected cycle.
- $\langle 3, 6, 3 \rangle$ is not a cycle



- $\langle 1, 2, 5, 1 \rangle$ is an undirected cycle.
- $\langle 3, 6, 3 \rangle$ is not a cycle , or is it?



- $\langle 1, 2, 5, 1 \rangle$ is an undirected cycle.
- $\langle 3, 6, 3 \rangle$ is not a cycle , or is it?

- A digraph with no self-loops is simple.
- Acyclic graph contains no cycles.

Special Cases of Graphs

Let $G = (V, E)$ be a graph with n vertices.

- Isolated graph Φ_n: $E = \emptyset$. (Null graph if even $V = \emptyset$.)

Special Cases of Graphs

Let $G = (V, E)$ be a graph with n vertices.

- Isolated graph Φ_n: $E = \emptyset$. (Null graph if even $V = \emptyset$.)
- Complete graph K_n: $E = \binom{V}{2}$.

Special Cases of Graphs

Let $G = (V, E)$ be a graph with n vertices.

- Isolated graph Φ_n: $E = \emptyset$. (Null graph if even $V = \emptyset$.)
- Complete graph K_n: $E = \binom{V}{2}$.
- Regular graph: For every $u, v \in V$, $d(u) = d(v)$.

Special Cases of Graphs

Let $G = (V, E)$ be a graph with n vertices.

- Isolated graph Φ_n: $E = \emptyset$. (Null graph if even $V = \emptyset$.)
- Complete graph K_n: $E = \binom{V}{2}$.
- Regular graph: For every $u, v \in V$, $d(u) = d(v)$.
- Cycle graph: $n \geq 3$ and vertices are connected in a closed chain.

Tree, Forest

- An undirected graph is connected if every pair of vertices is connected by a path.
- An connected, acyclic, undirected graph is a tree.
  - Homework: Prove that $|E| = |V| - 1$.
- In a rooted tree, there is one special vertex called root (with no parents).
- An acyclic, undirected graph is a forest (several trees).

Bipartite Graph

- Let $G = (V, E)$ be a undirected graph.
- We call G bipartite if the vertex set V can be partitioned into
  $V = L \cup R$,

  where L and R are disjoint and all edges in E go between L and R.
- L and R are called parts (disjoint and independent sets).

Bipartite Graph

- Let $G = (V, E)$ be a undirected graph.
- We call G bipartite if the vertex set V can be partitioned into
  $V = L \cup R$,

  where L and R are disjoint and all edges in E go between L and R.
- L and R are called parts (disjoint and independent sets).
- Optional additional condition:

  Every vertex in V has at least one incident edge.

Bipartite Graph

- Let $G = (V, E)$ be a undirected graph.
- We call G bipartite if the vertex set V can be partitioned into
  $V = L \cup R$,

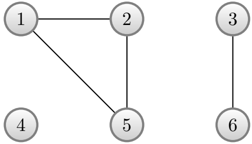
  where L and R are disjoint and all edges in E go between L and R.
- L and R are called parts (disjoint and independent sets).
- Optional additional condition:

  Every vertex in V has at least one incident edge.
- Complete bipartite graph $K_{m,n}$: $|L| = m$, $|R| = n$, and $|E| = mn$.

- Undirected graph is called connected, if there is a path between each pair of vertices.

- Undirected graph is called connected, if there is a path between each pair of vertices.
- Connected components of an undirected graph correspond to the equivalence classes by relation "is reachable from".

- Undirected graph is called connected, if there is a path between each pair of vertices.
- Connected components of an undirected graph correspond to the equivalence classes by relation "is reachable from".



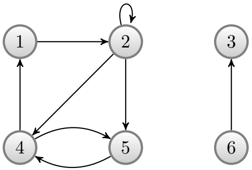
A graph with three connected components:

- $\{1, 2, 5\}$
- $\{3, 6\}$
- $\{4\}$

- Digraph is strongly connected, if there exists a path between each pair of vertices.

- Digraph is strongly connected, if there exists a path between each pair of vertices.
- Strongly connected components of graph are the equivalence classes of vertices according to the relation "mutually reachable".

- Digraph is strongly connected, if there exists a path between each pair of vertices.
- Strongly connected components of graph are the equivalence classes of vertices according to the relation "mutually reachable".



Graph has three strongly connected components:

- $\{1,2,4,5\}$
- $\{3\}$
- $\{6\}$

Graph Representation

Let $G = (V, E)$ be a graph. Denote:

- $n = |V|$
- $m = |E|$.

1. Adjacency-list representation
   - effective for sparse graphs ($m \ll n^2$);
   - we will use this representation in this talk.

Let $G = (V, E)$ be a graph. Denote:

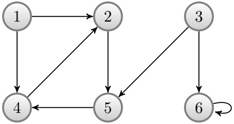
- $n = |V|$
- $m = |E|$.

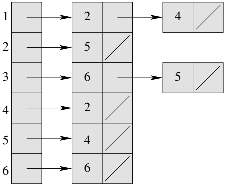
1. Adjacency-list representation
   - effective for sparse graphs ($m \ll n^2$);
   - we will use this representation in this talk.
2. Adjacency-matrix representation
   - effective for dense graphs (m close to n^2);
   - when we often need quick answer whether two given vertices are connected by an edge.

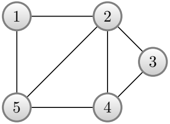
Adjacency-list representation

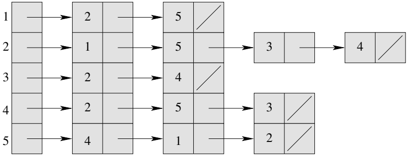
$G = (V, E)$ is represented as

- an array $Adj[1 \dots n]$ with n lists, one list for each vertex,
- where $Adj[u]$ stores all vertices v such that $(u, v) \in E$.









- Space complexity: $\Theta(m + n)$ (depends linearly on the size of the graph).

Weighted graph

- A weighted graph is a (di)graph where there is a value assigned to every edge using weight function $w : E \to \mathbb{R}$.

Weighted graph

- A weighted graph is a (di)graph where there is a value assigned to every edge using weight function $w : E \to \mathbb{R}$.

- Representation of $w(u, v)$ in adjacency list: extend the list item (a structure) for v in $Adj[u]$ with value $w(u, v)$.

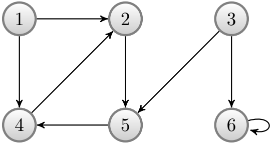
Weighted graph

- A weighted graph is a (di)graph where there is a value assigned to every edge using weight function $w : E \to \mathbb{R}$.
- Representation of $w(u, v)$ in adjacency list: extend the list item (a structure) for v in $Adj[u]$ with value $w(u, v)$.
- Disadvantage: Finding whether an edge (u, v) belongs to E requires the search of the whole list $Adj[u]$.

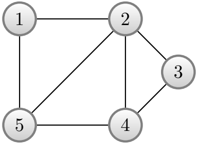
Adjacency-matrix representation

Let $G = (V, E)$ be a graph and assume $V = \{1, 2, \ldots, n\}$. Adjacency matrix $A = (a_{ij})$ is a matrix of size $n \times n$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

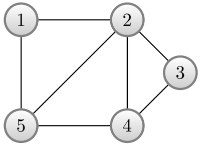


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1



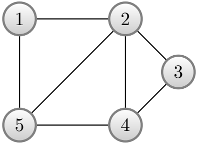
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Space complexity: $\Theta(n^2)$ (independent of the number of edges).



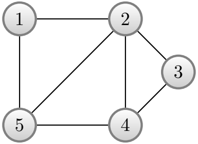
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Space complexity: $\Theta(n^2)$ (independent of the number of edges).
- Transpose matrix of $A = (a_{ij})$ is a matrix $A^T = (a^T_{ij})$, where $a^T_{ij} = a_{ji}$.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Space complexity: $\Theta(n^2)$ (independent of the number of edges).
- Transpose matrix of $A = (a_{ij})$ is a matrix $A^T = (a^T_{ij})$, where $a^T_{ij} = a_{ji}$.
- If A represents an undirected graph, then $A = A^T$. It is enough to store just one half of A.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Space complexity: $\Theta(n^2)$ (independent of the number of edges).
- Transpose matrix of $A = (a_{ij})$ is a matrix $A^T = (a^T_{ij})$, where $a^T_{ij} = a_{ji}$.
- If A represents an undirected graph, then $A = A^T$. It is enough to store just one half of A.
- Let $G = (V, E)$ be a weighted graph, then

$$a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E, \\ \text{NIL} & \text{otherwise,} \end{cases}$$

where NIL is a special value, mostly 0 or ∞.

Exercises

1. Given an adjacency-list representation of a directed graph and a vertex v, how long does it take to compute $deg_-(v)$ and $deg_+(v)$?
2. The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe an efficient algorithm for computing G^T from G for the adjacency-list representation of G. Analyze the time complexity of your algorithm.
3. The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only G contains a path with at most two edges between u and v. Describe an efficient algorithm for computing G^2 from G for the adjacency-list representation of G. Analyze the time complexity of your algorithm.

Breath-First Search

Breath-First Search (BFS)

- Input: (un)directed graph $G = (V, E)$ and a vertex $s \in V$.

Breath-First Search (BFS)

- Input: (un)directed graph $G = (V, E)$ and a vertex $s \in V$.
- Searches each vertex reachable from s and determines its distance (number of edges) from s.

Breath-First Search (BFS)

- Input: (un)directed graph $G = (V, E)$ and a vertex $s \in V$.
- Searches each vertex reachable from s and determines its distance (number of edges) from s.
- Creates BFS tree rooted at s containing all vertices reachable from s. $s \rightsquigarrow v$ is the shortest path in G.

Breath-First Search (BFS)

- Input: (un)directed graph $G = (V, E)$ and a vertex $s \in V$.
- Searches each vertex reachable from s and determines its distance (number of edges) from s.
- Creates BFS tree rooted at s containing all vertices reachable from s. $s \rightsquigarrow v$ is the shortest path in G.
- During the computation, BFS assigns a color representing a state to each vertex.

Breath-First Search (BFS)

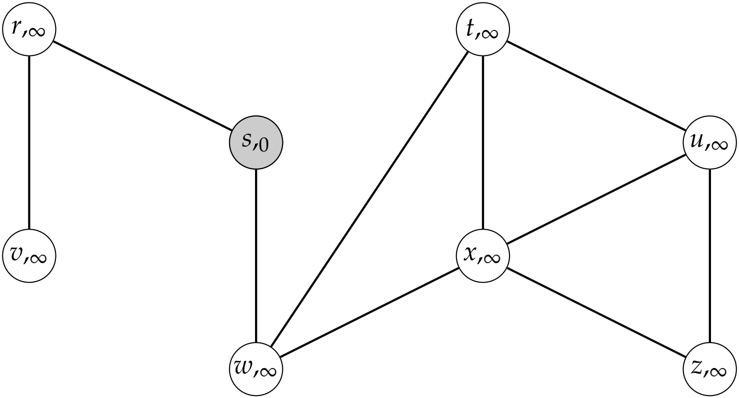
- Input: (un)directed graph $G = (V, E)$ and a vertex $s \in V$.
- Searches each vertex reachable from s and determines its distance (number of edges) from s.
- Creates BFS tree rooted at s containing all vertices reachable from s. $s \rightsquigarrow v$ is the shortest path in G.
- During the computation, BFS assigns a color representing a state to each vertex.
- Graph representation – Adjacency-list representation.
- $color[u] \in \{$WHITE, GREY, BLACK$\}$.

Breath-First Search (BFS)

- Input: (un)directed graph $G = (V, E)$ and a vertex $s \in V$.
- Searches each vertex reachable from s and determines its distance (number of edges) from s.
- Creates BFS tree rooted at s containing all vertices reachable from s. $s \rightsquigarrow v$ is the shortest path in G.
- During the computation, BFS assigns a color representing a state to each vertex.
- Graph representation – Adjacency-list representation.
- $color[u] \in \{$WHITE, GREY, BLACK$\}$.
- $\pi[u]$ denotes a predecessor of u at a path from s.
- $d[u]$ denotes a distance of u from s (the number of edges).

```
BFS(G, s)
for each vertex u ∈ V − {s}
     do color[u] ← WHITE
        d[u] ← ∞
        π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
     do u ← DEQUEUE(Q)
        for each v ∈ Adj[u]
            do if color[v] = WHITE
                  then color[v] ← GRAY
                       d[v] ← d[u] + 1
                       π[v] ← u
                       ENQUEUE(Q, v)
        color[u] ← BLACK
```

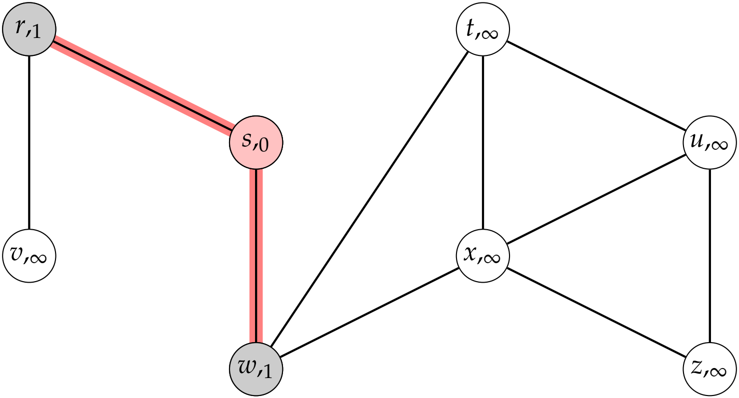
BFS – Example



$$Q = \langle s \rangle$$

Figure: Note: We use red color to show BLACK vertices.

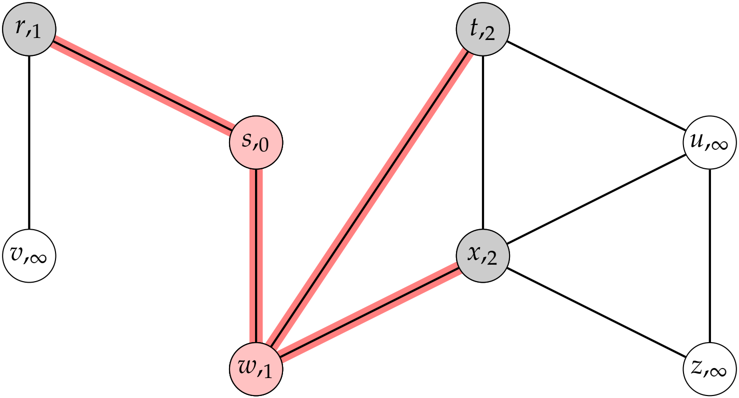
BFS – Example



$$Q = \langle wr \rangle$$

Figure: Note: We use red color to show BLACK vertices.

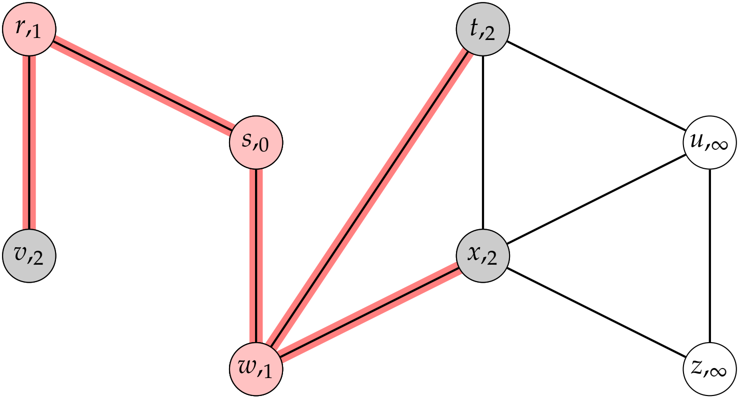
BFS – Example



$$Q = \langle rtx \rangle$$

Figure: Note: We use red color to show BLACK vertices.

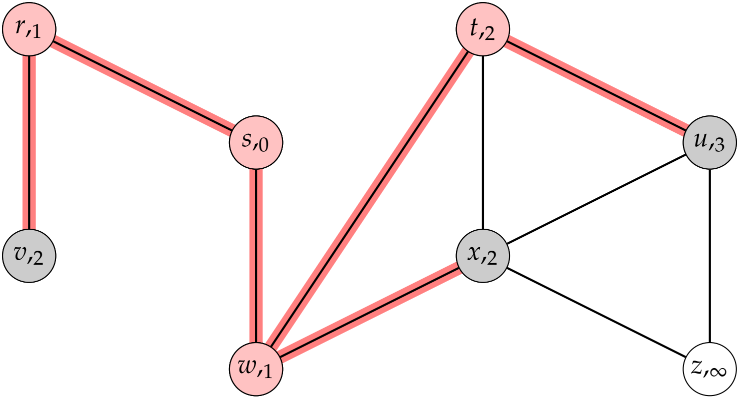
BFS – Example



$$Q = \langle txv \rangle$$

Figure: Note: We use red color to show BLACK vertices.

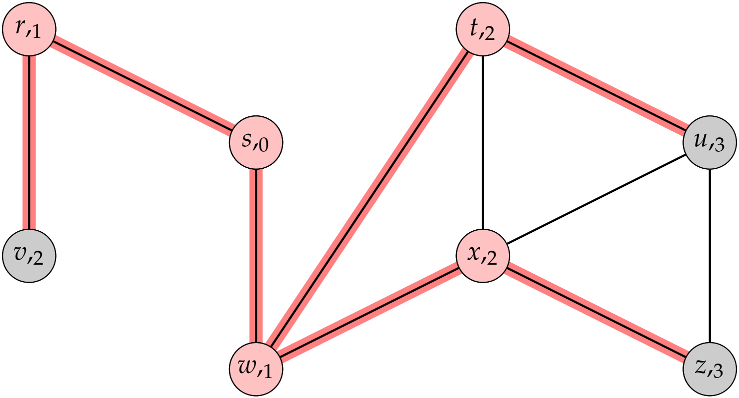
BFS – Example



$$Q = \langle xvu \rangle$$

Figure: Note: We use red color to show BLACK vertices.

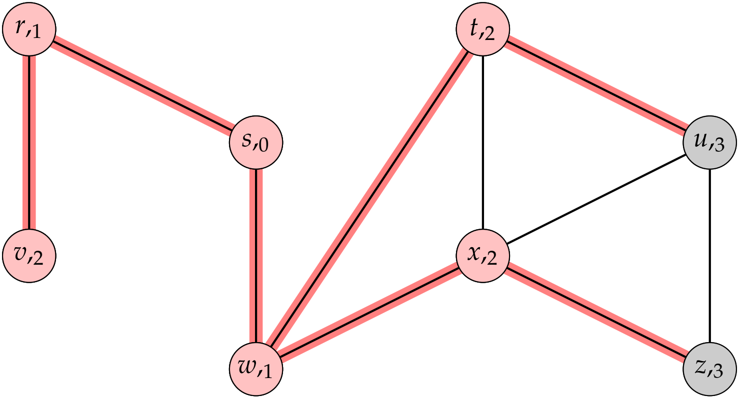
BFS – Example



$$Q = \langle vuz \rangle$$

Figure: Note: We use red color to show BLACK vertices.

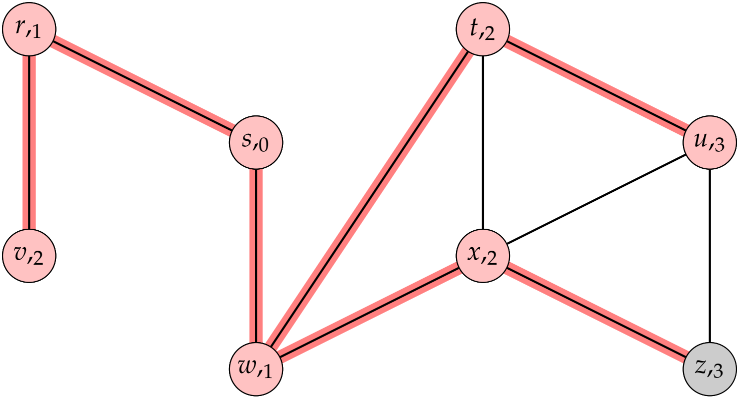
BFS – Example



$$Q = \langle uz \rangle$$

Figure: Note: We use red color to show BLACK vertices.

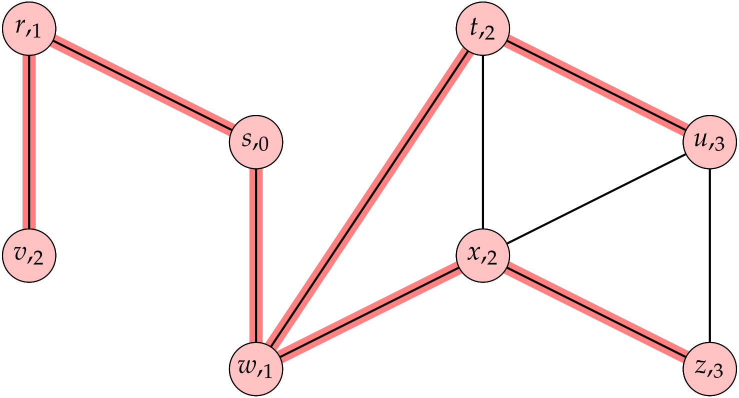
BFS – Example



$$Q = \langle z \rangle$$

Figure: Note: We use red color to show BLACK vertices.

BFS – Example



$$Q = \emptyset$$

Figure: Note: We use red color to show BLACK vertices.

Time Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V − {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if color[v] = WHITE
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       color[u] ← BLACK
```

- In **while**-loop no vertex is colored to $WHITE$.

Time Complexity of BFS

```
BFS(G, s)
1  for each vertex u ∈ V − {s}
2      do color[u] ← WHITE
3         d[u] ← ∞
4         π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     do u ← DEQUEUE(Q)
12        for each v ∈ Adj[u]
13            do if color[v] = WHITE
14                  then color[v] ← GRAY
15                       d[v] ← d[u] + 1
16                       π[v] ← u
17                       ENQUEUE(Q, v)
18        color[u] ← BLACK
```

- In **while**-loop no vertex is colored to $WHITE$.
- So line 13 guarantees that each vertex will be enqueued and then dequeued at most once.

Time Complexity of BFS

```
BFS(G, s)
1  for each vertex u ∈ V − {s}
2      do color[u] ← WHITE
3         d[u] ← ∞
4         π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     do u ← DEQUEUE(Q)
12        for each v ∈ Adj[u]
13            do if color[v] = WHITE
14                  then color[v] ← GRAY
15                       d[v] ← d[u] + 1
16                       π[v] ← u
17                       ENQUEUE(Q, v)
18        color[u] ← BLACK
```

- In **while**-loop no vertex is colored to $WHITE$.
- So line 13 guarantees that each vertex will be enqueued and then dequeued at most once.
- ENQUEUE and DEQUEUE takes $O(1)$, so the aggregation of all queue operations takes $O(n)$.

Time Complexity of BFS

```
BFS(G, s)
1  for each vertex u ∈ V − {s}
2      do color[u] ← WHITE
3         d[u] ← ∞
4         π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     do u ← DEQUEUE(Q)
12        for each v ∈ Adj[u]
13            do if color[v] = WHITE
14                  then color[v] ← GRAY
15                       d[v] ← d[u] + 1
16                       π[v] ← u
17                       ENQUEUE(Q, v)
18        color[u] ← BLACK
```

- In **while**-loop no vertex is colored to $WHITE$.
- So line 13 guarantees that each vertex will be enqueued and then dequeued at most once.
- ENQUEUE and DEQUEUE takes $O(1)$, so the aggregation of all queue operations takes $O(n)$.
- Since it scans the adjacency list of each vertex only after it is dequeued, each adjacency list is scanned at most once.

Time Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V − {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if color[v] = WHITE
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       color[u] ← BLACK
```

- Observe that the sum of the lengths of all the adjacency lists is $\Theta(m)$, the total time of scanning is $O(m)$.

Time Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V − {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if color[v] = WHITE
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       color[u] ← BLACK
```

- Observe that the sum of the lengths of all the adjacency lists is $\Theta(m)$, the total time of scanning is $O(m)$.
- The overhead for initialization is $O(n)$, so the total running time of BFS is $O(m+n)$. Thus, it is linear in the size of G (adjacency-list representation).

Shortest paths

- BFS finds the distance to each reachable vertex in G from a given source vertex $s \in V$. (No weight function yet)

Shortest paths

- BFS finds the distance to each reachable vertex in G from a given source vertex $s \in V$. (No weight function yet)
- Define the shortest-path distance $\delta(s, v)$ from s to v as the minimum number of edges in any path from s to v. If there is no path from s to v, then $\delta(s, v) = \infty$.

Shortest paths

- BFS finds the distance to each reachable vertex in G from a given source vertex $s \in V$. (No weight function yet)
- Define the shortest-path distance $\delta(s, v)$ from s to v as the minimum number of edges in any path from s to v. If there is no path from s to v, then $\delta(s, v) = \infty$.
- A path of length $\delta(s, v)$ from s to v is called a shortest path from s to v.

Lemma 2.

Let $G = (V, E)$ be a (di)graph and $s \in V$ be a vertex. Then, for every edge $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1 \, .$$

Proof.

- If vertex u is reachable from s, then vertex v is reachable from s as well. Therefore, the shortest path from s to v is no longer than a shortest path from s to u followed by edge (u, v). So inequality holds.

□

Lemma 2.

Let $G = (V, E)$ *be a (di)graph and* $s \in V$ *be a vertex. Then, for every edge* $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1 \, .$$

Proof.

- If vertex u is reachable from s, then vertex v is reachable from s as well. Therefore, the shortest path from s to v is no longer than a shortest path from s to u followed by edge (u, v). So inequality holds.
- If vertex u is not reachable from s, then $\delta(s, u) = \infty$ and, again, the inequality holds.

□

Lemma 3.

Let $G = (V, E)$ be a (di)graph and assume that BFS is executed on G from vertex $s \in V$. Then, when BFS finishes, then $d[v] \geq \delta(s, v)$ for every $v \in V$.

Proof.

- By induction on the number of ENQUEUE operations. Induction Hypothesis (IH): Assume that $d[v] \geq \delta(s, v)$ for every $v \in V$.

Lemma 3.

Let $G = (V, E)$ be a (di)graph and assume that BFS is executed on G from vertex $s \in V$. Then, when BFS finishes, then $d[v] \geq \delta(s, v)$ for every $v \in V$.

Proof.

- By induction on the number of ENQUEUE operations. Induction Hypothesis (IH): Assume that $d[v] \geq \delta(s, v)$ for every $v \in V$.
- Induction Basis (IB): $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$, $v \in V - \{s\}$.

Lemma 3.

Let $G = (V, E)$ be a (di)graph and assume that BFS is executed on G from vertex $s \in V$. Then, when BFS finishes, then $d[v] \geq \delta(s, v)$ for every $v \in V$.

Proof.

- By induction on the number of ENQUEUE operations. Induction Hypothesis (IH): Assume that $d[v] \geq \delta(s, v)$ for every $v \in V$.
- Induction Basis (IB): $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$, $v \in V - \{s\}$.
- Let v is WHITE vertex discovered during the exploration from u. By IH, we have $d[u] \geq \delta(s, u)$. By line 15 of BFS, IH, and the previous lemma,

  $$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v) .$$

  Since v is GREY now (and enqueued) and lines 14–17 are executed only for WHITE vertices, v cannot be enqueued again and its $d[v]$ value remains unchanged.

Lemma 4.

During the execution of BFS on $G = (V, E)$, *let queue* Q *contains vertices* $\langle v_1, v_2, \ldots, v_r \rangle$, *where* v_1 *is the front item of* Q *(leader) and* v_r *is the last item of* Q. *Then,* $d[v_r] \leq d[v_1] + 1$ *and* $d[v_i] \leq d[v_{i+1}]$ *for* $i = 1, 2, \ldots, r - 1$.

Proof.

- By induction on the number of queue operations. First, $Q = \langle s \rangle$, so lemma holds. It holds after execution of both queue operations:

□

Lemma 4.

During the execution of BFS on $G = (V, E)$, *let queue* Q *contains vertices* $\langle v_1, v_2, \ldots, v_r \rangle$, *where* v_1 *is the front item of* Q *(leader) and* v_r *is the last item of* Q. *Then,* $d[v_r] \leq d[v_1] + 1$ *and* $d[v_i] \leq d[v_{i+1}]$ *for* $i = 1, 2, \ldots, r-1$.

Proof.

- By induction on the number of queue operations. First, $Q = \langle s \rangle$, so lemma holds. It holds after execution of both queue operations:
- v_1 is removed so v_2 is new leader (if Q is emptied, it holds trivially). By IH, $d[v_1] \leq d[v_2]$. But then, $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$ and the rest of inequalities is unchanged.

□

Lemma 4.

During the execution of BFS on $G = (V, E)$, let queue Q contains vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the front item of Q (leader) and v_r is the last item of Q. Then, $d[v_r] \le d[v_1] + 1$ and $d[v_i] \le d[v_{i+1}]$ for $i = 1, 2, \ldots, r-1$.

Proof.

- By induction on the number of queue operations. First, $Q = \langle s \rangle$, so lemma holds. It holds after execution of both queue operations:
- v_1 is removed so v_2 is new leader (if Q is emptied, it holds trivially). By IH, $d[v_1] \le d[v_2]$. But then, $d[v_r] \le d[v_1] + 1 \le d[v_2] + 1$ and the rest of inequalities is unchanged.
- v_{r+1} is inserted into Q (line 17). In that time, u (whose adjacency list is being explored) is already removed from Q. By IH, $d[u] \le d[v_1]$. So, $d[v_{r+1}] = d[u] + 1 \le d[v_1] + 1$. Therefore, $d[v_r] \le_{IH} d[u] + 1 = d[v_{r+1}]$. The rest of inequalities is unchanged.

□

Corollary 5.

Let vertices v_i and v_j are stored in the queue during the computation of BFS such that v_i is inserted before v_j. Then, $d[v_i] \leq d[v_j]$ in the moment of insertion of v_j into the queue.

Proof.

By the previous lemma and the property that every vertex obtains final value of d at most once during the computation of BFS. □

Theorem 6 (Correctness of BFS).

Let $G = (V, E)$ be (di)graph and $s \in V$. Then, $BFS(G, s)$ explores all vertices $v \in V$ reachable from s and after it is finished $d[v] = \delta(s, v)$ for all $v \in V$. In addition, for every vertex $v \neq s$ reachable from s one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by edge $(\pi[v], v)$.

Proof.

- By contradiction. Let v is a vertex with minimal $\delta(s, v)$ such that $d[v] \neq \delta(s, v)$. Obviously, $v \neq s$.

Theorem 6 (Correctness of BFS).

Let $G = (V, E)$ be (di)graph and $s \in V$. Then, $BFS(G, s)$ explores all vertices $v \in V$ reachable from s and after it is finished $d[v] = \delta(s, v)$ for all $v \in V$. In addition, for every vertex $v \neq s$ reachable from s one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by edge $(\pi[v], v)$.

Proof.

- By contradiction. Let v is a vertex with minimal $\delta(s, v)$ such that $d[v] \neq \delta(s, v)$. Obviously, $v \neq s$.
- Lemma 3 states that $d[v] \geq \delta(s, v)$, therefore $d[v] > \delta(s, v)$. In addition, v must be reachable from s, otherwise $\delta(s, v) = \infty \geq d[v]$.

Theorem 6 (Correctness of BFS).

Let $G = (V, E)$ be (di)graph and $s \in V$. Then, $BFS(G, s)$ explores all vertices $v \in V$ reachable from s and after it is finished $d[v] = \delta(s, v)$ for all $v \in V$. In addition, for every vertex $v \neq s$ reachable from s one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by edge $(\pi[v], v)$.

Proof.

- By contradiction. Let v is a vertex with minimal $\delta(s, v)$ such that $d[v] \neq \delta(s, v)$. Obviously, $v \neq s$.
- Lemma 3 states that $d[v] \geq \delta(s, v)$, therefore $d[v] > \delta(s, v)$. In addition, v must be reachable from s, otherwise $\delta(s, v) = \infty \geq d[v]$.
- Let u be a vertex preceding v on a shortest path from s to v; that is, $\delta(s, v) = \delta(s, u) + 1$. Since $\delta(s, u) < \delta(s, v)$ and with respect to the choice of v, $d[u] = \delta(s, u)$.

Theorem 6 (Correctness of BFS).

Let $G = (V, E)$ be (di)graph and $s \in V$. Then, $BFS(G, s)$ explores all vertices $v \in V$ reachable from s and after it is finished $d[v] = \delta(s, v)$ for all $v \in V$. In addition, for every vertex $v \neq s$ reachable from s one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by edge $(\pi[v], v)$.

Proof.

- By contradiction. Let v is a vertex with minimal $\delta(s, v)$ such that $d[v] \neq \delta(s, v)$. Obviously, $v \neq s$.
- Lemma 3 states that $d[v] \geq \delta(s, v)$, therefore $d[v] > \delta(s, v)$. In addition, v must be reachable from s, otherwise $\delta(s, v) = \infty \geq d[v]$.
- Let u be a vertex preceding v on a shortest path from s to v; that is, $\delta(s, v) = \delta(s, u) + 1$. Since $\delta(s, u) < \delta(s, v)$ and with respect to the choice of v, $d[u] = \delta(s, u)$.
- Altogether, $d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$.

Proof (cont.).

- Consider BFS in the moment when we dequeue u from Q (line 11), i.e. v is either WHITE, GREY, or BLACK.

Proof (cont.).

- Consider BFS in the moment when we dequeue u from Q (line 11), i.e. v is either WHITE, GREY, or BLACK.
- v is WHITE, then line 15 sets $d[v] = d[u] + 1$ – contradiction.

Proof (cont.).

- Consider BFS in the moment when we dequeue u from Q (line 11), i.e. v is either WHITE, GREY, or BLACK.
- v is WHITE, then line 15 sets $d[v] = d[u] + 1$ – contradiction.
- v is BLACK, then v is already dequeued from Q and by Corollary 5, $d[v] \leq d[u]$ – contradiction.

Proof (cont.).

- Consider BFS in the moment when we dequeue u from Q (line 11), i.e. v is either WHITE, GREY, or BLACK.
- v is WHITE, then line 15 sets $d[v] = d[u] + 1$ – contradiction.
- v is BLACK, then v is already dequeued from Q and by Corollary 5, $d[v] \leq d[u]$ – contradiction.
- v is GREY, then v is greyed during picking another vertex w that was dequeued from Q before u. In addition, $d[v] = d[w] + 1$. By Corollary 5, $d[w] \leq d[u]$, i.e. $d[v] \leq d[u] + 1$ – contradiction.

Proof (cont.).

- Consider BFS in the moment when we dequeue u from Q (line 11), i.e. v is either WHITE, GREY, or BLACK.
- v is WHITE, then line 15 sets $d[v] = d[u] + 1$ – contradiction.
- v is BLACK, then v is already dequeued from Q and by Corollary 5, $d[v] \leq d[u]$ – contradiction.
- v is GREY, then v is greyed during picking another vertex w that was dequeued from Q before u. In addition, $d[v] = d[w] + 1$. By Corollary 5, $d[w] \leq d[u]$, i.e. $d[v] \leq d[u] + 1$ – contradiction.
- Therefore, $d[v] = \delta(s, v)$ for all $v \in V$. Furthermore, all vertices reachable from s must be visited, otherwise its d value is infinity.

Proof (cont.).

- Consider BFS in the moment when we dequeue u from Q (line 11), i.e. v is either WHITE, GREY, or BLACK.
- v is WHITE, then line 15 sets $d[v] = d[u] + 1$ – contradiction.
- v is BLACK, then v is already dequeued from Q and by Corollary 5, $d[v] \leq d[u]$ – contradiction.
- v is GREY, then v is greyed during picking another vertex w that was dequeued from Q before u. In addition, $d[v] = d[w] + 1$. By Corollary 5, $d[w] \leq d[u]$, i.e. $d[v] \leq d[u] + 1$ – contradiction.
- Therefore, $d[v] = \delta(s, v)$ for all $v \in V$. Furthermore, all vertices reachable from s must be visited, otherwise its d value is infinity.
- Finally, observe that if $\pi[v] = u$, then $d[v] = d[u] + 1$; that is, a shortest path from s to v can be obtained by addition of edge $(\pi[v], v)$ to the end of a shortest path from s to $\pi[v]$.

□

Breadth-First Search Tree (BFS Tree)

- Let π be an array of predecessors computed by $BFS(G,s)$ for some $G = (V,E)$ and $s \in V$.

Breadth-First Search Tree (BFS Tree)

- Let π be an array of predecessors computed by $BFS(G, s)$ for some $G = (V, E)$ and $s \in V$.
- Predecessor subgraph of G is defined as $G_\pi = (V_\pi, E_\pi)$, where
- $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$ and
- $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$.

Breadth-First Search Tree (BFS Tree)

- Let π be an array of predecessors computed by $BFS(G, s)$ for some $G = (V, E)$ and $s \in V$.
- Predecessor subgraph of G is defined as $G_\pi = (V_\pi, E_\pi)$, where
- $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$ and
- $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$.
- G_π is BFS tree, if V_π contains only vertices reachable from s and for all $v \in V_\pi$, there exists the only path from s to v that is the shortest path.
- Since G_π is connected and $|E_\pi| = |V_\pi| - 1$, G_π is a tree.

Lemma 7.

Let G be (di)graph. Procedure BFS constructs π such that G_π is BFS tree.

Proof.

- Line 16 of BFS sets $\pi[v] = u$ iff $(u, v) \in E$ and $\delta(s, v) < \infty$.

Lemma 7.
Let G be (di)graph. Procedure BFS constructs π such that G_π is BFS tree.

Proof.

- Line 16 of BFS sets $\pi[v] = u$ iff $(u, v) \in E$ and $\delta(s, v) < \infty$.
- V_π contains only vertices reachable from s.

Lemma 7.
Let G be (di)graph. Procedure BFS constructs π such that G_π is BFS tree.

Proof.

- Line 16 of BFS sets $\pi[v] = u$ iff $(u, v) \in E$ and $\delta(s, v) < \infty$.
- V_π contains only vertices reachable from s.
- Since G_π is tree, G_π contains only one path from s to each other vertex.

Lemma 7.

Let G be (di)graph. Procedure BFS constructs π such that G_π is BFS tree.

Proof.

- Line 16 of BFS sets $\pi[v] = u$ iff $(u, v) \in E$ and $\delta(s, v) < \infty$.
- V_π contains only vertices reachable from s.
- Since G_π is tree, G_π contains only one path from s to each other vertex.
- By inductive application of Theorem 6, each such path is a shortest one.

□

How to print the shortest path from s to v?

```
PRINT-PATH(G, s, v)
if v = s
   then print s
   else if π[v] = NIL
           then print "No path from " s " to " v "!"
           else PRINT-PATH(G, s, π[v])
                print v
```

Its time complexity is $O(n)$.

Exercises

1. Given an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet E_π cannot be produced by running $\mathrm{BFS}(G, s)$, no matter how the vertices are ordered in each adjacency list.
2. Give an efficient algorithm to compute whether the given undirected graph is bipartite.
3. The diameter of a tree $T = (V, E)$ is defined as $max_{u,v \in V}\delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Depth-First Search

Depth-First Search (DFS)

- Input: (un)directed graph $G = (V, E)$.

Depth-First Search (DFS)

- Input: (un)directed graph $G = (V, E)$.
- On contrary to BFS, DFS visits all vertices.
- It colors the vertices with WHITE, GREY, and BLACK color as well.
- The array of predecessors π is in use.

Depth-First Search (DFS)

- Input: (un)directed graph $G = (V, E)$.
- On contrary to BFS, DFS visits all vertices.
- It colors the vertices with WHITE, GREY, and BLACK color as well.
- The array of predecessors π is in use.
- Creates a DFS forest that contains all vertices such that $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(\pi[v], v) : v \in V,\ \pi[v] \neq \text{NIL}\}.$$

- Graph representation – Adjacency-list representation.
- $color[u] \in \{$WHITE, GREY, BLACK$\}$.
- $d[u]$ is a timestamp of the first vertex discover (color changed to GREY).
- $f[u]$ is a timestamp of the finishing time of vertex u (color changed to BLACK).

- Graph representation – Adjacency-list representation.
- $color[u] \in \{\text{WHITE, GREY, BLACK}\}$.
- $d[u]$ is a timestamp of the first vertex discover (color changed to GREY).
- $f[u]$ is a timestamp of the finishing time of vertex u (color changed to BLACK).
- $1 \leq d[u] < f[u] \leq 2n$.

- Graph representation – Adjacency-list representation.
- $color[u] \in \{$WHITE, GREY, BLACK$\}$.
- $d[u]$ is a timestamp of the first vertex discover (color changed to GREY).
- $f[u]$ is a timestamp of the finishing time of vertex u (color changed to BLACK).
- $1 \leq d[u] < f[u] \leq 2n$.
- $color[u] =$ *WHITE* before time $d[u]$.
- $color[u] =$ *GREY* between time $d[u]$ and $f[u]$.
- $color[u] =$ *BLACK* after time $f[u]$.

- Graph representation – Adjacency-list representation.
- $color[u] \in \{\text{WHITE, GREY, BLACK}\}$.
- $d[u]$ is a timestamp of the first vertex discover (color changed to GREY).
- $f[u]$ is a timestamp of the finishing time of vertex u (color changed to BLACK).
- $1 \leq d[u] < f[u] \leq 2n$.
- $color[u] = \mathit{WHITE}$ before time $d[u]$.
- $color[u] = \mathit{GREY}$ between time $d[u]$ and $f[u]$.
- $color[u] = \mathit{BLACK}$ after time $f[u]$.
- *time* is a global variable (ticks after each *color* change).

```
DFS(G)
for each vertex u ∈ V
    color[u] ← WHITE
    π[u] ← NIL
time ← 0
for each vertex u ∈ V
    if color[u] = WHITE
       then DFS-VISIT(G, u)
```

```
DFS(G)
for each vertex u ∈ V
    color[u] ← WHITE
    π[u] ← NIL
time ← 0
for each vertex u ∈ V
    if color[u] = WHITE
       then DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
color[u] ← GREY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    if color[v] = WHITE
       then π[v] ← u
            DFS-VISIT(G, v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

DFS – Example

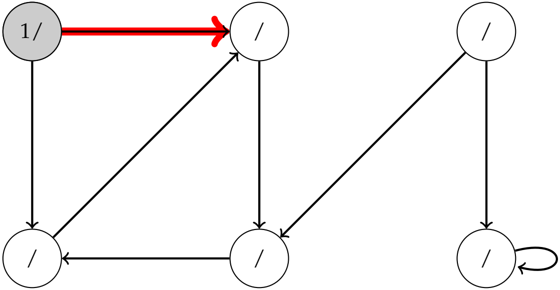


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

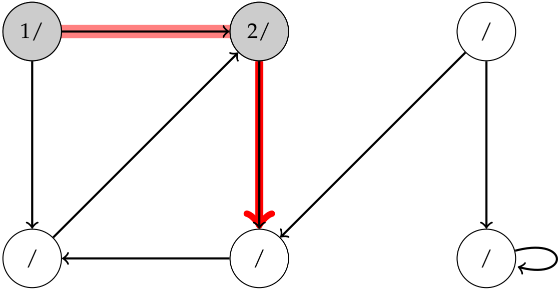


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

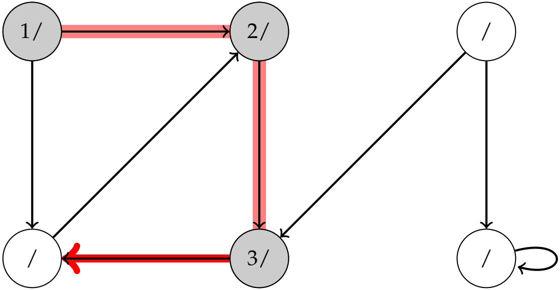


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

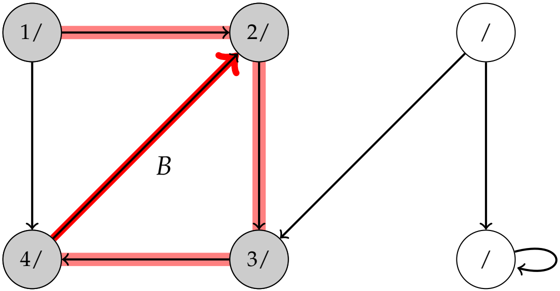


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

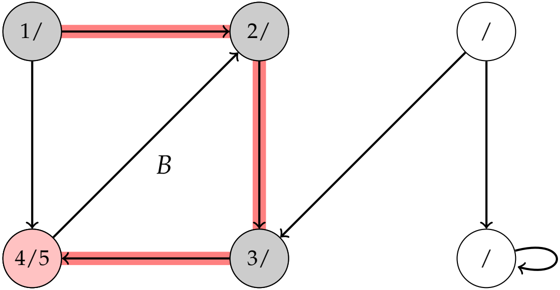


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

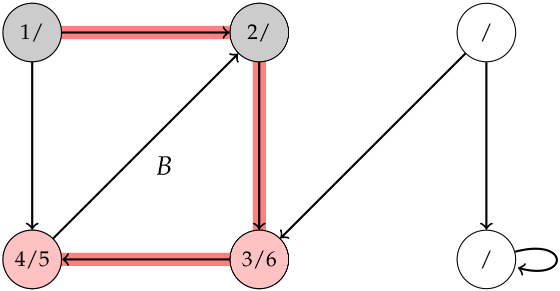


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

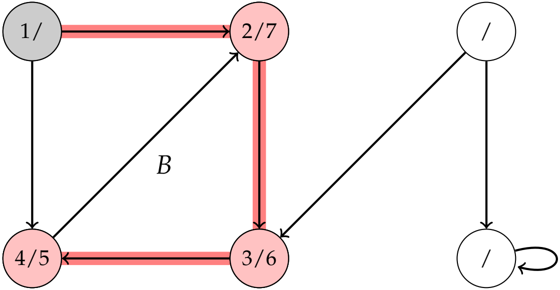


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

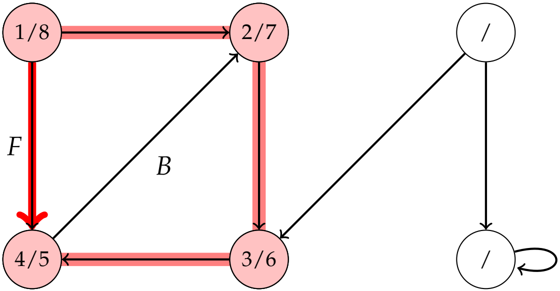


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

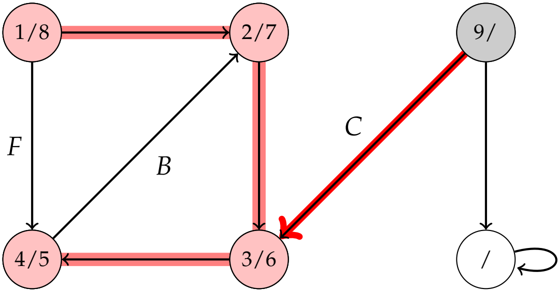


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

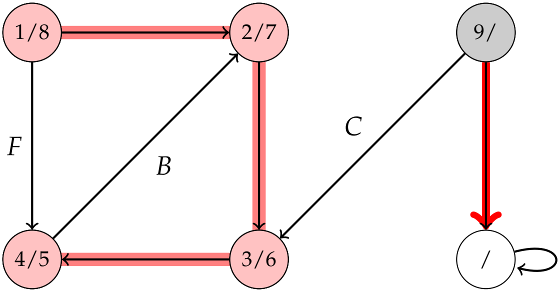


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

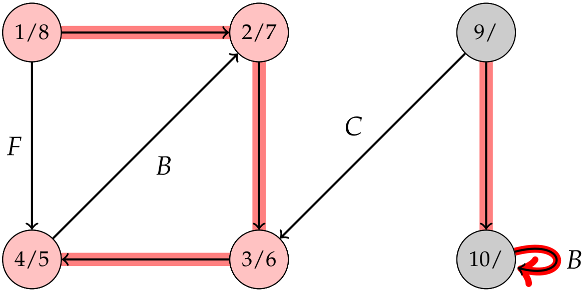


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

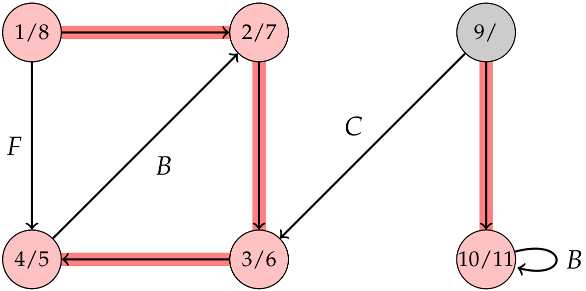


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

DFS – Example

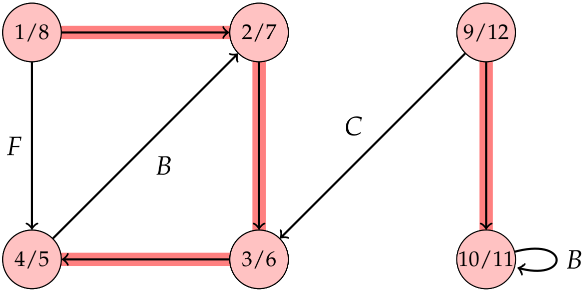


Figure: Vertex u is labeled by $d[u]/f[u]$. B, F, and C denote Back, Forward, and Cross edge, respectively.

Time Complexity of DFS

```
DFS(G)
1  for each vertex u ∈ V
2      color[u] ← WHITE
3      π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V
6      if color[u] = WHITE
7          then DFS-VISIT(G, u)
```

- Loops at lines 1–3 and 5–7 without DFS-VISIT calls take $\Theta(n)$.

Time Complexity of DFS-VISIT

```
DFS-VISIT(G, u)
    color[u] ← GREY
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        if color[v] = WHITE
            then π[v] ← u
                DFS-VISIT(G, v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

- DFS-VISIT is called only for white vertices and DFS-VISIT immediately changes their color to GREY. So, DFS-VISIT is called exactly once for each vertex $v \in V$.

Time Complexity of DFS-VISIT

```
DFS-VISIT(G, u)
1   color[u] ← GREY
2   time ← time + 1
3   d[u] ← time
4   for each v ∈ Adj[u]
5       if color[v] = WHITE
6          then π[v] ← u
7               DFS-VISIT(G, v)
8   color[u] ← BLACK
9   time ← time + 1
10  f[u] ← time
```

- DFS-VISIT is called only for white vertices and DFS-VISIT immediately changes their color to GREY. So, DFS-VISIT is called exactly once for each vertex $v \in V$.
- For each vertex v, the loop on lines 4–7 iterates $|Adj[v]|$-times.

Time Complexity of DFS-VISIT

```
DFS-VISIT(G, u)
1   color[u] ← GREY
2   time ← time + 1
3   d[u] ← time
4   for each v ∈ Adj[u]
5       if color[v] = WHITE
6          then π[v] ← u
7               DFS-VISIT(G, v)
8   color[u] ← BLACK
9   time ← time + 1
10  f[u] ← time
```

- DFS-VISIT is called only for white vertices and DFS-VISIT immediately changes their color to GREY. So, DFS-VISIT is called exactly once for each vertex $v \in V$.
- For each vertex v, the loop on lines 4–7 iterates $|Adj[v]|$-times.
- Since $\sum_{v \in V} |Adj[v]| = \Theta(m)$, the total cost of lines 4–7 is $\Theta(m)$.

Time Complexity of DFS-VISIT

```
DFS-VISIT(G, u)
1   color[u] ← GREY
2   time ← time + 1
3   d[u] ← time
4   for each v ∈ Adj[u]
5       if color[v] = WHITE
6          then π[v] ← u
7               DFS-VISIT(G, v)
8   color[u] ← BLACK
9   time ← time + 1
10  f[u] ← time
```

- DFS-VISIT is called only for white vertices and DFS-VISIT immediately changes their color to GREY. So, DFS-VISIT is called exactly once for each vertex $v \in V$.
- For each vertex v, the loop on lines 4–7 iterates $|Adj[v]|$-times.
- Since $\sum_{v \in V} |Adj[v]| = \Theta(m)$, the total cost of lines 4–7 is $\Theta(m)$.
- Therefore, the running time is $\Theta(m + n)$.

Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices u and v, exactly one of the following conditions holds:

- intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint, and neither u nor v is descendant of the other in DFS forest,
- interval $[d[u], f[u]]$ is contained within the interval $[d[v], f[v]]$ and u is a descendant of v in a DFS tree, or
- interval $[d[v], f[v]]$ is contained within the interval $[d[u], f[u]]$ and v is a descendant of u in a DFS tree.

Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices u and v, exactly one of the following conditions holds:

- intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint, and neither u nor v is descendant of the other in DFS forest,
- interval $[d[u], f[u]]$ is contained within the interval $[d[v], f[v]]$ and u is a descendant of v in a DFS tree, or
- interval $[d[v], f[v]]$ is contained within the interval $[d[u], f[u]]$ and v is a descendant of u in a DFS tree.

Proof for $d[u] < d[v]$ (Homework: prove case $d[v] < d[u]$).

- Subcase $d[v] < f[u]$: Then, v was discovered while u was still GREY. Since v was discovered later than u, v is finished before u. Hence, $f[v] < f[u]$.

Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices u and v, exactly one of the following conditions holds:

- intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint, and neither u nor v is descendant of the other in DFS forest,
- interval $[d[u], f[u]]$ is contained within the interval $[d[v], f[v]]$ and u is a descendant of v in a DFS tree, or
- interval $[d[v], f[v]]$ is contained within the interval $[d[u], f[u]]$ and v is a descendant of u in a DFS tree.

Proof for $d[u] < d[v]$ (Homework: prove case $d[v] < d[u]$).

- Subcase $d[v] < f[u]$: Then, v was discovered while u was still GREY. Since v was discovered later than u, v is finished before u. Hence, $f[v] < f[u]$.
- Subcase $f[u] < d[v]$: Then, from the definition $d[u] < f[u]$ and $d[v] < f[v]$, so both intervals are disjoint. Moreover, neither vertex was discovered while the other was GREY, and so neither vertex is a descendant of the other.

Corollary 8.

Vertex v is descendant of vertex u in DFS forest of $G = (V, E)$ iff

$$d[u] < d[v] < f[v] < f[u] .$$

White Path Theorem

In DFS forest of graph $G = (V, E)$, vertex v is descendant of vertex u iff in time $d[u]$ there is a path from u to v from WHITE vertices only.

White Path Theorem

In DFS forest of graph $G = (V, E)$, vertex v is descendant of vertex u iff in time $d[u]$ there is a path from u to v from WHITE vertices only.

Proof.

$\Rightarrow$: Let v be descendant of u. Let w be a vertex on the path from u to v in the DFS forest. Since w is descendant of u and by the previous corollary, it holds that $d[u] < d[w]$. So, w is WHITE in time $d[u]$.

White Path Theorem

In DFS forest of graph $G = (V, E)$, vertex v is descendant of vertex u iff in time $d[u]$ there is a path from u to v from WHITE vertices only.

Proof.

$\Rightarrow$: Let v be descendant of u. Let w be a vertex on the path from u to v in the DFS forest. Since w is descendant of u and by the previous corollary, it holds that $d[u] < d[w]$. So, w is WHITE in time $d[u]$.

$\Leftarrow$: Let v be the nearest vertex of u reachable from u in time $d[u]$ by some WHITE path such that v is not a descendant of u in DFS forest.

White Path Theorem

In DFS forest of graph $G = (V, E)$, vertex v is descendant of vertex u iff in time $d[u]$ there is a path from u to v from WHITE vertices only.

Proof.

$\Rightarrow$: Let v be descendant of u. Let w be a vertex on the path from u to v in the DFS forest. Since w is descendant of u and by the previous corollary, it holds that $d[u] < d[w]$. So, w is WHITE in time $d[u]$.

$\Leftarrow$: Let v be the nearest vertex of u reachable from u in time $d[u]$ by some WHITE path such that v is not a descendant of u in DFS forest.

- Let w be predecessor of v on the WHITE path. Then, w is descendant of u and, by the previous corollary, $f[w] \leq f[u]$ (w can coincide with u).

White Path Theorem

In DFS forest of graph $G = (V, E)$, vertex v is descendant of vertex u iff in time $d[u]$ there is a path from u to v from WHITE vertices only.

Proof.

- $\Rightarrow$: Let v be descendant of u. Let w be a vertex on the path from u to v in the DFS forest. Since w is descendant of u and by the previous corollary, it holds that $d[u] < d[w]$. So, w is WHITE in time $d[u]$.
- $\Leftarrow$: Let v be the nearest vertex of u reachable from u in time $d[u]$ by some WHITE path such that v is not a descendant of u in DFS forest.
  - Let w be predecessor of v on the WHITE path. Then, w is descendant of u and, by the previous corollary, $f[w] \leq f[u]$ (w can coincide with u).
  - Since v must be discovered after u but before finishing w, we have $d[u] < d[v] < f[w] \leq f[u]$.

White Path Theorem

In DFS forest of graph $G = (V,E)$, vertex v is descendant of vertex u iff in time $d[u]$ there is a path from u to v from WHITE vertices only.

Proof.

$\Rightarrow$: Let v be descendant of u. Let w be a vertex on the path from u to v in the DFS forest. Since w is descendant of u and by the previous corollary, it holds that $d[u] < d[w]$. So, w is WHITE in time $d[u]$.

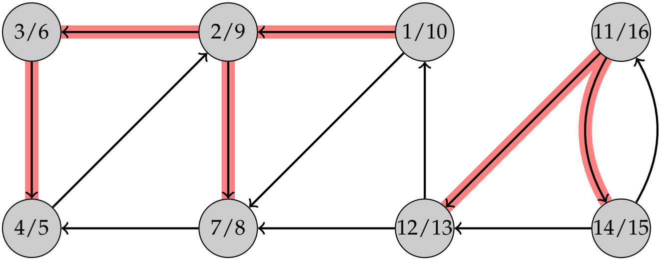
$\Leftarrow$: Let v be the nearest vertex of u reachable from u in time $d[u]$ by some WHITE path such that v is not a descendant of u in DFS forest.

- Let w be predecessor of v on the WHITE path. Then, w is descendant of u and, by the previous corollary, $f[w] \leq f[u]$ (w can coincide with u).
- Since v must be discovered after u but before finishing w, we have $d[u] < d[v] < f[w] \leq f[u]$.
- Parenthesis Theorem says that interval $[d[v], f[v]]$ is completely included in interval $[d[u], f[u]]$. And by the previous corollary, v is descendant of u.

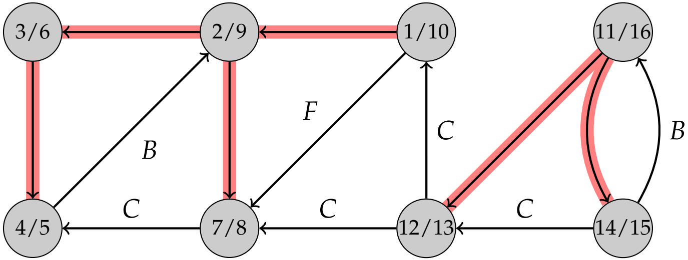
Edge Classification

1. **Tree edges** are edges in DFS forest G_π. (u, v) is a tree edge if v was firstly discovered by exploring edge (u, v). These edges are highlighted using red color in the figures.
2. **Back edges** are edges (u, v) connecting u to its predecessor v in DFS forest. Self-loop is always back edge.
3. **Forward edges** are non-tree edges (u, v) connecting u to its descendant v in DFS forest.
4. **Cross edges** are all other edges.

Edge Classification – Example

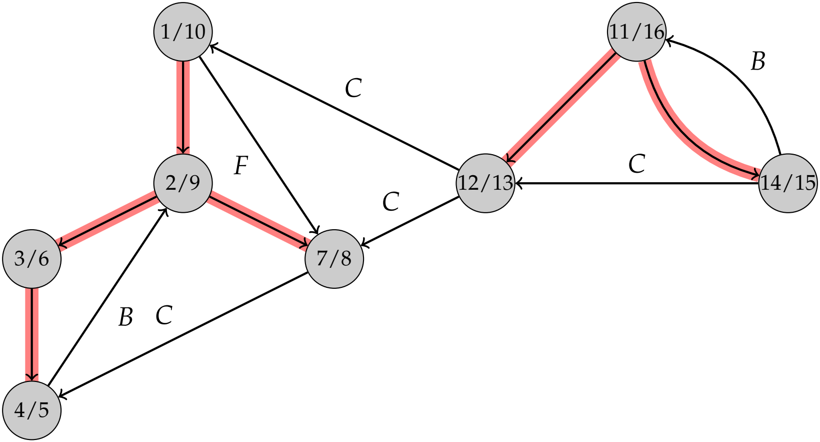


Edge Classification – Example



Drawing a Graph

We can draw every graph such that tree and forward edges lead downwards and back edges lead upwards.



DFS and Edge Classification

Let (u, v) be an edge. Then, using a color of v during DFS computation, we can classify (u, v) as follows:

1. WHITE indicates a tree edge,

DFS and Edge Classification

Let (u, v) be an edge. Then, using a color of v during DFS computation, we can classify (u, v) as follows:

1. WHITE indicates a tree edge,
2. GREY indicates a back edge, and

DFS and Edge Classification

Let (u, v) be an edge. Then, using a color of v during DFS computation, we can classify (u, v) as follows:

1. WHITE indicates a tree edge,
2. GREY indicates a back edge, and
3. BLACK indicates a forward or cross edge:
   - (u, v) is a forward edge, if $d[u] < d[v]$.
   - (u, v) is a cross edge, if $d[u] > d[v]$.

Edge Classification in Undirected Graph

Theorem 9.
During the DFS computation of undirected graph G, each edge is either a tree edge or a back edge.

Proof.

- Let (u,v) is an arbitrary edge of G and let $d[u] < d[v]$.

Edge Classification in Undirected Graph

Theorem 9.
During the DFS computation of undirected graph G, each edge is either a tree edge or a back edge.

Proof.

- Let (u,v) is an arbitrary edge of G and let $d[u] < d[v]$.
- Then, v becomes BLACK while u is still GREY.

Edge Classification in Undirected Graph

Theorem 9.
During the DFS computation of undirected graph G, each edge is either a tree edge or a back edge.

Proof.

- Let (u, v) is an arbitrary edge of G and let $d[u] < d[v]$.
- Then, v becomes BLACK while u is still GREY.
- If (u, v) is firstly explored in the direction from u to v, then v is WHITE – otherwise we would have explored (u, v) in the other direction (from v to u). Thus, (u, v) is a tree edge.

Edge Classification in Undirected Graph

Theorem 9.
During the DFS computation of undirected graph G, each edge is either a tree edge or a back edge.

Proof.

- Let (u, v) is an arbitrary edge of G and let $d[u] < d[v]$.
- Then, v becomes BLACK while u is still GREY.
- If (u, v) is firstly explored in the direction from u to v, then v is WHITE – otherwise we would have explored (u, v) in the other direction (from v to u). Thus, (u, v) is a tree edge.
- If (u, v) is firstly explored in the direction from v to u, u is still GREY – since u is still GREY at the time the edge is explored for the first time, then (u, v) is a back edge.

□

Exercises

1. Give an efficient algorithm to find whether a given directed graph contains a cycle, and analyze the running time of your algorithm.

2. Let G be an undirected graph. Show how to modify $\textsc{Dfs}$ so that it assigns to each vertex v an integer label between 1 and k in array cc, where k is the number of connected components of G, such that $cc[u] = cc[v]$ if and only if u and v are in the same connected component.

Topological sort

Topological sort

- An application of DFS

Topological sort

- An application of DFS
- A topological sort of directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that if $(u, v) \in E$, then u appears before v in the ordering.

Topological sort

- An application of DFS
- A topological sort of directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that if $(u, v) \in E$, then u appears before v in the ordering.
- If G contains a cycle, then no linear ordering is possible.

Topological sort

- An application of DFS
- A topological sort of directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that if $(u, v) \in E$, then u appears before v in the ordering.
- If G contains a cycle, then no linear ordering is possible.

TOPOLOGICAL-SORT(G)
$L \leftarrow \emptyset$
call DFS(G) to compute finishing times $f[v]$
as each vertex is finished, insert it onto the front of L
return the linked list of vertices L

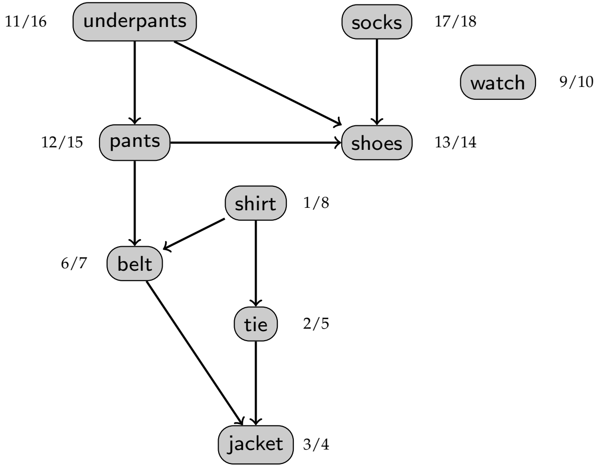
Topological sort

- An application of DFS
- A topological sort of directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that if $(u, v) \in E$, then u appears before v in the ordering.
- If G contains a cycle, then no linear ordering is possible.

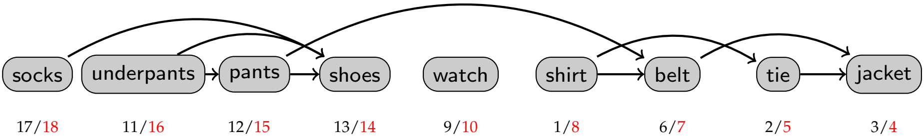
```
TOPOLOGICAL-SORT(G)
L ← ∅
call DFS(G) to compute finishing times f[v]
as each vertex is finished, insert it onto the front of L
return the linked list of vertices L
```

- Time complexity: DFS is $\Theta(m+n)$, add a vertex to the list is constant, so, in total, $\Theta(m+n)$.

Topological sort – Example



Topological sort – Example



Lemma 10.
Digraph G is acyclic iff $\textsc{Dfs}(G)$ finds no back edge.

Proof.

Lemma 10.
Digraph G is acyclic iff $\textsc{Dfs}(G)$ finds no back edge.

Proof.

$\Rightarrow$: Let (u, v) be a back edge. Then, u is descendant of v in DFS forest; that is, there is a path from v to u. So edge (u, v) closes a cycle.

Lemma 10.
Digraph G is acyclic iff $\textsc{Dfs}(G)$ finds no back edge.

Proof.

- $\Rightarrow$: Let (u, v) be a back edge. Then, u is descendant of v in DFS forest; that is, there is a path from v to u. So edge (u, v) closes a cycle.
- $\Leftarrow$: Let G contain a cycle, c. Let us show that then $\textsc{Dfs}(G)$ finds a back edge.

Lemma 10.

Digraph G is acyclic iff $\textsc{Dfs}(G)$ finds no back edge.

Proof.

$\Rightarrow$: Let (u, v) be a back edge. Then, u is descendant of v in DFS forest; that is, there is a path from v to u. So edge (u, v) closes a cycle.

$\Leftarrow$: Let G contain a cycle, c. Let us show that then $\textsc{Dfs}(G)$ finds a back edge.

- Let v be the first vertex of c discovered by $\textsc{Dfs}(G)$ procedure and let (u, v) be an edge that completes cycle c.

Lemma 10.
Digraph G is acyclic iff $\textsc{Dfs}(G)$ *finds no back edge.*

Proof.

$\Rightarrow$: Let (u,v) be a back edge. Then, u is descendant of v in DFS forest; that is, there is a path from v to u. So edge (u,v) closes a cycle.

$\Leftarrow$: Let G contain a cycle, c. Let us show that then $\textsc{Dfs}(G)$ finds a back edge.

- Let v be the first vertex of c discovered by $\textsc{Dfs}(G)$ procedure and let (u,v) be an edge that completes cycle c.
- In time $d[v]$, the edges of cycle c determine WHITE path from v to u.

Lemma 10.
Digraph G is acyclic iff $\text{DFS}(G)$ finds no back edge.

Proof.

$\Rightarrow$: Let (u, v) be a back edge. Then, u is descendant of v in DFS forest; that is, there is a path from v to u. So edge (u, v) closes a cycle.

$\Leftarrow$: Let G contain a cycle, c. Let us show that then $\text{DFS}(G)$ finds a back edge.

- Let v be the first vertex of c discovered by $\text{DFS}(G)$ procedure and let (u, v) be an edge that completes cycle c.
- In time $d[v]$, the edges of cycle c determine WHITE path from v to u.
- By WHITE path theorem, it holds that u is descendant of v in DFS forest. Therefore, (u, v) is a back edge.

□

Theorem 11.
TOPOLOGICAL-SORT(G) *procedure gives topological order for acyclic digraph* G*.*

Proof.

Theorem 11.
TOPOLOGICAL-SORT(G) *procedure gives topological order for acyclic digraph G.*

Proof.

- Let DFS be executed on an acyclic digraph $G = (V, E)$ such that DFS determines the values of $f[v]$.
- Now we need to show that if $(u, v) \in E$, then $f[v] < f[u]$.

Theorem 11.
TOPOLOGICAL-SORT(G) *procedure gives topological order for acyclic digraph G.*

Proof.

- Let DFS be executed on an acyclic digraph $G = (V, E)$ such that DFS determines the values of $f[v]$.
- Now we need to show that if $(u, v) \in E$, then $f[v] < f[u]$.
- Let (u, v) be an edge that is being explored by DFS(G) procedure. Then, v cannot be grey, otherwise v would be predecessor of u and (u, v) would be a back edge – contradiction to the previous lemma.

Theorem 11.
$\textsc{Topological-Sort}(G)$ *procedure gives topological order for acyclic digraph G.*

Proof.

- Let DFS be executed on an acyclic digraph $G = (V, E)$ such that DFS determines the values of $f[v]$.
- Now we need to show that if $(u, v) \in E$, then $f[v] < f[u]$.
- Let (u, v) be an edge that is being explored by $\textsc{Dfs}(G)$ procedure. Then, v cannot be grey, otherwise v would be predecessor of u and (u, v) would be a back edge – contradiction to the previous lemma.
- If v is WHITE, then v is descendant of u in DFS forest, so $f[v] < f[u]$.

Theorem 11.
$\textsc{Topological-Sort}(G)$ *procedure gives topological order for acyclic digraph G.*

Proof.

- Let DFS be executed on an acyclic digraph $G = (V, E)$ such that DFS determines the values of $f[v]$.
- Now we need to show that if $(u, v) \in E$, then $f[v] < f[u]$.
- Let (u, v) be an edge that is being explored by $\textsc{Dfs}(G)$ procedure. Then, v cannot be grey, otherwise v would be predecessor of u and (u, v) would be a back edge – contradiction to the previous lemma.
- If v is WHITE, then v is descendant of u in DFS forest, so $f[v] < f[u]$.
- If v is BLACK, then $f[v]$ is already set. Since u is still in exploration process (grey), its $f[u]$ is not set yet, so $f[v] < f[u]$.

□

Exercises

1. Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G.

2. Prove or disprove: If a directed graph G contains cycles, then $\textsc{Topological-Sort}(G)$ produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

Strongly Connected Components

Strongly Connected Components (SCC)

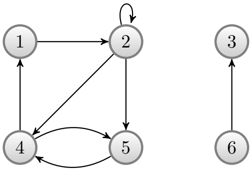
- An application of DFS

Strongly Connected Components (SCC)

- An application of DFS
- For digraph $G = (V, E)$, strongly connected component is the maximal set $C \subseteq V$ such that for every $u, v \in C$, $u \rightsquigarrow v$ (and also $v \rightsquigarrow u$).

Strongly Connected Components (SCC)

- An application of DFS
- For digraph $G = (V, E)$, strongly connected component is the maximal set $C \subseteq V$ such that for every $u, v \in C$, $u \rightsquigarrow v$ (and also $v \rightsquigarrow u$).



Graph with 3 SCCs:

- $\{1, 2, 4, 5\}$
- $\{3\}$
- $\{6\}$

- The transpose graph of $G = (V, E)$ is $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

- The transpose graph of $G = (V, E)$ is $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

```
SCC(G)
1  call DFS(G) to compute all f[u]
2  compute G^T
3  call modified DFS(G^T) such that DFS's main iteration takes vertices
           in the decreasing order according to f[u]
4  output all vertices of each DFS tree computed in line 3
           as a new strongly connected component
```

- The transpose graph of $G = (V, E)$ is $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

```
SCC(G)
1  call DFS(G) to compute all f[u]
2  compute G^T
3  call modified DFS(G^T) such that DFS's main iteration takes vertices
               in the decreasing order according to f[u]
4  output all vertices of each DFS tree computed in line 3
               as a new strongly connected component
```

- Time complexity: $\Theta(m + n)$.

- The transpose graph of $G = (V, E)$ is $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

```
SCC(G)
1  call DFS(G) to compute all f[u]
2  compute G^T
3  call modified DFS(G^T) such that DFS's main iteration takes vertices
             in the decreasing order according to f[u]
4  output all vertices of each DFS tree computed in line 3
             as a new strongly connected component
```

- Time complexity: $\Theta(m + n)$.
- How to create G^T from G in the adjacency-lists representation in time $O(m + n)$?

- The transpose graph of $G = (V, E)$ is $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

```
SCC(G)
1  call DFS(G) to compute all f[u]
2  compute G^T
3  call modified DFS(G^T) such that DFS's main iteration takes vertices
                in the decreasing order according to f[u]
4  output all vertices of each DFS tree computed in line 3
                as a new strongly connected component
```

- Time complexity: $\Theta(m + n)$.
- How to create G^T from G in the adjacency-lists representation in time $O(m + n)$?
- G and G^T has the same SCCs – u and v are mutually reachable in G if and only if they are mutually reachable in G^T.

SCC – Example

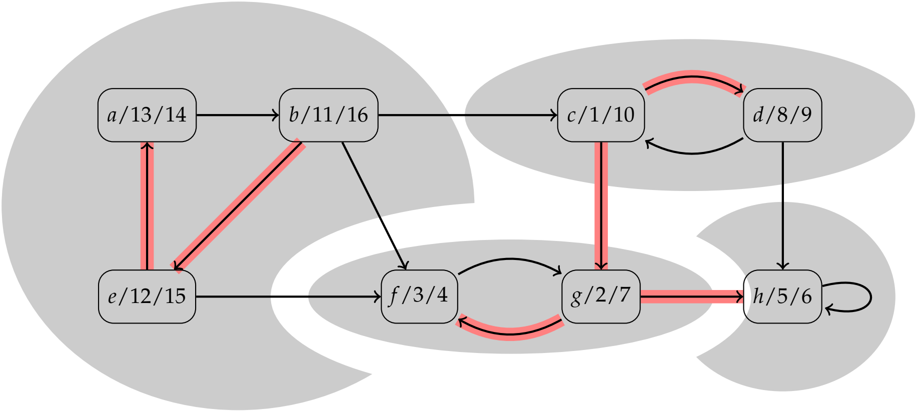


Figure: Result of line 1 of $\text{SCC}(G)$. Tree edges are red. Grey background forms the boundary of SCCs.

SCC – Example

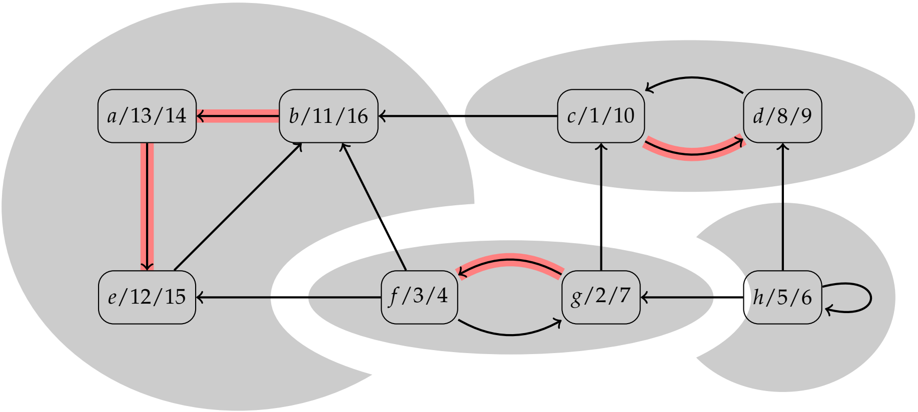
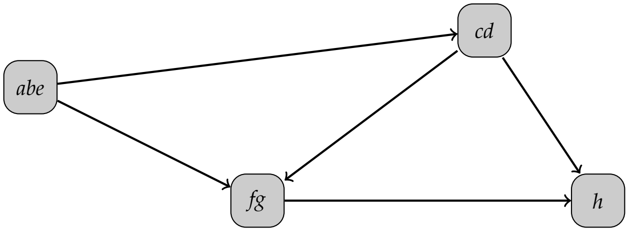


Figure: Graph G^T and result of line 3 of $\textsc{Scc}(G)$. b, c, g and h – roots in DFS forest. Each tree $\approx$ one SCC.

- The component graph of $G = (V, E)$ is graph $G^{scc} = (V^{scc}, E^{scc})$ defined as follows:
  - Let $C_1, C_2, \ldots, C_k$ be SCCs of G.
  - $V^{scc} = \{v_1, v_2, \ldots, v_k\} \subseteq V$, $V^{scc} \cap C_i \neq \emptyset$, $i = 1, 2, \ldots, k$.
  - $(v_i, v_j) \in E^{scc}$, if there exist $x \in C_i$ and $y \in C_j$ such that $(x, y) \in E$.
  - Informally: By contracting all edges incident to the vertices of the same SCCs, we get G^{SCC}.



Properties of Component Graph

Lemma 12.
Let C, C' *be two different SCCs of a digraph* $G = (V, E)$. *Let* $u, v \in C$, $u', v' \in C'$ *and* $u \rightsquigarrow u'$ *in* G. *Then, it DOES NOT hold that* $v' \rightsquigarrow v$.

Properties of Component Graph

Lemma 12.
Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $u, v \in C$, $u', v' \in C'$ and $u \rightsquigarrow u'$ in G. Then, it DOES NOT hold that $v' \rightsquigarrow v$.

Proof.
If $v' \rightsquigarrow v$, then $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$; that is, u and v' are mutually reachable – contradiction. □

Properties of Component Graph

Lemma 12.

Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $u, v \in C$, $u', v' \in C'$ and $u \rightsquigarrow u'$ in G. Then, it DOES NOT hold that $v' \rightsquigarrow v$.

Proof.

If $v' \rightsquigarrow v$, then $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$; that is, u and v' are mutually reachable – contradiction. □

- In what follows, consider only times $d[u]$ and $f[u]$ computed by the first call of DFS procedure.
- If necessary, the values from the second call of DFS are denotes as $d_3[u]$ and $f_3[u]$.

- Let $U \subseteq V$. Then, $d(U) = \min_{u \in U}\{d[u]\}$ and $f(U) = \max_{u \in U}\{f[u]\}$.

- Let $U \subseteq V$. Then, $d(U) = \min_{u \in U}\{d[u]\}$ and $f(U) = \max_{u \in U}\{f[u]\}$.

Lemma 13.

Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $(u, v) \in E$, $u \in C$, $v \in C'$. Then, $f(C) > f(C')$.

- Let $U \subseteq V$. Then, $d(U) = \min_{u \in U}\{d[u]\}$ and $f(U) = \max_{u \in U}\{f[u]\}$.

Lemma 13.

Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $(u, v) \in E$, $u \in C$, $v \in C'$. Then, $f(C) > f(C')$.

Proof

- 1) $d(C) < d(C')$ – let x be the first discovered vertex in C. In time $d[x]$, all vertices from $C \cup C'$ are WHITE. For $w \in C'$ there exists a WHITE path $x \rightsquigarrow u \to v \rightsquigarrow w$. By WHITE path theorem, all vertices from $C \cup C'$ are descendants of x in its DFS tree. Then, collorary from Parenthesis theorem says that $f[x] = f(C) > f(C')$.

- Let $U \subseteq V$. Then, $d(U) = \min_{u \in U}\{d[u]\}$ and $f(U) = \max_{u \in U}\{f[u]\}$.

Lemma 13.

Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $(u, v) \in E$, $u \in C$, $v \in C'$. Then, $f(C) > f(C')$.

Proof

- 1) $d(C) < d(C')$ – let x be the first discovered vertex in C. In time $d[x]$, all vertices from $C \cup C'$ are WHITE. For $w \in C'$ there exists a WHITE path $x \rightsquigarrow u \to v \rightsquigarrow w$. By WHITE path theorem, all vertices from $C \cup C'$ are descendants of x in its DFS tree. Then, collorary from Parenthesis theorem says that $f[x] = f(C) > f(C')$.
- 2) $d(C) > d(C')$ – let y be the first discovered in C'. In time $d[y]$, all vertices from C' are WHITE and there exists a WHITE path from y to every vertex of C'. By WHITE path theorem and corollary of Parenthesis theorem, we have $f[y] = f(C')$.

- Let $U \subseteq V$. Then, $d(U) = \min_{u \in U}\{d[u]\}$ and $f(U) = \max_{u \in U}\{f[u]\}$.

Lemma 13.

Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $(u, v) \in E$, $u \in C$, $v \in C'$. Then, $f(C) > f(C')$.

Proof

- 1) $d(C) < d(C')$ – let x be the first discovered vertex in C. In time $d[x]$, all vertices from $C \cup C'$ are WHITE. For $w \in C'$ there exists a WHITE path $x \rightsquigarrow u \to v \rightsquigarrow w$. By WHITE path theorem, all vertices from $C \cup C'$ are descendants of x in its DFS tree. Then, collorary from Parenthesis theorem says that $f[x] = f(C) > f(C')$.
- 2) $d(C) > d(C')$ – let y be the first discovered in C'. In time $d[y]$, all vertices from C' are WHITE and there exists a WHITE path from y to every vertex of C'. By WHITE path theorem and corollary of Parenthesis theorem, we have $f[y] = f(C')$. In time $d[y]$, all vertices from C are WHITE. From the previous lemma, there is no path from C' to C. Therefore, vertices from C are WHITE in time $f[y]$ too. That is, $f[w] > f[y]$, $w \in C$, which gives us $f(C) > f(C')$.

Corollary 14.

Let C*,* C' *be two different SCCs of a digraph* $G = (V, E)$*. Let* $(u, v) \in E^T$*,* $u \in C$*,* $v \in C'$*. Then,* $f(C) < f(C')$*.*

Corollary 14.

Let C, C' be two different SCCs of a digraph $G = (V,E)$. Let $(u,v) \in E^T$, $u \in C$, $v \in C'$. Then, $f(C) < f(C')$.

Proof.

$(u,v) \in E^T$ implies that $(v,u) \in E$. Since SCCs of G and SCCs of G^T coincide, the previous lemma implies $f(C) < f(C')$. □

Corollary 14.

Let C, C' be two different SCCs of a digraph $G = (V, E)$. Let $(u, v) \in E^T$, $u \in C$, $v \in C'$. Then, $f(C) < f(C')$.

Proof.

$(u, v) \in E^T$ implies that $(v, u) \in E$. Since SCCs of G and SCCs of G^T coincide, the previous lemma implies $f(C) < f(C')$. □

Closing times of the second DFS

Observe that $f_3(C) > f_3(C')$ so $(u, v) \in E^T$ is a cross edge according to the classification from the second DFS.

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

- By induction on the number of DFS trees found at line 3. IH: First k trees found by line 3 of $\textsc{Scc}(G)$ are SCCs. IB: Trivial for $k = 0$.

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

- By induction on the number of DFS trees found at line 3. IH: First k trees found by line 3 of $\textsc{Scc}(G)$ are SCCs. IB: Trivial for $k = 0$.
- IS: Assume $(k+1)$-th found tree. Let u be its root and let u be in a SCC C.

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

- By induction on the number of DFS trees found at line 3. IH: First k trees found by line 3 of $\textsc{Scc}(G)$ are SCCs. IB: Trivial for $k = 0$.
- IS: Assume $(k+1)$-th found tree. Let u be its root and let u be in a SCC C.
- $f[u] = f(C) > f(C')$ for any SCC C' (different from C) that is not visited yet.

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

- By induction on the number of DFS trees found at line 3. IH: First k trees found by line 3 of $\textsc{Scc}(G)$ are SCCs. IB: Trivial for $k = 0$.
- IS: Assume $(k+1)$-th found tree. Let u be its root and let u be in a SCC C.
- $f[u] = f(C) > f(C')$ for any SCC C' (different from C) that is not visited yet.
- By IH, in time $d_3[u]$ all vertices in C are WHITE. By White Path Theorem, the rest of vertices from C are descendants of u in a DFS tree.

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

- By induction on the number of DFS trees found at line 3. IH: First k trees found by line 3 of $\textsc{Scc}(G)$ are SCCs. IB: Trivial for $k = 0$.
- IS: Assume $(k+1)$-th found tree. Let u be its root and let u be in a SCC C.
- $f[u] = f(C) > f(C')$ for any SCC C' (different from C) that is not visited yet.
- By IH, in time $d_3[u]$ all vertices in C are WHITE. By White Path Theorem, the rest of vertices from C are descendants of u in a DFS tree.
- By IH and the previous corollary, every edge of G^T leads from C to some already visited SCC.

Theorem 15.
$\textsc{Scc}(G)$ *procedure is correct.*

Proof

- By induction on the number of DFS trees found at line 3. IH: First k trees found by line 3 of $\textsc{Scc}(G)$ are SCCs. IB: Trivial for $k = 0$.
- IS: Assume $(k+1)$-th found tree. Let u be its root and let u be in a SCC C.
- $f[u] = f(C) > f(C')$ for any SCC C' (different from C) that is not visited yet.
- By IH, in time $d_3[u]$ all vertices in C are WHITE. By White Path Theorem, the rest of vertices from C are descendants of u in a DFS tree.
- By IH and the previous corollary, every edge of G^T leads from C to some already visited SCC.
- So no vertex from another SCC (different from C) is descendant of u during DFS of G^T. Therefore, the vertices of the tree form an SCC.

Exercises

1. How can the number of strongly connected components of a graph change if a new edge is added?

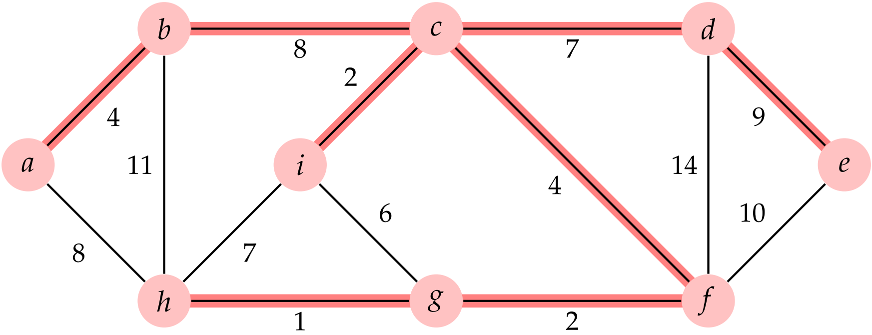
2. Give an $O(n+m)$-time algorithm to compute the component graph of digraph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the resulting graph (E is not a multiset).

Minimum Spanning Trees

Minimum Spanning Tree (MST)

- The first algorithm by mathematician from Brno, O. Borůvka, 1926 (in Czech).
- Let $G = (V, E)$ be a connected undirected graph with weight function
$$w : E \to \mathbb{R}\,.$$
- Goal: Find a subset of edges $T \subseteq E$ such that subgraph (V, T) is connected, acyclic and
$$w(T) = \sum_{(u,v) \in T} w(u, v)$$
is minimal.

Minimum Spanning Tree – Example



Generic Algorithm

```
GENERIC-MST(G, w)
A ← ∅
while A does not form a spanning tree
      do find an edge (u, v) ∈ E that is safe for A
          A ← A ∪ {(u, v)}
return A
```

- Loop invariant: Prior to each iteration, A is a subset of some MST.
- Edge $(u, v) \in E$ is safe edge for A, since $A \cup \{(u, v)\}$ maintains the invariant.
- Note: Greedy algorithm – making choice that is the best at the moment.

Definitions

- A cut of $G = (V, E)$ is a pair $(S, V - S)$ of V, $S \subseteq V$.

Definitions

- A cut of $G = (V, E)$ is a pair $(S, V - S)$ of V, $S \subseteq V$.
- An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of endpoints is in S and the other in $V - S$.

Definitions

- A cut of $G = (V, E)$ is a pair $(S, V - S)$ of V, $S \subseteq V$.
- An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of endpoints is in S and the other in $V - S$.
- A cut respects a set of edges A if no edge from A crosses the cut.

Definitions

- A cut of $G = (V, E)$ is a pair $(S, V - S)$ of V, $S \subseteq V$.
- An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of endpoints is in S and the other in $V - S$.
- A cut respects a set of edges A if no edge from A crosses the cut.
- An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

Theorem 16.

- *Let* $G = (V, E)$ *be a connected, undirected graph with real-valued weight function* w*.*
- *Let* $A \subseteq E$ *is included in some MST for* G*.*
- *Let* $(S, V - S)$ *be any cut of* G *that respects* A*.*
- *Let* (u, v) *be a light edge crossing* $(S, V - S)$*.*

Then, edge (u, v) *is safe for* A*.*

Proof

Theorem 16.

- *Let $G = (V, E)$ be a connected, undirected graph with real-valued weight function w.*
- *Let $A \subseteq E$ is included in some MST for G.*
- *Let $(S, V - S)$ be any cut of G that respects A.*
- *Let (u, v) be a light edge crossing $(S, V - S)$.*

Then, edge (u, v) is safe for A.

Proof

- Let T be a MST for G, $A \subseteq T$, $(u, v) \notin T$.

Theorem 16.

- *Let $G = (V, E)$ be a connected, undirected graph with real-valued weight function w.*
- *Let $A \subseteq E$ is included in some MST for G.*
- *Let $(S, V - S)$ be any cut of G that respects A.*
- *Let (u, v) be a light edge crossing $(S, V - S)$.*

Then, edge (u, v) is safe for A.

Proof

- Let T be a MST for G, $A \subseteq T$, $(u, v) \notin T$.
- $u \rightsquigarrow v$ is a path in T, and by adding (u, v) we create a cycle. E.g. let $u \in S$ and $v \in V - S$.

Theorem 16.

- *Let $G = (V, E)$ be a connected, undirected graph with real-valued weight function w.*
- *Let $A \subseteq E$ is included in some MST for G.*
- *Let $(S, V - S)$ be any cut of G that respects A.*
- *Let (u, v) be a light edge crossing $(S, V - S)$.*

Then, edge (u, v) is safe for A.

Proof

- Let T be a MST for G, $A \subseteq T$, $(u, v) \notin T$.
- $u \rightsquigarrow v$ is a path in T, and by adding (u, v) we create a cycle. E.g. let $u \in S$ and $v \in V - S$.
- Let (x, y) lies on $u \rightsquigarrow v$ in T crossing $(S, V - S)$. Since, the cut respects A, $(x, y) \notin A$.

Theorem 16.

- *Let $G = (V, E)$ be a connected, undirected graph with real-valued weight function w.*
- *Let $A \subseteq E$ is included in some MST for G.*
- *Let $(S, V - S)$ be any cut of G that respects A.*
- *Let (u, v) be a light edge crossing $(S, V - S)$.*

Then, edge (u, v) is safe for A.

Proof

- Let T be a MST for G, $A \subseteq T$, $(u, v) \notin T$.
- $u \rightsquigarrow v$ is a path in T, and by adding (u, v) we create a cycle. E.g. let $u \in S$ and $v \in V - S$.
- Let (x, y) lies on $u \rightsquigarrow v$ in T crossing $(S, V - S)$. Since, the cut respects A, $(x, y) \notin A$.
- $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$ is a spanning tree of G. Is T' minimal?

Proof.

- (u, v) is light edge crossing $(S, V - S)$ and (x, y) crossing the cut as well, so $w(u, v) \leq w(x, y)$.

Proof.

- (u, v) is light edge crossing $(S, V - S)$ and (x, y) crossing the cut as well, so $w(u, v) \leq w(x, y)$.
- Hence, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.

Proof.

- (u, v) is light edge crossing $(S, V - S)$ and (x, y) crossing the cut as well, so $w(u, v) \leq w(x, y)$.
- Hence, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.
- T is a MST, therefore $w(T) \leq w(T')$.

Proof.

- (u, v) is light edge crossing $(S, V - S)$ and (x, y) crossing the cut as well, so $w(u, v) \leq w(x, y)$.
- Hence, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.
- T is a MST, therefore $w(T) \leq w(T')$.

- Since $A \subseteq T$ and $(x, y) \notin A$, $A \subseteq T'$.

Proof.

- (u, v) is light edge crossing $(S, V - S)$ and (x, y) crossing the cut as well, so $w(u, v) \leq w(x, y)$.
- Hence, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.
- T is a MST, therefore $w(T) \leq w(T')$.

- Since $A \subseteq T$ and $(x, y) \notin A$, $A \subseteq T'$.
- Finally, $A \cup \{(u, v)\} \subseteq T'$. Since T' is MST as well, (u, v) is safe for A.

□

Exercises

1. Give a simple example of a connected graph $G = (V, E)$ such that the set of edges $\{(u, v) :$ there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a MST for G.

2. Show that a graph has a unique MST if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

Kruskal and Prim (Jarník) Algorithms – Principle

- Based on the generic greedy algorithm.
- Difference: How to pickup safe edge (line 3 of generic algorithm)?

Kruskal and Prim (Jarník) Algorithms – Principle

- Based on the generic greedy algorithm.
- Difference: How to pickup safe edge (line 3 of generic algorithm)?

- Kruskal: Set A forms a forest. Safe edge for A is an edge with the smallest weight connecting two different connected components.

Kruskal and Prim (Jarník) Algorithms – Principle

- Based on the generic greedy algorithm.
- Difference: How to pickup safe edge (line 3 of generic algorithm)?

- Kruskal: Set A forms a forest. Safe edge for A is an edge with the smallest weight connecting two different connected components.

- Prim (Jarník): Set A is a tree. Safe edge for A is an edge with the smallest weight connecting tree A with a (yet) non-tree vertex.

Kruskal Algorithm

Disjoint Dynamic Sets

- Set of non-empty sets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$
- Each set S_i identified by a representative (some member of S_i)
- Use: to represent a vertex membership to a tree in the given forest ($S_i \subseteq V$)

Disjoint Dynamic Sets

- Set of non-empty sets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$
- Each set S_i identified by a representative (some member of S_i)
- Use: to represent a vertex membership to a tree in the given forest ($S_i \subseteq V$)

Operations

- $\textsc{Make-Set}(v)$ creates a disjoint set for v.
- $\textsc{Find-Set}(v)$ returns the representative (pointer) from set containing v.
- $\textsc{Union}(u, v)$ unites two sets that contain u and v.

Disjoint Dynamic Sets

- Set of non-empty sets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$
- Each set S_i identified by a representative (some member of S_i)
- Use: to represent a vertex membership to a tree in the given forest ($S_i \subseteq V$)

Operations

- $\textsc{Make-Set}(v)$ creates a disjoint set for v.
- $\textsc{Find-Set}(v)$ returns the representative (pointer) from set containing v.
- $\textsc{Union}(u, v)$ unites two sets that contain u and v.

Implementation (Data structure)

- Linked-list representation (with weight-union heuristic; $O(m + n \log n)$)
- Rooted trees (with heuristics "union by rank" and "path compression"; $O(m\alpha(n))$, where α grows very slowly ($\alpha(n) \leq 4$))

Kruskal Algorithm

```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3       do MAKE-SET(v)
4  sort the edges of E into nondescreasing order by weight w
5  for each edge (u, v) ∈ E, taken in the order from step 4
6       do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9  return A
```

- MAKE-SET(v) creates a disjoint set for v.
- FIND-SET(v) returns a representative vertex from set containing v.
- UNION(u, v) combines two disjoint sets containing u and v.

Kruskal Algorithm – Time Complexity

```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3      do MAKE-SET(v)
4  sort the edges of E into nondescreasing order by weight w
5  for each edge (u, v) ∈ E, taken in the order from step 4
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

- Line 1: $O(1)$, Line 4: $O(m \log m)$. Lines 2-3: n-times MAKE-SET. Lines 5-8: $O(m)$-times FIND-SET and UNION – implementation-dependent running time (lines 2-3 and 5-8):

Kruskal Algorithm – Time Complexity

```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3      do MAKE-SET(v)
4  sort the edges of E into nondescreasing order by weight w
5  for each edge (u, v) ∈ E, taken in the order from step 4
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

- Line 1: $O(1)$, Line 4: $O(m \log m)$. Lines 2-3: n-times MAKE-SET. Lines 5-8: $O(m)$-times FIND-SET and UNION – implementation-dependent running time (lines 2-3 and 5-8):
  - By a linked-lists with heuristic: $O(m + n \log n)$.

Kruskal Algorithm – Time Complexity

```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3      do MAKE-SET(v)
4  sort the edges of E into nondescreasing order by weight w
5  for each edge (u, v) ∈ E, taken in the order from step 4
6      do if FIND-SET(u) ≠ FIND-SET(v)
7             then A ← A ∪ {(u, v)}
8                  UNION(u, v)
9  return A
```

- Line 1: $O(1)$, Line 4: $O(m \log m)$. Lines 2-3: n-times MAKE-SET. Lines 5-8: $O(m)$-times FIND-SET and UNION – implementation-dependent running time (lines 2-3 and 5-8):
  - By a linked-lists with heuristic: $O(m + n \log n)$.
  - By a rooted trees with 2 heuristics: $O((m + n)\alpha(n))$.

Kruskal Algorithm – Time Complexity

```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3      do MAKE-SET(v)
4  sort the edges of E into nondescreasing order by weight w
5  for each edge (u, v) ∈ E, taken in the order from step 4
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

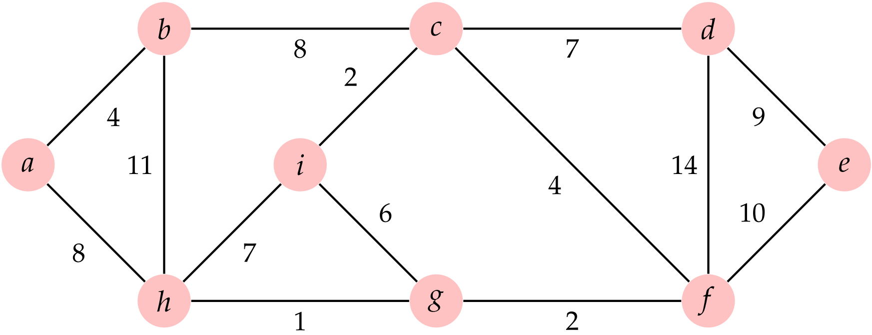
- Line 1: $O(1)$, Line 4: $O(m \log m)$. Lines 2-3: n-times MAKE-SET. Lines 5-8: $O(m)$-times FIND-SET and UNION – implementation-dependent running time (lines 2-3 and 5-8):
  - By a linked-lists with heuristic: $O(m + n \log n)$.
  - By a rooted trees with 2 heuristics: $O((m + n)\alpha(n))$.
- G is connected, so $m \geq n - 1$. Then, sets operations take $O(m\alpha(n))$. Since $\alpha(n) = O(\log n) = O(\log m)$, sorting outweighs by $O(m \log m)$.

Kruskal Algorithm – Time Complexity

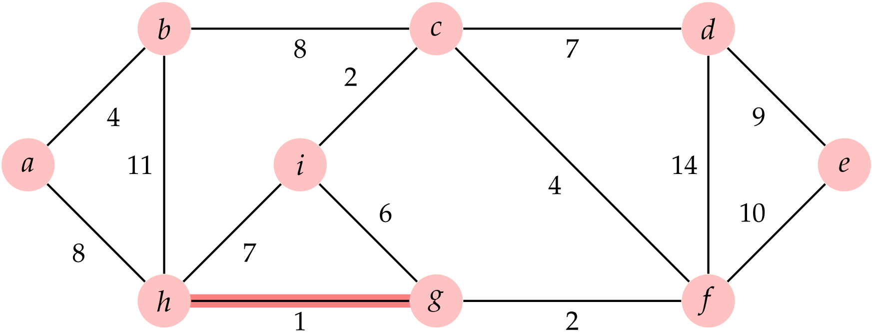
```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3      do MAKE-SET(v)
4  sort the edges of E into nondescreasing order by weight w
5  for each edge (u, v) ∈ E, taken in the order from step 4
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

- Line 1: $O(1)$, Line 4: $O(m \log m)$. Lines 2-3: n-times MAKE-SET. Lines 5-8: $O(m)$-times FIND-SET and UNION – implementation-dependent running time (lines 2-3 and 5-8):
  - By a linked-lists with heuristic: $O(m + n \log n)$.
  - By a rooted trees with 2 heuristics: $O((m + n)\alpha(n))$.
- G is connected, so $m \geq n - 1$. Then, sets operations take $O(m\alpha(n))$. Since $\alpha(n) = O(\log n) = O(\log m)$, sorting outweighs by $O(m \log m)$.
- Notice that $m < n^2$, so $\log m = O(\log n)$. Therefore, $O(m \log n)$.

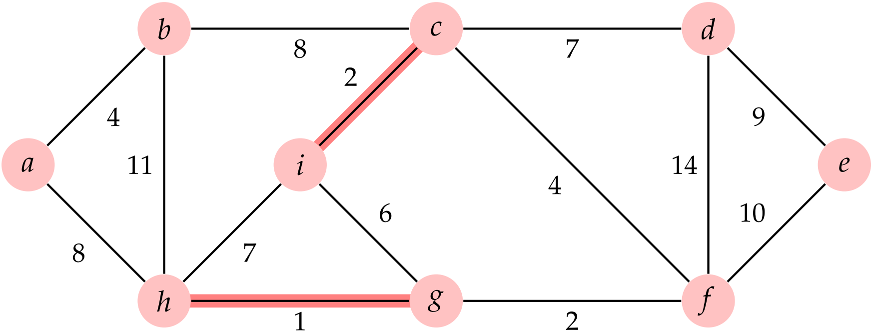
Kruskal Algorithm – Example



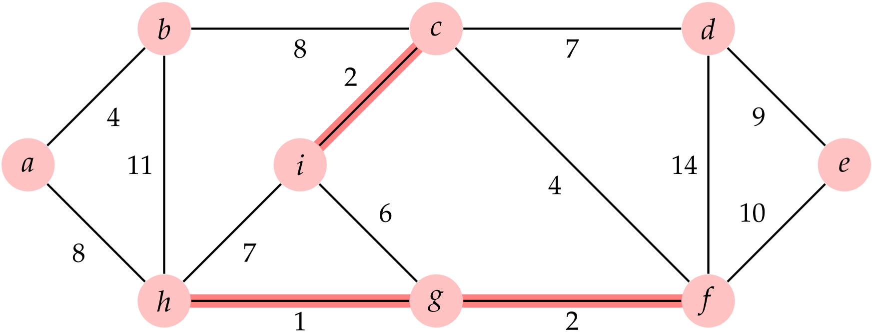
Kruskal Algorithm – Example



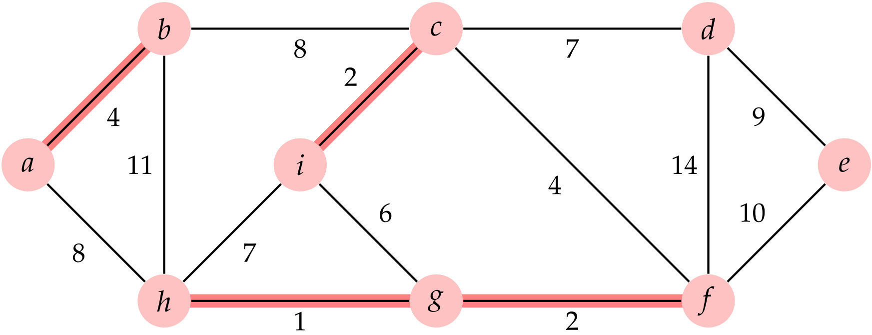
Kruskal Algorithm – Example



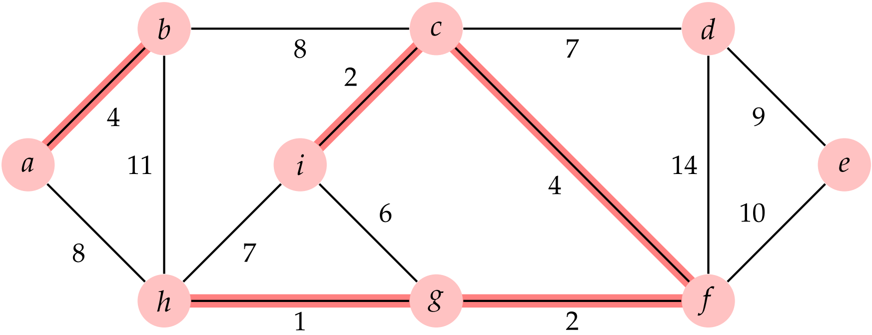
Kruskal Algorithm – Example



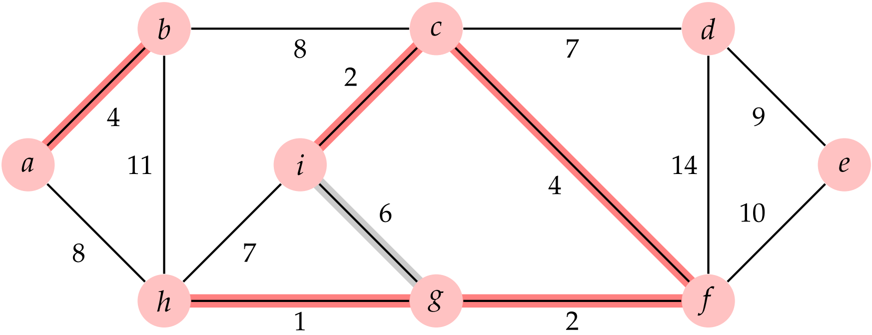
Kruskal Algorithm – Example



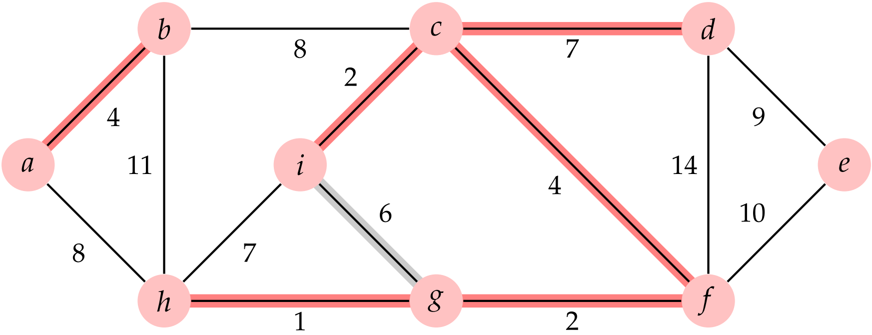
Kruskal Algorithm – Example



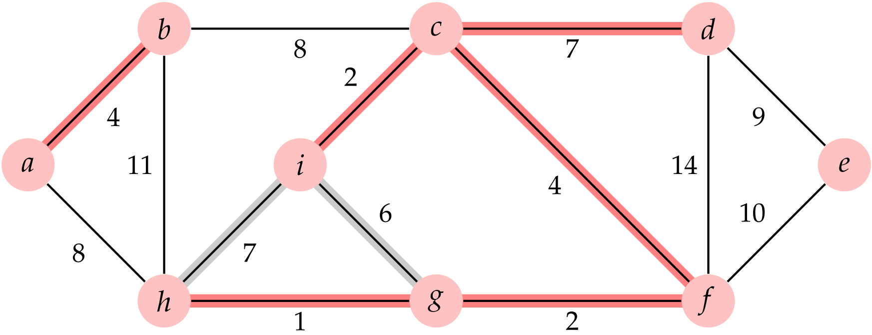
Kruskal Algorithm – Example



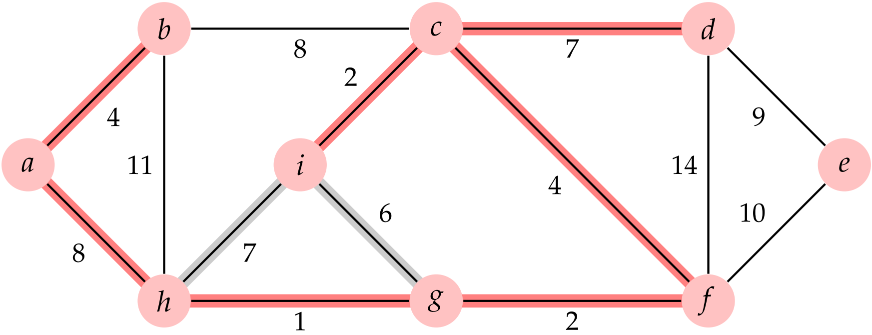
Kruskal Algorithm – Example



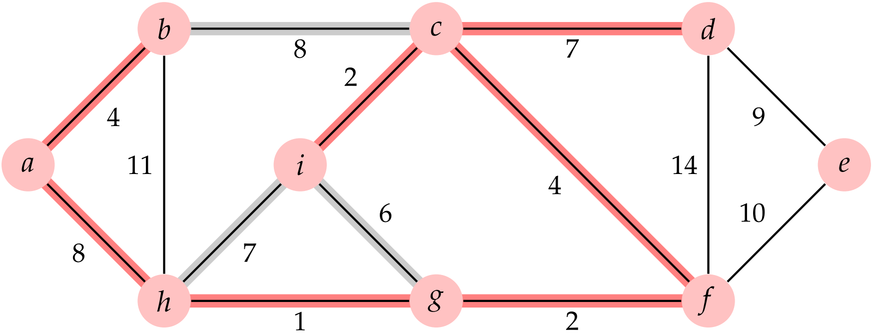
Kruskal Algorithm – Example



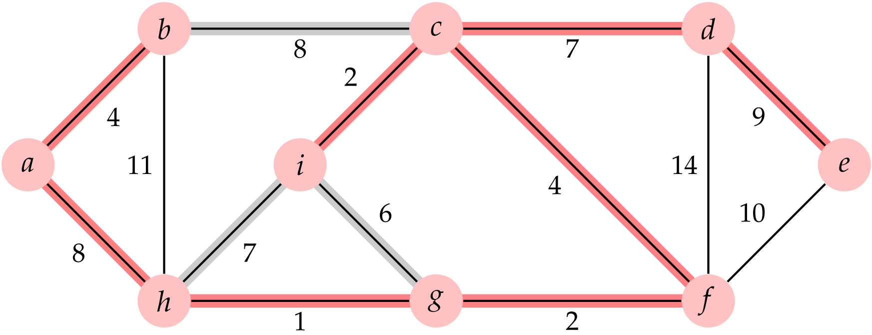
Kruskal Algorithm – Example



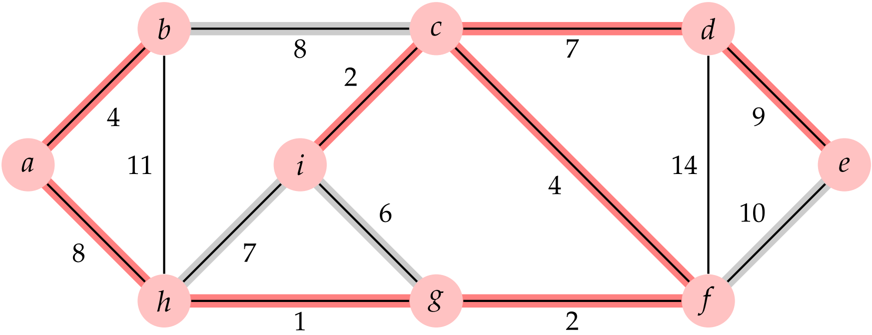
Kruskal Algorithm – Example



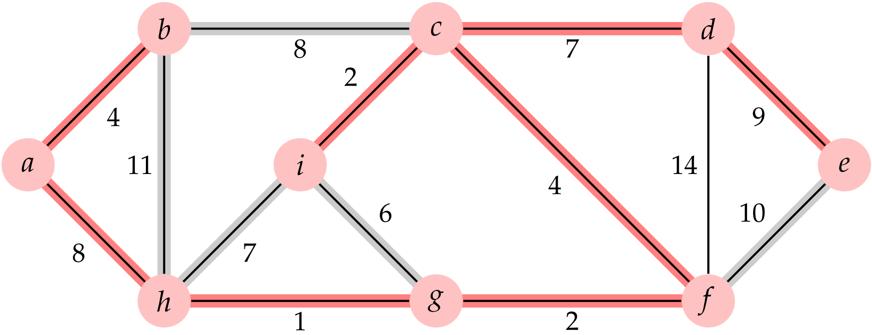
Kruskal Algorithm – Example



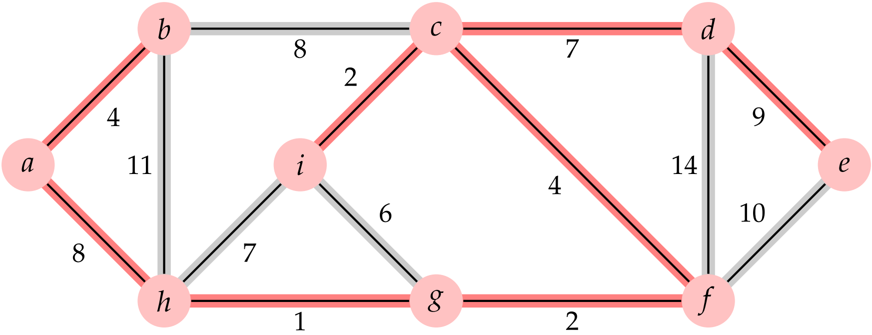
Kruskal Algorithm – Example



Kruskal Algorithm – Example



Kruskal Algorithm – Example



Prim Algorithm

Min-Priority Queue

- Data structure for maintaining a set of elements, each with an associated key (priority)
- Duality with max-priority queue
- Use: to represent an dynamic set of vertices with given priorities

Min-Priority Queue

- Data structure for maintaining a set of elements, each with an associated key (priority)
- Duality with max-priority queue
- Use: to represent an dynamic set of vertices with given priorities

Operations

- $\textsc{Insert}(Q, v)$ inserts vertex v into queue Q ($Q = Q \cup \{v\}$).
- $\textsc{Extract-Min}(Q)$ removes and returns the element of Q with the smallest key.
- $\textsc{Decrease-Key}(Q, v, k)$ decreases key of vertex v to new value k.

Min-Priority Queue

- Data structure for maintaining a set of elements, each with an associated key (priority)
- Duality with max-priority queue
- Use: to represent an dynamic set of vertices with given priorities

Operations

- $\text{INSERT}(Q, v)$ inserts vertex v into queue Q ($Q = Q \cup \{v\}$).
- $\text{EXTRACT-MIN}(Q)$ removes and returns the element of Q with the smallest key.
- $\text{DECREASE-KEY}(Q, v, k)$ decreases key of vertex v to new value k.

Implementation (Data structure)

- Binary heap in array $A[1..n]$ with $A[\text{PARENT}(i)] \leq A[i]$ (each operation: $O(\log n)$)
- Fibonacci heap (DECREASE-KEY only $O(1)$)

Prim algorithm

```
PRIM-MST(G, w, r)
for each vertex u ∈ V
     do key[u] ← ∞
        π[u] ← NIL
key[r] ← 0
Q ← V
while Q ≠ ∅
     do u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            do if v ∈ Q and w(u, v) < key[v]
                  then π[v] ← u
                       DECREASE-KEY(Q, v, w(u, v))
```

Invariant:

- $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
- If v belongs to a MST, then $v \in V - Q$.
- For all $v \in Q$, if $\pi[v] \neq$ NIL, then $key[v] < \infty$ and $key[v]$ is the weight of light edge $(v, \pi[v])$ that connects v to some vertex in $V - Q$.

Prim algorithm – Time Complexity (Binary Heap)

```
PRIM-MST(G, w, r)
1  for each vertex u ∈ V
2      do key[u] ← ∞
3         π[u] ← NIL
4  key[r] ← 0
5  Q ← V
6  while Q ≠ ∅
7      do u ← EXTRACT-MIN(Q)
8         for each v ∈ Adj[u]
9             do if v ∈ Q and w(u, v) < key[v]
10                  then π[v] ← u
11                       DECREASE-KEY(Q, v, w(u, v))
```

- Lines 1-5: $O(n)$ (no heapify necessary).

Prim algorithm – Time Complexity (Binary Heap)

```
PRIM-MST(G, w, r)
1  for each vertex u ∈ V
2      do key[u] ← ∞
3         π[u] ← NIL
4  key[r] ← 0
5  Q ← V
6  while Q ≠ ∅
7      do u ← EXTRACT-MIN(Q)
8         for each v ∈ Adj[u]
9             do if v ∈ Q and w(u, v) < key[v]
10                then π[v] ← u
11                     DECREASE-KEY(Q, v, w(u, v))
```

- Lines 1-5: $O(n)$ (no heapify necessary).
- **while** iterates n-times and each EXTRACT-MIN takes $O(\log n)$, so the total complexity of all calls of EXTRACT-MIN is $O(n \log n)$.

Prim algorithm – Time Complexity (Binary Heap)

```
PRIM-MST(G, w, r)
1  for each vertex u ∈ V
2      do key[u] ← ∞
3         π[u] ← NIL
4  key[r] ← 0
5  Q ← V
6  while Q ≠ ∅
7      do u ← EXTRACT-MIN(Q)
8         for each v ∈ Adj[u]
9             do if v ∈ Q and w(u, v) < key[v]
10                then π[v] ← u
11                     DECREASE-KEY(Q, v, w(u, v))
```

- Lines 1-5: $O(n)$ (no heapify necessary).
- **while** iterates n-times and each EXTRACT-MIN takes $O(\log n)$, so the total complexity of all calls of EXTRACT-MIN is $O(n \log n)$.
- **for** iterates $O(m)$-times (in total), since the sum of length of all adjacency lists is $2m$.
- Line 9 can be done in $O(1)$. Why?

Prim algorithm – Time Complexity (Binary Heap)

```
PRIM-MST(G, w, r)
1  for each vertex u ∈ V
2       do key[u] ← ∞
3          π[u] ← NIL
4  key[r] ← 0
5  Q ← V
6  while Q ≠ ∅
7       do u ← EXTRACT-MIN(Q)
8          for each v ∈ Adj[u]
9              do if v ∈ Q and w(u, v) < key[v]
10                   then π[v] ← u
11                        DECREASE-KEY(Q, v, w(u, v))
```

- Lines 1-5: $O(n)$ (no heapify necessary).
- **while** iterates n-times and each $\textsc{Extract-Min}$ takes $O(\log n)$, so the total complexity of all calls of $\textsc{Extract-Min}$ is $O(n \log n)$.
- **for** iterates $O(m)$-times (in total), since the sum of length of all adjacency lists is $2m$.
- Line 9 can be done in $O(1)$. Why?
- Line 11 takes $O(\log n)$.

Prim algorithm – Time Complexity (Binary Heap)

PRIM-MST(G, w, r)
1 **for** each vertex $u \in V$
2   **do** $key[u] \leftarrow \infty$
3         $\pi[u] \leftarrow$ NIL
4 $key[r] \leftarrow 0$
5 $Q \leftarrow V$
6 **while** $Q \neq \emptyset$
7   **do** $u \leftarrow$ EXTRACT-MIN(Q)
8   **for** each $v \in Adj[u]$
9   **do if** $v \in Q$ and $w(u, v) < key[v]$
10   **then** $\pi[v] \leftarrow u$
11   DECREASE-KEY$(Q, v, w(u, v))$

- Lines 1-5: $O(n)$ (no heapify necessary).
- **while** iterates n-times and each EXTRACT-MIN takes $O(\log n)$, so the total complexity of all calls of EXTRACT-MIN is $O(n \log n)$.
- **for** iterates $O(m)$-times (in total), since the sum of length of all adjacency lists is $2m$.
- Line 9 can be done in $O(1)$. Why?
- Line 11 takes $O(\log n)$.
- In total, $O(n \log n + m \log n) = O(m \log n)$.

Prim Algorithm – Time Complexity

Implementation of Q by Fibonacci heap:

- EXTRACT-MIN operation takes $O(\log n)$ amortized time.
- DECREASE-KEY operation takes only $O(1)$ amortized time.

Prim Algorithm – Time Complexity

Implementation of Q by Fibonacci heap:

- EXTRACT-MIN operation takes $O(\log n)$ amortized time.
- DECREASE-KEY operation takes only $O(1)$ amortized time.
- Together, we have $O(m + n \log n)$.

Prim Algorithm – Example

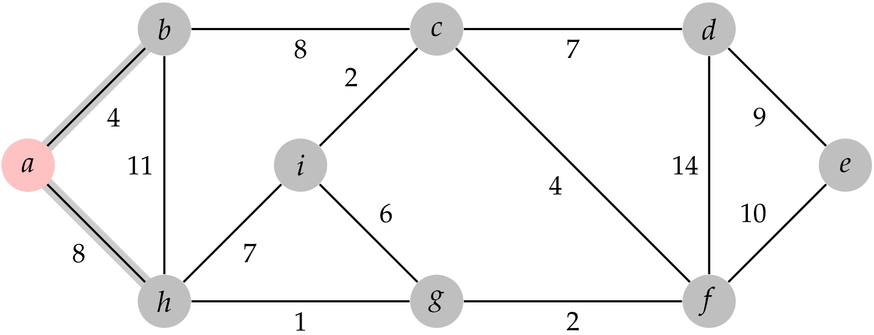


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

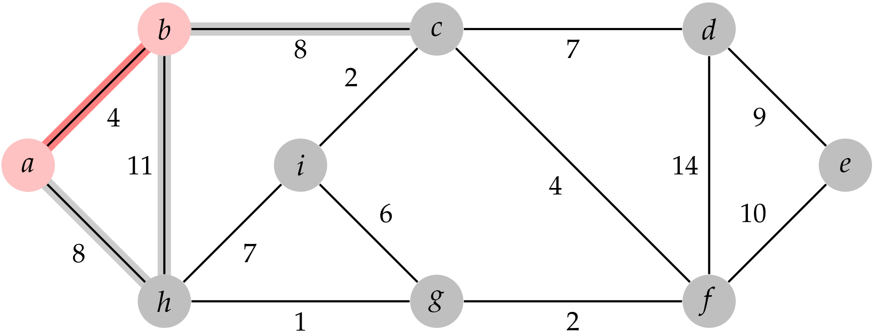


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

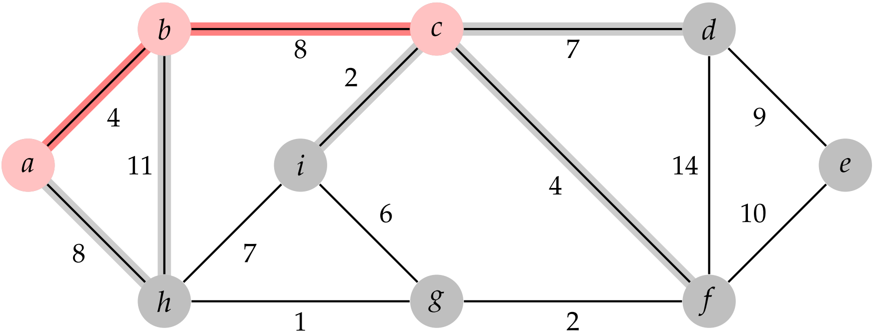


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

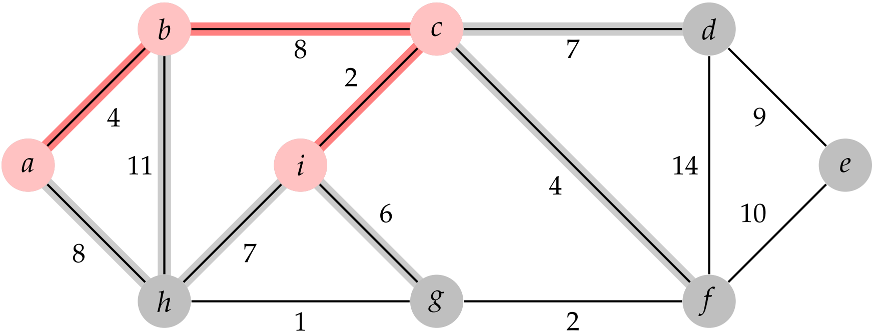


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

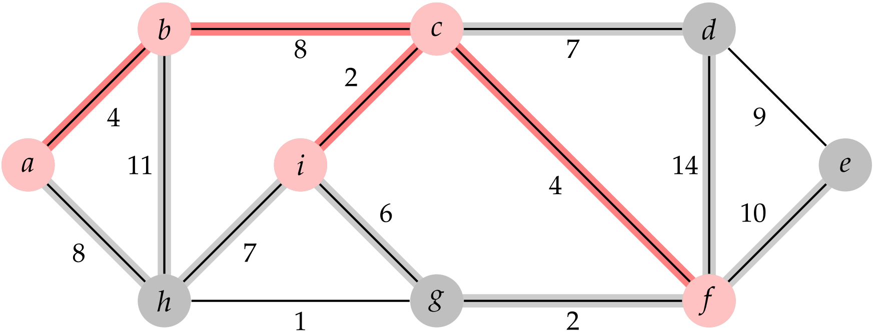


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

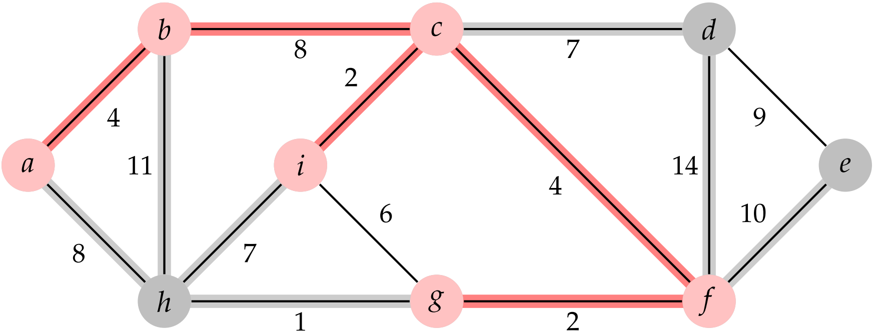


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

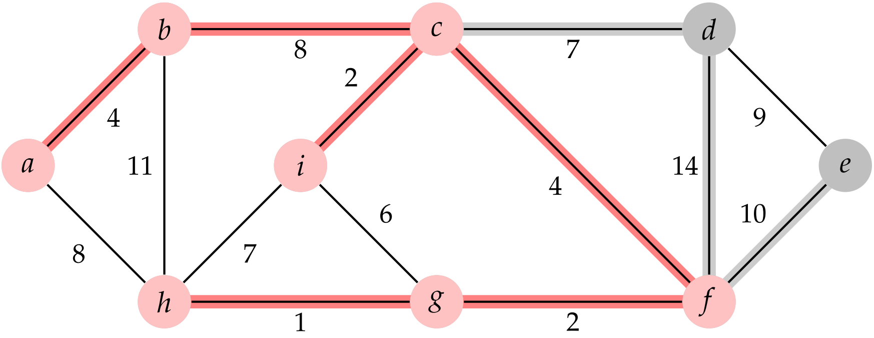


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

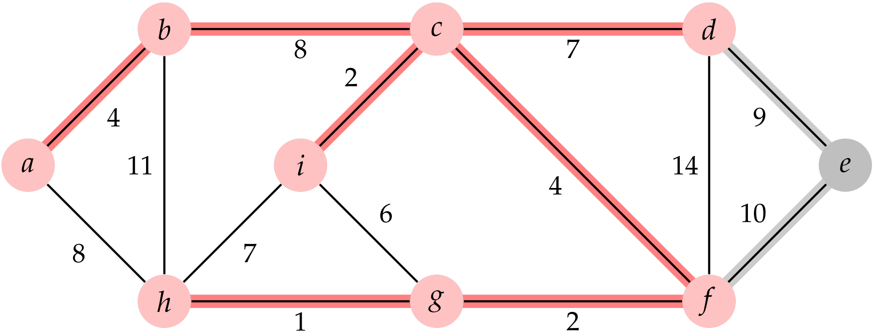


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Prim Algorithm – Example

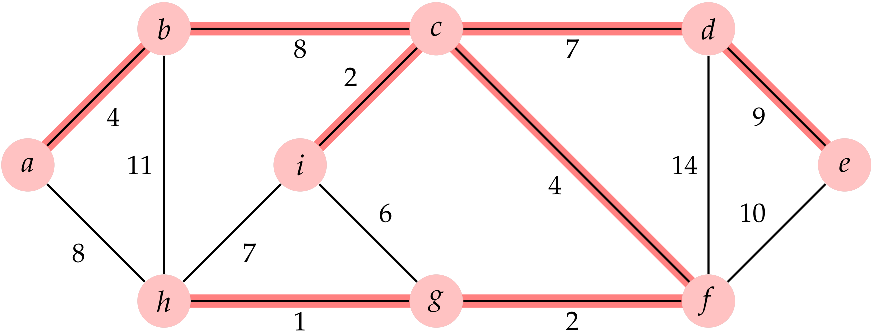


Figure: Gray edges crosses the cut $(V - Q, Q)$.

Exercises

1. Show that for each MST T of G, there is a way to sort the edges of G in Kruskal's algorithm so that it returns T.

2. Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(n^2)$ time.

Single-Source Shortest Paths

Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

- The weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

- The weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The shortest-path weight from u to v is

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- A shortest path from u to v is any path p from u to v with $w(p) = \delta(u,v)$.

Shortest Paths – Variants

- Single-source shortest-paths problem
- Single-destination shortest-paths problem – by reversing the direction of each edge
- Single-pair shortest-path problem – is there faster solution?
- All-pairs shortest-paths problem – single-source from each vertex or faster?

Subpaths of Shortest Paths

Lemma 17.
Let $G = (V, E)$ *be directed graph with weight function* $w : E \to \mathbb{R}$*. Let* $p = \langle v_1, v_2, \ldots, v_k \rangle$ *be a shortest path from* v_1 *to* v_k*.*
For any $1 \leq i \leq j \leq k$*, let* $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ *be the subpath of* p *from* v_i *to* v_j*.*
Then, p_{ij} *is a shortest path from* v_i *to* v_j*.*

Proof.

□

Subpaths of Shortest Paths

Lemma 17.
Let $G = (V, E)$ be directed graph with weight function $w : E \to \mathbb{R}$. Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from v_1 to v_k.
For any $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from v_i to v_j.
Then, p_{ij} is a shortest path from v_i to v_j.

Proof.

- p is $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, where $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.

□

Subpaths of Shortest Paths

Lemma 17.
Let $G = (V, E)$ be directed graph with weight function $w : E \to \mathbb{R}$. Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from v_1 to v_k.
For any $1 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from v_i to v_j.
Then, p_{ij} is a shortest path from v_i to v_j.

Proof.

- p is $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, where $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.
- Assume that there is p'_{ij} from v_i to v_j with $w(p'_{ij}) < w(p_{ij})$.

□

Subpaths of Shortest Paths

Lemma 17.
Let $G = (V, E)$ be directed graph with weight function $w : E \to \mathbb{R}$. Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from v_1 to v_k.
For any $1 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from v_i to v_j.
Then, p_{ij} is a shortest path from v_i to v_j.

Proof.

- p is $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, where $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.
- Assume that there is p'_{ij} from v_i to v_j with $w(p'_{ij}) < w(p_{ij})$.
- Then, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, where $w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$. Contradiction.

□

Negative-weight edges

- If G contains no negative-weight cycles reachable from the source s, then for all $v \in V$, $\delta(s, v)$ remains well defined (even if negative).

Negative-weight edges

- If G contains no negative-weight cycles reachable from the source s, then for all $v \in V$, $\delta(s, v)$ remains well defined (even if negative).

- If G contains a negative-weight cycle reachable from s, δ is not well defined – repeating traverse of the negative-weight cycle.
- If there is negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

Negative-weight edges

- If G contains no negative-weight cycles reachable from the source s, then for all $v \in V$, $\delta(s, v)$ remains well defined (even if negative).

- If G contains a negative-weight cycle reachable from s, δ is not well defined – repeating traverse of the negative-weight cycle.
- If there is negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

- Note: There is always the shortest simple path, but not path. The algorithms work with paths $\Rightarrow$ problem.

Representing Shortest Paths

- Let $G = (V, E)$ be a graph.
- $\pi[v]$ is set to a predecessor to v; that is, a vertex or NIL.
- Use procedure PRINT-PATH(G, s, v) to print the path from s to v stored in π

Representing Shortest Paths

- Let $G = (V, E)$ be a graph.
- $\pi[v]$ is set to a predecessor to v; that is, a vertex or NIL.
- Use procedure PRINT-PATH(G, s, v) to print the path from s to v stored in π

- Predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by π
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$

Representing Shortest Paths

- Let $G = (V, E)$ be a graph.
- $\pi[v]$ is set to a predecessor to v; that is, a vertex or NIL.
- Use procedure $\text{PRINT-PATH}(G, s, v)$ to print the path from s to v stored in π

- Predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by π
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$

- After the algorithm is finished, G_π is a shortest-paths tree rooted at s containing shortest paths from s to all other reachable vertices.

Shortest paths are not necessarily unique – Example

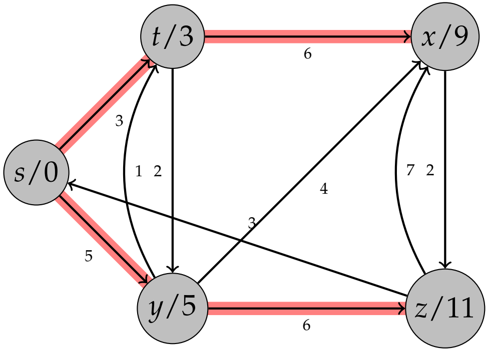


Figure: Shortest paths.

Shortest paths are not necessarily unique – Example

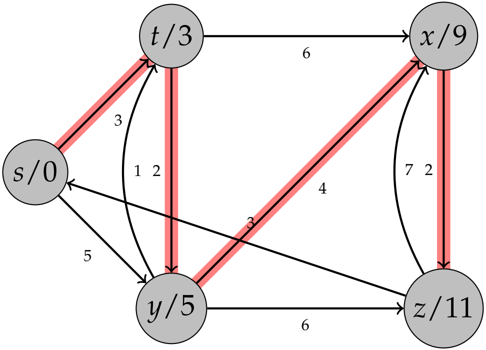


Figure: Shortest paths.

Relaxation

- $d[v]$ – shortest-path estimate (upper bound of weight)

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in V$
  do $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$

- Time complexity: $\Theta(n)$.

Relaxation

- $d[v]$ – shortest-path estimate (upper bound of weight)

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

- Time complexity: $\Theta(n)$.

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

Bellman-Ford Algorithm

Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to n − 1
    do for each edge (u, v) ∈ E
           do RELAX(u, v, w)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
          then return FALSE
return TRUE
```

- If it returns FALSE, G contains negative-weight cycles reachable from s.
- If it returns TRUE, π contains the shortest paths.

Bellman-Ford – Example

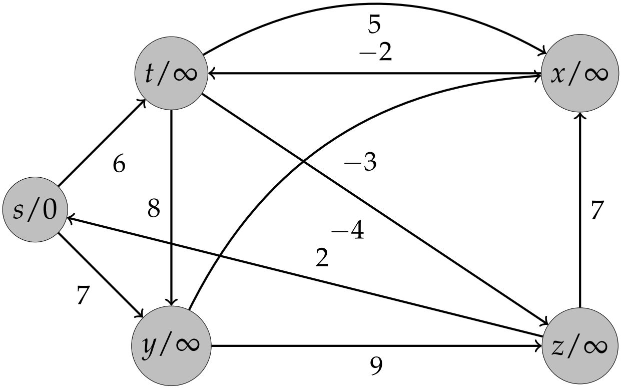


Figure: Computation by Bellman-Ford Algorithm.

- If $(u, v) \in E$ is highlighted, then $\pi[v] = u$.
- Edges are relaxed in the following order: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Bellman-Ford – Example

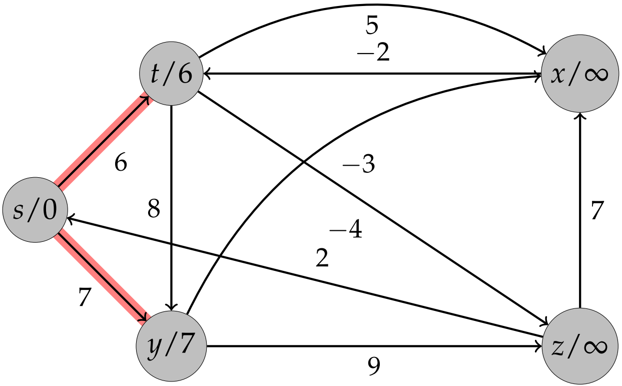


Figure: Computation by Bellman-Ford Algorithm.

- If $(u, v) \in E$ is highlighted, then $\pi[v] = u$.
- Edges are relaxed in the following order: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Bellman-Ford – Example

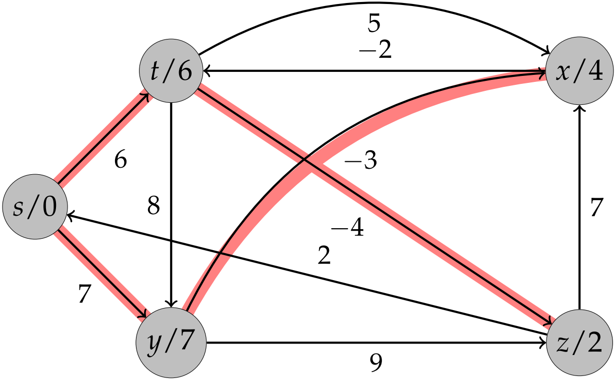


Figure: Computation by Bellman-Ford Algorithm.

- If $(u, v) \in E$ is highlighted, then $\pi[v] = u$.
- Edges are relaxed in the following order: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Bellman-Ford – Example

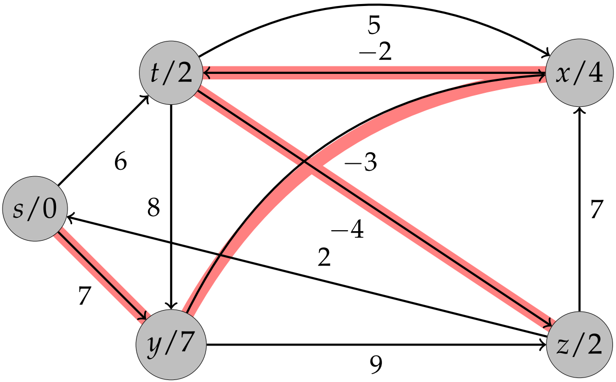


Figure: Computation by Bellman-Ford Algorithm.

- If $(u, v) \in E$ is highlighted, then $\pi[v] = u$.
- Edges are relaxed in the following order: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Bellman-Ford – Example

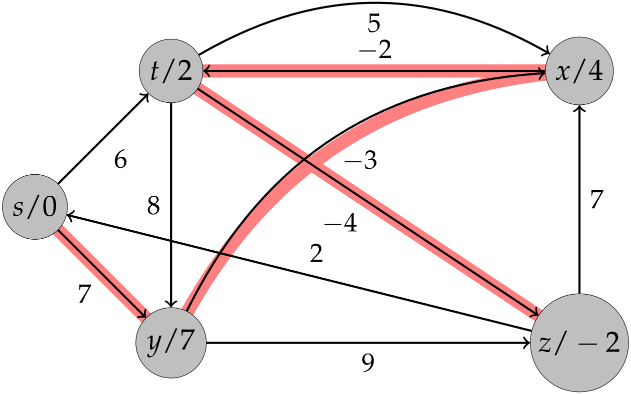


Figure: Computation by Bellman-Ford Algorithm.

- If $(u,v) \in E$ is highlighted, then $\pi[v] = u$.
- Edges are relaxed in the following order: $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Bellman-Ford Algorithm – Time Complexity

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to n − 1
3       do for each edge (u, v) ∈ E
4               do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d[u] + w(u, v)
7             then return FALSE
8  return TRUE
```

- Line 1 takes $\Theta(n)$.

Bellman-Ford Algorithm – Time Complexity

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to n − 1
3      do for each edge (u, v) ∈ E
4              do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6      do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

- Line 1 takes $\Theta(n)$.
- Lines 2-4 take $(n-1)$-times $\Theta(m)$.

Bellman-Ford Algorithm – Time Complexity

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to n − 1
3       do for each edge (u, v) ∈ E
4              do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

- Line 1 takes $\Theta(n)$.
- Lines 2-4 take $(n-1)$-times $\Theta(m)$.
- Lines 5-7 take $O(m)$.

Bellman-Ford Algorithm – Time Complexity

```
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i ← 1 to n − 1
3     do for each edge (u, v) ∈ E
4            do RELAX(u, v, w)
5 for each edge (u, v) ∈ E
6     do if d[v] > d[u] + w(u, v)
7          then return FALSE
8 return TRUE
```

- Line 1 takes $\Theta(n)$.
- Lines 2-4 take $(n-1)$-times $\Theta(m)$.
- Lines 5-7 take $O(m)$.
- In total, $\Theta(mn)$.

Bellman-Ford Algorithm – Correctness

Lemma 18.
Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n - 1$ iterations of **for***-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Bellman-Ford Algorithm – Correctness

Lemma 18.
Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n - 1$ iterations of **for***-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Proof.

- Let $v \in V$ be reachable from s.

Bellman-Ford Algorithm – Correctness

Lemma 18.
Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n - 1$ iterations of **for***-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Proof.

- Let $v \in V$ be reachable from s.
- Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v; $s = v_0$ and $v = v_k$.

Bellman-Ford Algorithm – Correctness

Lemma 18.

*Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n - 1$ iterations of **for**-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Proof.

- Let $v \in V$ be reachable from s.
- Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v; $s = v_0$ and $v = v_k$.
- p contains at most $n - 1$ edges, so $k \le n - 1$.

Bellman-Ford Algorithm – Correctness

Lemma 18.
*Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n - 1$ iterations of **for**-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Proof.

- Let $v \in V$ be reachable from s.
- Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v; $s = v_0$ and $v = v_k$.
- p contains at most $n - 1$ edges, so $k \leq n - 1$.
- Each of $n - 1$ iterations on lines 2-4 relaxes all m edges.

Bellman-Ford Algorithm – Correctness

Lemma 18.

*Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n-1$ iterations of **for**-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Proof.

- Let $v \in V$ be reachable from s.
- Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v; $s = v_0$ and $v = v_k$.
- p contains at most $n-1$ edges, so $k \leq n-1$.
- Each of $n-1$ iterations on lines 2-4 relaxes all m edges.
- Amongst the relaxed edges in i-th iteration, there is edge (v_{i-1}, v_i) and then $d[v_i] = \delta(s, v_i)$. (Prove by induction.)

Bellman-Ford Algorithm – Correctness

Lemma 18.

Let $G = (V, E)$ be weighted digraf with source s and weight function $w : E \to \mathbb{R}$. Assume that G contains no negative cycle reachable from s. Then after $n-1$ iterations of **for***-cycle (lines 2-4), $d[v] = \delta(s, v)$ for all $v \in V$ reachable from s.* **Note:** $d[v] = \infty$ *implies* $s \not\rightsquigarrow v$.

Proof.

- Let $v \in V$ be reachable from s.
- Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v; $s = v_0$ and $v = v_k$.
- p contains at most $n-1$ edges, so $k \leq n-1$.
- Each of $n-1$ iterations on lines 2-4 relaxes all m edges.
- Amongst the relaxed edges in i-th iteration, there is edge (v_{i-1}, v_i) and then $d[v_i] = \delta(s, v_i)$. (Prove by induction.)
- Therefore, after k-th iteration, $d[v_k] = \delta(s, v_k)$.

Bellman-Ford Algorithm – Correctness

Theorem 19 (Correctness I).

- *If* G *contains* *no negative cycle* *reachable from* s*, the algorithm returns* $\textsc{True}$ *and* $d[v] = \delta(s, v)$ *for all* $v \in V$*.*

Bellman-Ford Algorithm – Correctness

Theorem 19 (Correctness I).

- *If G contains no negative cycle reachable from s, the algorithm returns $\textsc{True}$ and $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof.

- Let G contains no negative cycle reachable from s.

□

Bellman-Ford Algorithm – Correctness

Theorem 19 (Correctness I).

- *If G contains no negative cycle reachable from s, the algorithm returns* $\textsc{True}$ *and $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof.

- Let G contains no negative cycle reachable from s.
- When the algorithms is finished, $d[v] = \delta(s, v)$ for all $v \in V$ (Lemma 18)

□

Bellman-Ford Algorithm – Correctness

Theorem 19 (Correctness I).

- *If G contains no negative cycle reachable from s, the algorithm returns* $\textsc{True}$ *and $d[v] = \delta(s,v)$ for all $v \in V$.*

Proof.

- Let G contains no negative cycle reachable from s.
- When the algorithms is finished, $d[v] = \delta(s,v)$ for all $v \in V$ (Lemma 18)
- Moreover, $d[v] = \delta(s,v) \leq \delta(s,u) + w(u,v) = d[u] + w(u,v)$. So the algorithm returns $\textsc{True}$.

□

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE*.*

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE.

Proof.

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$, be negative-weight cycle reachable from s.

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE.

Proof.

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$, be negative-weight cycle reachable from s.
- Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE.

Proof.

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$, be negative-weight cycle reachable from s.
- Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- By contradiction – alg. returns TRUE, so $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE.

Proof.

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$, be negative-weight cycle reachable from s.
- Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- By contradiction – alg. returns TRUE, so $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- But then $\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE*.*

Proof.

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$, be negative-weight cycle reachable from s.
- Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- By contradiction – alg. returns TRUE, so $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- But then $\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$.
- Since $v_0 = v_k$, we have $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$.

Bellman-Ford Algorithm – Correctness

Theorem 20 (Correctness II).

- *If G contains a negative-weight cycle reachable from s, the algorithm returns* FALSE*.*

Proof.

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$, be negative-weight cycle reachable from s.
- Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- By contradiction – alg. returns TRUE, so $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- But then $\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$.
- Since $v_0 = v_k$, we have $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$.
- Because for $i = 1, 2, \ldots, k$ $d[v_i] < \infty$, we have $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Contradiction.

Single-Source Shortest Paths in Directed Acyclic Graphs

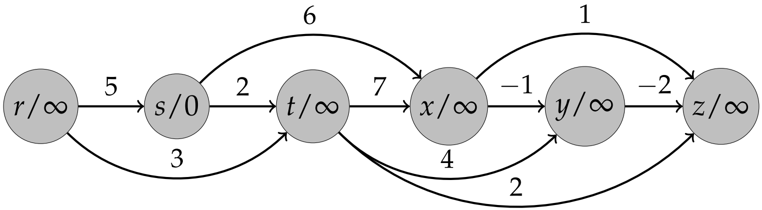
Shortest Paths in Directed Acyclic Graphs

- For DAG, there is significantly faster method than Bellman-Ford.

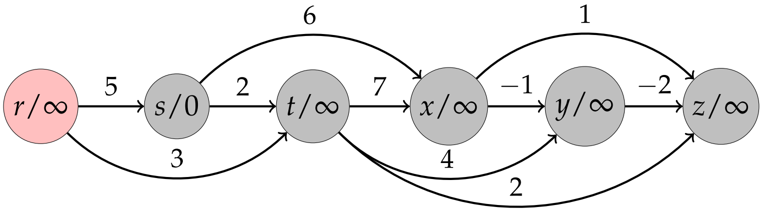
```
DAG-SHORTEST-PATHS(G, w, s)
1  Topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       do for each vertex v ∈ Adj[u]
5              do RELAX(u, v, w)
```

- Time complexity: $\Theta(n+m)$.
  - We get a topological order in $\Theta(n+m)$.
  - Line 2 takes $\Theta(n)$.
  - Lines 3-5 checks every edge exactly once; that is, the iteration is executed m-times. RELAX takes $\Theta(1)$.

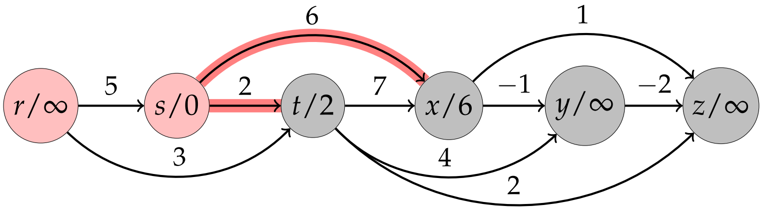
Example



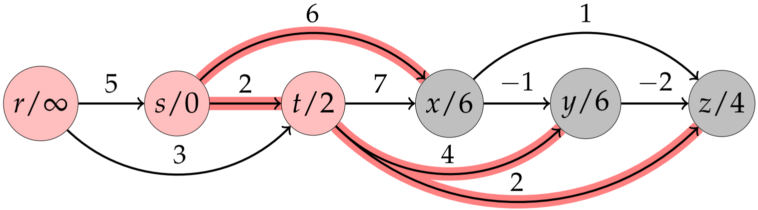
Example



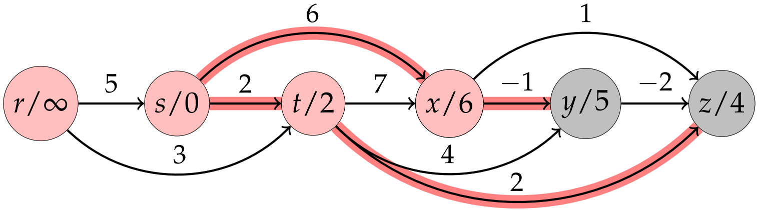
Example



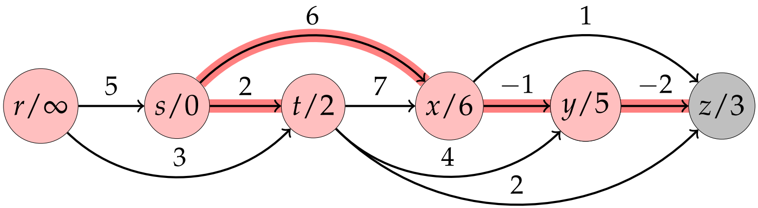
Example



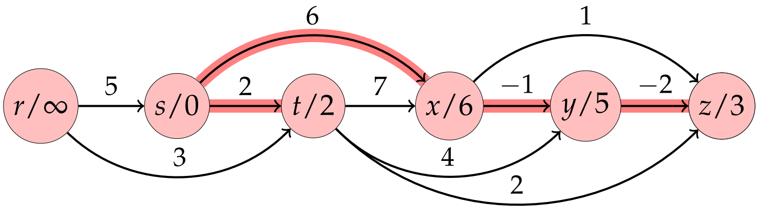
Example



Example



Example



Correctness

Theorem 21.
If a weighted, digraph $G = (V, E)$ *has source vertex* s *and no cycles, then* DAG-SHORTEST-PATHS *computes* $d[v] = \delta(s, v)$ *for all* $v \in V$.

Correctness

Theorem 21.
If a weighted, digraph $G = (V, E)$ has source vertex s and no cycles, then DAG-SHORTEST-PATHS *computes $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof.

- If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$.

□

Correctness

Theorem 21.
If a weighted, digraph $G = (V, E)$ has source vertex s and no cycles, then DAG-SHORTEST-PATHS *computes $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof.

- If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$.
- Suppose there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $s = v_0$ and $v = v_k$.

□

Correctness

Theorem 21.
If a weighted, digraph $G = (V, E)$ has source vertex s and no cycles, then DAG-SHORTEST-PATHS *computes $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof.

- If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$.
- Suppose there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $s = v_0$ and $v = v_k$.
- Because we process the vertices in topological order, we relax edges on p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.

□

Correctness

Theorem 21.
If a weighted, digraph $G = (V, E)$ has source vertex s and no cycles, then $\textsc{Dag-Shortest-Paths}$ *computes $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof.

- If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$.
- Suppose there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $s = v_0$ and $v = v_k$.
- Because we process the vertices in topological order, we relax edges on p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.
- That implies that $d[v_i] = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$.

□

Dijkstra Algorithm

Dijkstra Algorithm

- Only for weighted, directed graphs without negative edges:
- $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra Algorithm

- Only for weighted, directed graphs without negative edges:
- $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

- Can we implement it with lower time complexity than Bellman-Ford algorithm?

Dijkstra Algorithm

DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
$S \leftarrow \emptyset$
$Q \leftarrow V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        $S \leftarrow S \cup \{u\}$
        for each vertex $v \in Adj[u]$
            do RELAX(u, v, w)

- S is a set of finished vertices (their shortest distance from s is already computed).
- Q is a min-priority queue; the vertex with the lowest d-value is at the beginning of Q.

Dijkstra Algorithm – Example

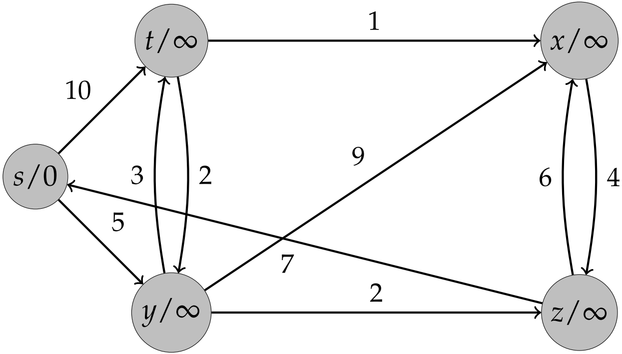


Figure: The computation by Dijkstra Algorithm. Highlighted vertices belong to set S.

Dijkstra Algorithm – Example

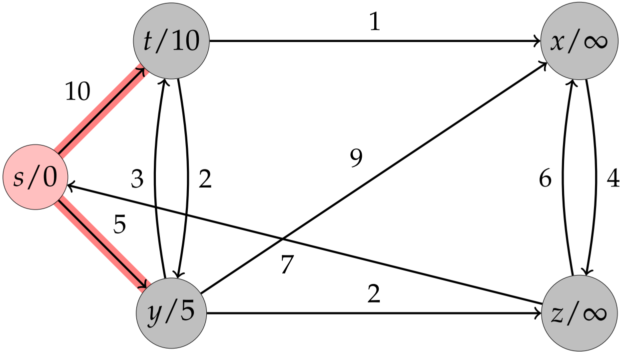


Figure: The computation by Dijkstra Algorithm. Highlighted vertices belong to set S.

Dijkstra Algorithm – Example

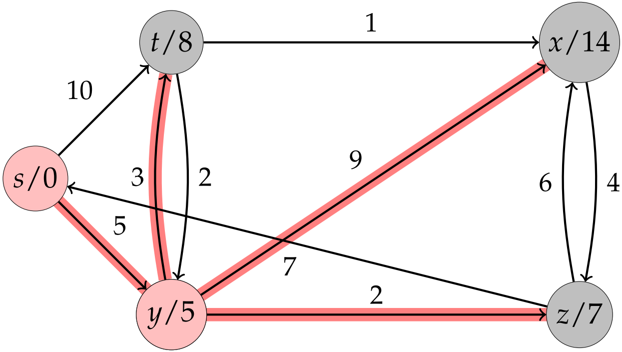


Figure: The computation by Dijkstra Algorithm. Highlighted vertices belong to set S.

Dijkstra Algorithm – Example

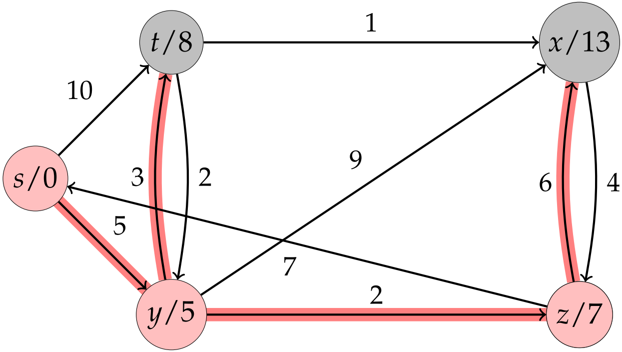


Figure: The computation by Dijkstra Algorithm. Highlighted vertices belong to set S.

Dijkstra Algorithm – Example

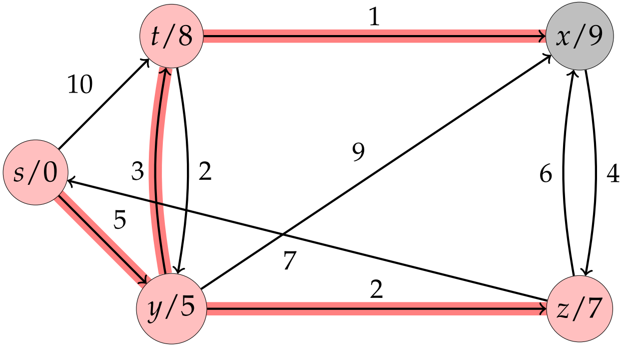


Figure: The computation by Dijkstra Algorithm. Highlighted vertices belong to set S.

Dijkstra Algorithm – Example

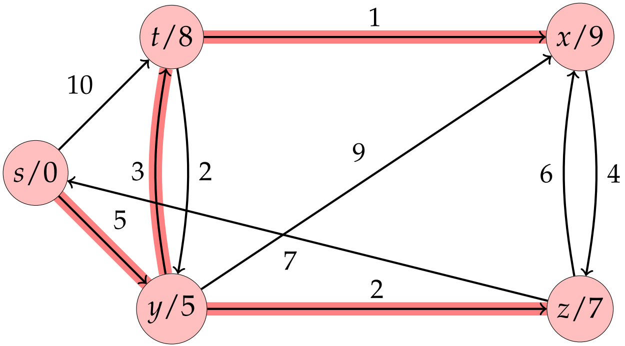


Figure: The computation by Dijkstra Algorithm. Highlighted vertices belong to set S.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ *without negative-weight edges and with source* s *finishes with* $d[v] = \delta(s, v)$ *for all* $v \in V$.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ without negative-weight edges and with source s finishes with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ *without negative-weight edges and with source* s *finishes with* $d[v] = \delta(s, v)$ *for all* $v \in V$.

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.
- It holds for $S = \emptyset$.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ *without negative-weight edges and with source* s *finishes with* $d[v] = \delta(s, v)$ *for all* $v \in V$.

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.
- It holds for $S = \emptyset$.
- Let u be first vertex such that $d[u] \neq \delta(s, u)$ in the moment of its inclusion into S.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ without negative-weight edges and with source s finishes with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.
- It holds for $S = \emptyset$.
- Let u be first vertex such that $d[u] \neq \delta(s, u)$ in the moment of its inclusion into S.
- Then, necessarily $u \neq s$, because s is included as the first into S and $d[s] = \delta(s, s) = 0$ holds in the moment of inclusion of s into S.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ *without negative-weight edges and with source* s *finishes with* $d[v] = \delta(s, v)$ *for all* $v \in V$*.*

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.
- It holds for $S = \emptyset$.
- Let u be first vertex such that $d[u] \neq \delta(s, u)$ in the moment of its inclusion into S.
- Then, necessarily $u \neq s$, because s is included as the first into S and $d[s] = \delta(s, s) = 0$ holds in the moment of inclusion of s into S.
- Since $u \neq s$, $S \neq \emptyset$ right before inclusion of u.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ without negative-weight edges and with source s finishes with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.
- It holds for $S = \emptyset$.
- Let u be first vertex such that $d[u] \neq \delta(s, u)$ in the moment of its inclusion into S.
- Then, necessarily $u \neq s$, because s is included as the first into S and $d[s] = \delta(s, s) = 0$ holds in the moment of inclusion of s into S.
- Since $u \neq s$, $S \neq \emptyset$ right before inclusion of u.
- The assumption $d[u] \neq \delta(s, u)$ implies that $s \rightsquigarrow u$ – otherwise $d[u] = \delta(s, u) = \infty$.

Correctness

Theorem 22.
Dijkstra algorithm on weighted digraph $G = (V, E)$ without negative-weight edges and with source s finishes with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof.

- Invariant: In the beginning of each **while**-iteration, $d[v] = \delta(s, v)$ for all $v \in S$.
- It holds for $S = \emptyset$.
- Let u be first vertex such that $d[u] \neq \delta(s, u)$ in the moment of its inclusion into S.
- Then, necessarily $u \neq s$, because s is included as the first into S and $d[s] = \delta(s, s) = 0$ holds in the moment of inclusion of s into S.
- Since $u \neq s$, $S \neq \emptyset$ right before inclusion of u.
- The assumption $d[u] \neq \delta(s, u)$ implies that $s \rightsquigarrow u$ – otherwise $d[u] = \delta(s, u) = \infty$.
- So there is a shortest path p from s to u.

Correctness

Part II of the Proof.

- There is a shortest path p from s to u.

□

Correctness

Part II of the Proof.

- There is a shortest path p from s to u.
- Right before inclusion of u into S, p connects vertex $s \in S$ with vertex $u \in V - S$.

□

Correctness

Part II of the Proof.

- There is a shortest path p from s to u.
- Right before inclusion of u into S, p connects vertex $s \in S$ with vertex $u \in V - S$.
- Split p as:

$$s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u\,,$$

where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.

□

Correctness

Part II of the Proof.

- There is a shortest path p from s to u.
- Right before inclusion of u into S, p connects vertex $s \in S$ with vertex $u \in V - S$.
- Split p as:

$$s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u\,,$$

  where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- By assumption, we have $d[x] = \delta(s, x)$ in the moment of inclusion of x into S.

□

Correctness

Part II of the Proof.

- There is a shortest path p from s to u.
- Right before inclusion of u into S, p connects vertex $s \in S$ with vertex $u \in V - S$.
- Split p as:

$$s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u\,,$$

  where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- By assumption, we have $d[x] = \delta(s, x)$ in the moment of inclusion of x into S.
- Since edge (x, y) was already relaxed in that moment, we have $d[y] = \delta(s, y)$ in the moment of inclusion of u into S. (Prove it!)

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.
- Since y precedes u on the shortest path from s to u and all weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$.

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.
- Since y precedes u on the shortest path from s to u and all weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$.
- Therefore, $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$.

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.
- Since y precedes u on the shortest path from s to u and all weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$.
- Therefore, $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$.
- Since both vertices $y, u \in V - S$ in the moment of dequeuing of u, it holds that $d[u] \leq d[y]$.

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.
- Since y precedes u on the shortest path from s to u and all weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$.
- Therefore, $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$.
- Since both vertices $y, u \in V - S$ in the moment of dequeuing of u, it holds that $d[u] \leq d[y]$.
- In total, $d[u] = \delta(s, u)$. Contradiction of the assumption.

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.
- Since y precedes u on the shortest path from s to u and all weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$.
- Therefore, $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$.
- Since both vertices $y, u \in V - S$ in the moment of dequeuing of u, it holds that $d[u] \leq d[y]$.
- In total, $d[u] = \delta(s, u)$. Contradiction of the assumption.
- $Q = \emptyset$ when alg. finishes. Since $Q = V - S$ (Do the reasoning!), we have $S = V$. So $d[v] = \delta(s, v)$ for all $v \in V$.

□

Correctness

Part III of the Proof.

- $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on p that belongs to $V - S$ and x is its predecessor on p.
- $d[y] = \delta(s, y)$ in the moment of inclusion of u into S.
- Since y precedes u on the shortest path from s to u and all weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$.
- Therefore, $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$.
- Since both vertices $y, u \in V - S$ in the moment of dequeuing of u, it holds that $d[u] \leq d[y]$.
- In total, $d[u] = \delta(s, u)$. Contradiction of the assumption.
- $Q = \emptyset$ when alg. finishes. Since $Q = V - S$ (Do the reasoning!), we have $S = V$. So $d[v] = \delta(s, v)$ for all $v \in V$.
- Done!. . .

□

Time Complexity of Dijkstra algorithm

Min-Priority Queue Implemented by Array

- INSERT and DECREASE-KEY take $O(1)$.
- EXTRACT-MIN takes $O(n)$ for each vertex (line 5).

Time Complexity of Dijkstra algorithm

Min-Priority Queue Implemented by Array

- INSERT and DECREASE-KEY take $O(1)$.
- EXTRACT-MIN takes $O(n)$ for each vertex (line 5).
- RELAX is repeated m-times (line 8).

Time Complexity of Dijkstra algorithm

Min-Priority Queue Implemented by Array

- INSERT and DECREASE-KEY take $O(1)$.
- EXTRACT-MIN takes $O(n)$ for each vertex (line 5).
- RELAX is repeated m-times (line 8).

- In total, $O(n^2 + m) = O(n^2)$.

Min-Priority Queue Implemented by Heaps

Time Complexity of Dijkstra algorithm

Min-Priority Queue Implemented by Array

- INSERT and DECREASE-KEY take $O(1)$.
- EXTRACT-MIN takes $O(n)$ for each vertex (line 5).
- RELAX is repeated m-times (line 8).

- In total, $O(n^2 + m) = O(n^2)$.

Min-Priority Queue Implemented by Heaps

- For sparse graphs, we get the time complexity $O(m \log n)$ using binary heap.

Time Complexity of Dijkstra algorithm

Min-Priority Queue Implemented by Array

- INSERT and DECREASE-KEY take $O(1)$.
- EXTRACT-MIN takes $O(n)$ for each vertex (line 5).
- RELAX is repeated m-times (line 8).

- In total, $O(n^2 + m) = O(n^2)$.

Min-Priority Queue Implemented by Heaps

- For sparse graphs, we get the time complexity $O(m \log n)$ using binary heap.

- In general, using Fibonacci heap we get the time complexity $O(n \log n + m)$.

Exercises

1. Modify the Bellman-Ford algorithm so that it sets $d[v]$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source s to v.
2. A critical path is a *longest* path through the DAG. Modify the DAG-SHORTEST-PATHS procedure to find a critical path in the given DAG.
3. Give a simple example of a digraph with negative-weight edge(s) for which Dijkstra's algorithm produces incorrect answers. Why?

All-Pairs Shortest Paths

All-Pairs Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

All-Pairs Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

- Trivial approach: n-times use of an algorithm for shortest path problem from one source vertex to all other vertices.
- Dijkstra algorithm (n-times): Time $O(n^3 + nm) = O(n^3)$ for array, or $O(n^2 \log n + nm)$ for Fibonacci heap.

All-Pairs Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

- Trivial approach: n-times use of an algorithm for shortest path problem from one source vertex to all other vertices.
- Dijkstra algorithm (n-times): Time $O(n^3 + nm) = O(n^3)$ for array, or $O(n^2 \log n + nm)$ for Fibonacci heap.
- If we permit negative-weight edges, we need n-times Bellman-Ford algorithm $\Rightarrow$ time $O(n^2 m)$ resulting into $O(n^4)$ for dense graphs.

All-Pairs Shortest Paths

- Given weighted directed graph $G = (V, E)$ and
- weight function $w : E \to \mathbb{R}$.

- Trivial approach: n-times use of an algorithm for shortest path problem from one source vertex to all other vertices.
- Dijkstra algorithm (n-times): Time $O(n^3 + nm) = O(n^3)$ for array, or $O(n^2 \log n + nm)$ for Fibonacci heap.
- If we permit negative-weight edges, we need n-times Bellman-Ford algorithm $\Rightarrow$ time $O(n^2 m)$ resulting into $O(n^4)$ for dense graphs.

- Let us examine methods based on dynamic programming...

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- Negative-weight edges allowed.

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- Negative-weight edges allowed.
- Restriction: No negative-weight cycles.

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- Negative-weight edges allowed.
- Restriction: No negative-weight cycles.

- Result stored in matrix $D = (d_{ij})$, where $d_{ij} = \delta(i,j)$ after the end of algorithm.

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- Negative-weight edges allowed.
- Restriction: No negative-weight cycles.

- Result stored in matrix $D = (d_{ij})$, where $d_{ij} = \delta(i,j)$ after the end of algorithm.
- Predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- Negative-weight edges allowed.
- Restriction: No negative-weight cycles.

- Result stored in matrix $D = (d_{ij})$, where $d_{ij} = \delta(i,j)$ after the end of algorithm.
- Predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is
  1. NIL, if $i = j$ or there is no path from i to j,

Adjacency-matrix Representation

- This time, we prefer to use an adjacency matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{for } i = j, \\ w(i,j) & \text{for } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{for } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- Negative-weight edges allowed.
- Restriction: No negative-weight cycles.

- Result stored in matrix $D = (d_{ij})$, where $d_{ij} = \delta(i,j)$ after the end of algorithm.
- Predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is
  1. NIL, if $i = j$ or there is no path from i to j,
  2. predecessor of j on some shortest path from i.

Printing All-Pairs Shortest Paths

```
PRINT-ALL-SHORTEST-PATH(Π, i, j)
if i = j
   then print i
   else if π_ij = NIL
           then print "No path from " i " to " j " exists!"
           else PRINT-ALL-SHORTEST-PATH(Π, i, π_ij)
                print j
```

Matrix Multiplication

Matrix Multiplication – Structure of Shortest Paths

- Representation – adjacency matrix $W = (w_{ij})$.

Matrix Multiplication – Structure of Shortest Paths

- Representation – adjacency matrix $W = (w_{ij})$.
- Let p be a shortest path from i to j that has m' edges.

Matrix Multiplication – Structure of Shortest Paths

- Representation – adjacency matrix $W = (w_{ij})$.
- Let p be a shortest path from i to j that has m' edges.
- If p has no negative-weight cycle, then $m' < \infty$.

Matrix Multiplication – Structure of Shortest Paths

- Representation – adjacency matrix $W = (w_{ij})$.
- Let p be a shortest path from i to j that has m' edges.
- If p has no negative-weight cycle, then $m' < \infty$.
- For $i = j$ is $m' = 0$ and $w_{ij} = \delta(i, j) = 0$.

Matrix Multiplication – Structure of Shortest Paths

- Representation – adjacency matrix $W = (w_{ij})$.
- Let p be a shortest path from i to j that has m' edges.
- If p has no negative-weight cycle, then $m' < \infty$.
- For $i = j$ is $m' = 0$ and $w_{ij} = \delta(i, j) = 0$.
- For $i \neq j$ we split path p as:

$$i \overset{p'}{\leadsto} k \to j\,,$$

  where p' has $m' - 1$ edges.

Matrix Multiplication – Structure of Shortest Paths

- Representation – adjacency matrix $W = (w_{ij})$.
- Let p be a shortest path from i to j that has m' edges.
- If p has no negative-weight cycle, then $m' < \infty$.
- For $i = j$ is $m' = 0$ and $w_{ij} = \delta(i,j) = 0$.
- For $i \neq j$ we split path p as:

  $$i \overset{p'}{\leadsto} k \to j\,,$$

  where p' has $m' - 1$ edges.
- p' is a shortest path from i to k – HOMEWORK – so $\delta(i,j) = \delta(i,k) + w_{kj}$.

Matrix Multiplication – Recursion

- Let $l_{ij}^{(m)}$ be the minimal weight of any path from i to j that contains at most m edges.

Matrix Multiplication – Recursion

- Let $l_{ij}^{(m)}$ be the minimal weight of any path from i to j that contains at most m edges.
- $m = 0$ if and only if $i = j$. Thus, $l_{ij}^{(0)} = \begin{cases} 0 & \text{for } i = j \\ \infty & \text{for } i \neq j \end{cases}$

Matrix Multiplication – Recursion

- Let $l_{ij}^{(m)}$ be the minimal weight of any path from i to j that contains at most m edges.
- $m = 0$ if and only if $i = j$. Thus, $l_{ij}^{(0)} = \begin{cases} 0 & \text{for } i = j \\ \infty & \text{for } i \neq j \end{cases}$
- $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\{l_{ik}^{(m-1)} + w_{kj}\}) = \min_{1 \leq k \leq n}\{l_{ik}^{(m-1)} + w_{kj}\}.$

Matrix Multiplication – Recursion

- Let $l_{ij}^{(m)}$ be the minimal weight of any path from i to j that contains at most m edges.
- $m = 0$ if and only if $i = j$. Thus, $l_{ij}^{(0)} = \begin{cases} 0 & \text{for } i = j \\ \infty & \text{for } i \neq j \end{cases}$
- $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{l_{ik}^{(m-1)} + w_{kj}\}) = \min_{1 \leq k \leq n} \{l_{ik}^{(m-1)} + w_{kj}\}$.
- A path from i to j with no more then $n - 1$ edges, so

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots.$$

(No negative-weight cycle.)

Matrix Multiplication – Computation

- Input: matrix $W = (w_{ij})$.

Matrix Multiplication – Computation

- Input: matrix $W = (w_{ij})$.
- Compute matrices: $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where for $m = 1, 2, \ldots, n-1$,
$$L^{(m)} = (l_{ij}^{(m)}) \, .$$

Matrix Multiplication – Computation

- Input: matrix $W = (w_{ij})$.
- Compute matrices: $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where for $m = 1, 2, \ldots, n-1$,
$$L^{(m)} = (l_{ij}^{(m)}) \, .$$
- $L^{(n-1)}$, then it contains weights of shortest paths.

Matrix Multiplication – Computation

- Input: matrix $W = (w_{ij})$.
- Compute matrices: $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where for $m = 1, 2, \ldots, n-1$,
$$L^{(m)} = (l_{ij}^{(m)}) .$$
- $L^{(n-1)}$, then it contains weights of shortest paths.
- $l_{ij}^{(1)} = w_{ij}$, i.e. $L^{(1)} = W$.

Algorithm Core

EXTEND-SHORTEST-PATHS(L, W)

$n \leftarrow rows[L]$
let $L' = (l'_{ij})$ be an $n \times n$ matrix
for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
        do $l'_{ij} \leftarrow \infty$
            for $k \leftarrow 1$ **to** n
                do $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

- $rows[L]$ denotes the line number of L.
- Time complexity $\Theta(n^3)$.

All-Pairs Shortest Paths Vs. Matrix Multiplication

- Let $C = A \cdot B$, where A and B are matrices of order n.

All-Pairs Shortest Paths Vs. Matrix Multiplication

- Let $C = A \cdot B$, where A and B are matrices of order n.
- Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

All-Pairs Shortest Paths Vs. Matrix Multiplication

- Let $C = A \cdot B$, where A and B are matrices of order n.
- Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- For the comparison:

$$l_{ij}^{(m)} = \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \}$$

Find 3 differences (skip the naming and names of variables)

EXTEND-SHORTEST-PATHS(L, W)

$n \leftarrow rows[L]$
let $L' = (l'_{ij})$ be an $n \times n$ matrix
for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
        do $l'_{ij} \leftarrow \infty$
            for $k \leftarrow 1$ **to** n
                do $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

MATRIX-MULTIPLY(A, B)

$n \leftarrow rows[A]$
let $C = (c_{ij})$ be an $n \times n$ matrix
for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow 0$
            for $k \leftarrow 1$ **to** n
                do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
return C

Matrix multiplication revisited

- Notation $X \cdot Y$ represents a matrix computed by $\textsc{Extend-Shortest-Paths}(X, Y)$.

Matrix multiplication revisited

- Notation $X \cdot Y$ represents a matrix computed by $\textsc{Extend-Shortest-Paths}(X, Y)$.
- Then, we compute the whole sequence of matrices

$$
\begin{array}{ccccc}
L^{(1)} & = & L^{(0)} \cdot W & = & W \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^2 \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^3 \\
 & & \vdots & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1}
\end{array}
$$

where W^{n-1} contains the weights of shortest paths.

Slow method

```
SLOW-ALL-SHORTEST-PATHS(W)
n ← rows[W]
L^(1) ← W
for m ← 2 to n − 1
    do L^(m) ← EXTEND-SHORTEST-PATHS(L^(m−1), W)
return L^(n−1)
```

- Time complexity $\Theta(n^4)$.

Faster method

- How to make the slow method faster?

Faster method

- How to make the slow method faster?
- If there is no negative-weight cycle, then $L^{(m)} = L^{(n-1)}$ for all $m \geq n-1$.

Faster method

- How to make the slow method faster?
- If there is no negative-weight cycle, then $L^{(m)} = L^{(n-1)}$ for all $m \geq n-1$.
- Matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.

Faster method

- How to make the slow method faster?
- If there is no negative-weight cycle, then $L^{(m)} = L^{(n-1)}$ for all $m \geq n-1$.
- Matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.
- Therefore, instead of $n-1$ multiplications, only $\lceil \log n - 1 \rceil$ suffice.

Faster method

- How to make the slow method faster?
- If there is no negative-weight cycle, then $L^{(m)} = L^{(n-1)}$ for all $m \geq n-1$.
- Matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.
- Therefore, instead of $n-1$ multiplications, only $\lceil \log n - 1 \rceil$ suffice.
- We compute the following sequence of matrices

$$\begin{array}{cccc} L^{(1)} & = & W & \\ L^{(2)} & = & W^2 & \\ L^{(4)} & = & W^4 & = & W^2 \cdot W^2 \\ L^{(8)} & = & W^8 & = & W^4 \cdot W^4 \\ & & \vdots & & \\ L^{(2^{\lceil \log n-1 \rceil})} & = & W^{(2^{\lceil \log n-1 \rceil})} & = & W^{2^{\lceil \log n-1 \rceil - 1}} \cdot W^{2^{\lceil \log n-1 \rceil - 1}} \end{array}$$

Since $2^{\lceil \log n-1 \rceil} \geq n-1$, we have $L^{(2^{\lceil \log n-1 \rceil})} = L^{(n-1)}$.

Faster method

```
FAST-ALL-SHORTEST-PATHS(W)
n ← rows[W]
L^(1) ← W
m ← 1
while m < n − 1
        do L^(2m) ← EXTEND-SHORTEST-PATHS(L^(m), L^(m))
           m ← 2m
return L^(m)
```

- Time complexity $\Theta(n^3 \log n)$.

The Floyd-Warshall algorithm

The Floyd-Warshall algorithm

- Negative-weight edges are allowed,
- but we assume, there are no negative-weight cycle.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.
- Let $\{1, 2, \ldots, k\} \subseteq V = \{1, 2, \ldots, n\}$.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.
- Let $\{1, 2, \ldots, k\} \subseteq V = \{1, 2, \ldots, n\}$.
- For $i, j \in V$, consider all paths from i to j, where the inner vertices are from set $\{1, 2, \ldots, k\}$.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.
- Let $\{1, 2, \ldots, k\} \subseteq V = \{1, 2, \ldots, n\}$.
- For $i, j \in V$, consider all paths from i to j, where the inner vertices are from set $\{1, 2, \ldots, k\}$.
- Let p be such shortest path.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.
- Let $\{1, 2, \ldots, k\} \subseteq V = \{1, 2, \ldots, n\}$.
- For $i, j \in V$, consider all paths from i to j, where the inner vertices are from set $\{1, 2, \ldots, k\}$.
- Let p be such shortest path.
- Floyd-Warshall algorithm uses the relation between p and a shortest path from i to j that has inner vertices from set $\{1, 2, \ldots, k-1\}$.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.
- Let $\{1, 2, \ldots, k\} \subseteq V = \{1, 2, \ldots, n\}$.
- For $i, j \in V$, consider all paths from i to j, where the inner vertices are from set $\{1, 2, \ldots, k\}$.
- Let p be such shortest path.
- Floyd-Warshall algorithm uses the relation between p and a shortest path from i to j that has inner vertices from set $\{1, 2, \ldots, k-1\}$.
  - If k is not an inner vertex of p, then all inner vertices of p are from $\{1, 2, \ldots, k-1\}$. So, a shortest path from i to j with inner vertices from $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with inner vertices from $\{1, 2, \ldots, k\}$.

Structure of shortest paths

- Inner vertex of shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is a vertex v_i for $1 < i < k$.
- Let $\{1, 2, \ldots, k\} \subseteq V = \{1, 2, \ldots, n\}$.
- For $i, j \in V$, consider all paths from i to j, where the inner vertices are from set $\{1, 2, \ldots, k\}$.
- Let p be such shortest path.
- Floyd-Warshall algorithm uses the relation between p and a shortest path from i to j that has inner vertices from set $\{1, 2, \ldots, k-1\}$.
  - If k is not an inner vertex of p, then all inner vertices of p are from $\{1, 2, \ldots, k-1\}$. So, a shortest path from i to j with inner vertices from $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with inner vertices from $\{1, 2, \ldots, k\}$.
  - If k is an inner vertex of p, then $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ such that p_1 is a shortest path from i to k with inner vertices from $\{1, 2, \ldots, k-1\}$ and p_2 is a shortest path from k to j with inner vertices from $\{1, 2, \ldots, k-1\}$.

Recursion

- Let $d_{ij}^{(k)}$ is a weight of a shortest path from i to j that has all inner vertices from set $\{1, 2, \ldots, k\}$.

Recursion

- Let $d_{ij}^{(k)}$ is a weight of a shortest path from i to j that has all inner vertices from set $\{1, 2, \ldots, k\}$.
- $k = 0$ if and only if $d_{ij}^{(0)} = w_{ij}$. Therefore,

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{for } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{for } k \geq 1 \end{cases}$$

Recursion

- Let $d_{ij}^{(k)}$ is a weight of a shortest path from i to j that has all inner vertices from set $\{1, 2, \ldots, k\}$.
- $k = 0$ if and only if $d_{ij}^{(0)} = w_{ij}$. Therefore,

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{for } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{for } k \geq 1 \end{cases}$$

- Since for $k = n$ all inner vertices are from $V = \{1, 2, \ldots, n\}$, the matrix $D^{(n)} = (d_{ij}^{(n)})$ contains $d_{ij}^{(n)} = \delta(i, j)$ for $i, j \in V$.

Computation

FLOYD-WARSHALL(W)

$n \leftarrow rows[W]$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
    do for $i \leftarrow 1$ **to** n
        do for $j \leftarrow 1$ **to** n
            do $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
return $D^{(n)}$

- Time complexity $\Theta(n^3)$.

Construction of shortest paths

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{for } i = j \text{ or } w_{ij} = \infty \\ i & \text{for } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

Construction of shortest paths

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{for } i = j \text{ or } w_{ij} = \infty \\ i & \text{for } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

For $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{for } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{for } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Transitive closure of graph

- Given digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.

Transitive closure of graph

- Given digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
- Transitive closure of graph G is graph $G^* = (V, E^*)$, where

$$E^* = \{(i,j) : \text{there is a path from } i \text{ to } j \text{ in } G\}\,.$$

Transitive closure of graph

- Given digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
- Transitive closure of graph G is graph $G^* = (V, E^*)$, where

$$E^* = \{(i,j) : \text{there is a path from } i \text{ to } j \text{ in } G\}.$$

- To each edge assign value 1 and run Floyd-Warshall (in $\Theta(n^3)$ time).

Transitive closure of graph

- Given digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
- Transitive closure of graph G is graph $G^* = (V, E^*)$, where

$$E^* = \{(i,j) : \text{there is a path from } i \text{ to } j \text{ in } G\} .$$

- To each edge assign value 1 and run Floyd-Warshall (in $\Theta(n^3)$ time).
  - If there is a path from i to j, then $d_{ij} < n$.

Transitive closure of graph

- Given digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
- Transitive closure of graph G is graph $G^* = (V, E^*)$, where

$$E^* = \{(i,j) : \text{there is a path from } i \text{ to } j \text{ in } G\} .$$

- To each edge assign value 1 and run FLOYD-WARSHALL (in $\Theta(n^3)$ time).
  - If there is a path from i to j, then $d_{ij} < n$.
  - Otherwise, $d_{ij} = \infty$.

Transitive closure of graph

- Given digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
- Transitive closure of graph G is graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}.$$

- To each edge assign value 1 and run FLOYD-WARSHALL (in $\Theta(n^3)$ time).
  - If there is a path from i to j, then $d_{ij} < n$.
  - Otherwise, $d_{ij} = \infty$.
- We can improve a little bit . . . .

Transitive closure of graph II

- We use logical operators $\vee$, $\wedge$ instead of $\min$, $+$, respectively.

Transitive closure of graph II

- We use logical operators $\vee$, $\wedge$ instead of $\min$, $+$, respectively.
- Define $t_{ij}^{(k)}$, $i,j,k \in \{1,2,\ldots,n\}$ such that $t_{ij}^{(k)} = 1$ if there is a path from i to j with inner vertices from $\{1,2,\ldots,k\}$; otherwise, 0.

Transitive closure of graph II

- We use logical operators $\vee$, $\wedge$ instead of $\min$, $+$, respectively.
- Define $t_{ij}^{(k)}$, $i, j, k \in \{1, 2, \ldots, n\}$ such that $t_{ij}^{(k)} = 1$ if there is a path from i to j with inner vertices from $\{1, 2, \ldots, k\}$; otherwise, 0.
- So

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{for } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{for } i = j \text{ or } (i,j) \in E \end{cases}$$

Transitive closure of graph II

- We use logical operators $\vee$, $\wedge$ instead of $\min$, $+$, respectively.
- Define $t_{ij}^{(k)}$, $i,j,k \in \{1,2,\ldots,n\}$ such that $t_{ij}^{(k)} = 1$ if there is a path from i to j with inner vertices from $\{1,2,\ldots,k\}$; otherwise, 0.
- So

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{for } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{for } i = j \text{ or } (i,j) \in E \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) .$$

Transitive closure of graph II

- We use logical operators $\vee$, $\wedge$ instead of $\min$, $+$, respectively.
- Define $t_{ij}^{(k)}$, $i, j, k \in \{1, 2, \ldots, n\}$ such that $t_{ij}^{(k)} = 1$ if there is a path from i to j with inner vertices from $\{1, 2, \ldots, k\}$; otherwise, 0.
- So

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{for } i \neq j \text{ and } (i, j) \notin E \\ 1 & \text{for } i = j \text{ or } (i, j) \in E \end{cases}$$

  and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) .$$

- Similarly to Floyd-Warshall algorithm, we have 3 **for**-cycles, so the time complexity is $\Theta(n^3)$. Is it really better?

Transitive closure of graph II

- We use logical operators $\vee$, $\wedge$ instead of $\min$, $+$, respectively.
- Define $t_{ij}^{(k)}$, $i, j, k \in \{1, 2, \ldots, n\}$ such that $t_{ij}^{(k)} = 1$ if there is a path from i to j with inner vertices from $\{1, 2, \ldots, k\}$; otherwise, 0.
- So

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{for } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{for } i = j \text{ or } (i,j) \in E \end{cases}$$

  and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) .$$

- Similarly to Floyd-Warshall algorithm, we have 3 **for**-cycles, so the time complexity is $\Theta(n^3)$. Is it really better?
- Logical operations with bits are usually faster than arithmetical operations with integers (not asymptotically). Moreover, lower space complexity (bits vs. bytes).

Flow Networks

Network

- A flow network (or simply, network) $G = (V, E)$ is a directed graph

Network

- A flow network (or simply, network) $G = (V, E)$ is a directed graph
- in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.

Network

- A flow network (or simply, network) $G = (V, E)$ is a directed graph
- in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.
- If $(u, v) \notin E$, then assume that $c(u, v) = 0$.

Network

- A flow network (or simply, network) $G = (V, E)$ is a directed graph
- in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.
- If $(u, v) \notin E$, then assume that $c(u, v) = 0$.
- Two distinguishable vertices: a source s and a sink t (or terminator/target).

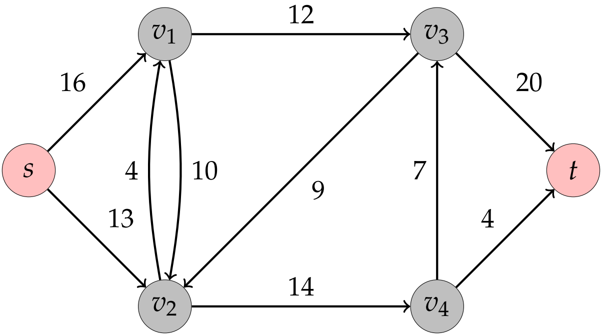
Network

- A flow network (or simply, network) $G = (V, E)$ is a directed graph
- in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.
- If $(u, v) \notin E$, then assume that $c(u, v) = 0$.
- Two distinguishable vertices: a source s and a sink t (or terminator/target).
- Every vertex lies on some path from s to t. That is, there is $s \rightsquigarrow v \rightsquigarrow t$ for every $v \in V$.

Network

- A flow network (or simply, network) $G = (V, E)$ is a directed graph
- in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.
- If $(u, v) \notin E$, then assume that $c(u, v) = 0$.
- Two distinguishable vertices: a source s and a sink t (or terminator/target).
- Every vertex lies on some path from s to t. That is, there is $s \leadsto v \leadsto t$ for every $v \in V$.
- Therefore, a flow network is connected graph with $m \geq n - 1$.

Flow network – Example



Flow

- A flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ satisfying 3 conditions:

Flow

- A flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ satisfying 3 conditions:

  1. **Capacity constraint:** For all $u, v \in V$, $f(u, v) \leq c(u, v)$.

Flow

- A flow in G is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ satisfying 3 conditions:

  1. **Capacity constraint:** For all $u, v \in V$, $f(u, v) \leq c(u, v)$.
  2. **Skew symmetry:** For all $u, v \in V$, $f(u, v) = -f(v, u)$.

Flow

- A flow in G is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ satisfying 3 conditions:

    1. **Capacity constraint:** For all $u, v \in V$, $f(u,v) \leq c(u,v)$.
    2. **Skew symmetry:** For all $u, v \in V$, $f(u,v) = -f(v,u)$.
    3. **Flow conservation:** For all $u \in V - \{s,t\}$, $\sum_{v \in V} f(u,v) = 0$.

Flow

- A flow in G is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ satisfying 3 conditions:

    1. **Capacity constraint:** For all $u, v \in V$, $f(u,v) \leq c(u,v)$.
    2. **Skew symmetry:** For all $u, v \in V$, $f(u,v) = -f(v,u)$.
    3. **Flow conservation:** For all $u \in V - \{s,t\}$, $\sum_{v \in V} f(u,v) = 0$.

- The quantity $f(u,v)$ is called the flow from vertex u to vertex v.

Flow

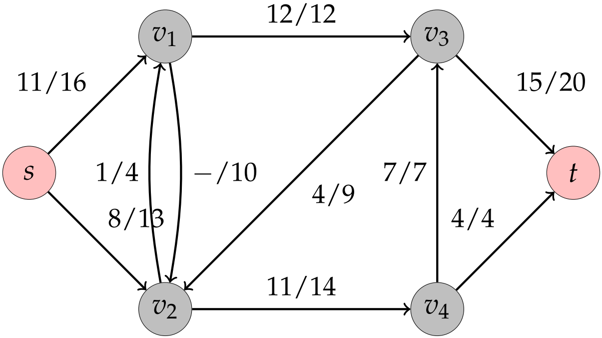
- A flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ satisfying 3 conditions:

  1. **Capacity constraint:** For all $u, v \in V$, $f(u,v) \le c(u,v)$.
  2. **Skew symmetry:** For all $u, v \in V$, $f(u,v) = -f(v,u)$.
  3. **Flow conservation:** For all $u \in V - \{s,t\}$, $\sum_{v \in V} f(u,v) = 0$.

- The quantity $f(u,v)$ is called the flow from vertex u to vertex v.
- The value of a flow f is defined as

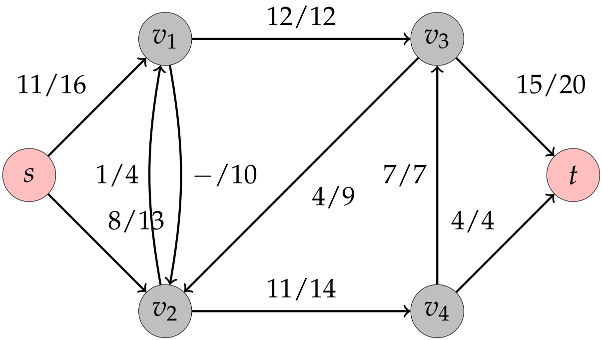
$$|f| = \sum_{v \in V} f(s,v) .$$

Flow network – Example



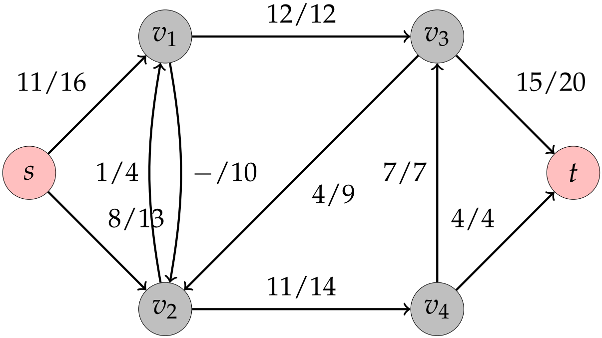
- Edges labeled with $f(u,v)/c(u,v)$. Only positive flows are shown.

Flow network – Example



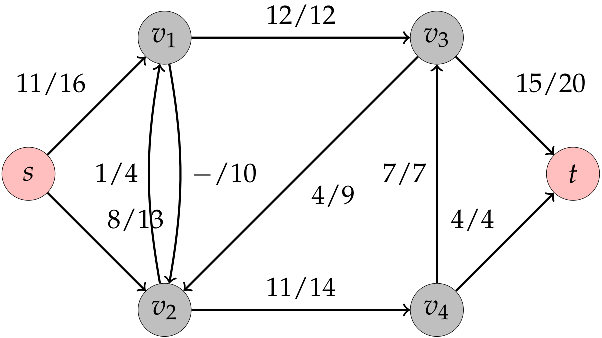
- Edges labeled with $f(u, v)/c(u, v)$. Only positive flows are shown.
- Verify that it is a flow network and some flow.

Flow network – Example



- Edges labeled with $f(u,v)/c(u,v)$. Only positive flows are shown.
- Verify that it is a flow network and some flow.
- $|f| = ???$

Flow network – Example

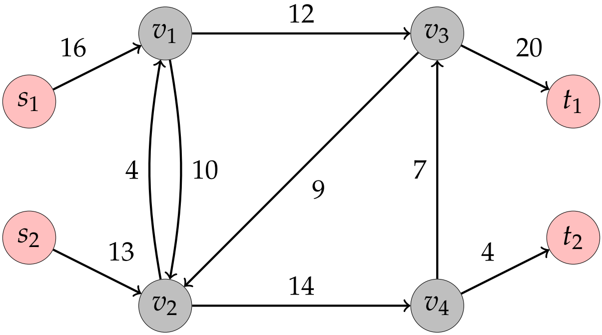


- Edges labeled with $f(u, v)/c(u, v)$. Only positive flows are shown.
- Verify that it is a flow network and some flow.
- $|f| =???$
- $|f| = 19$.

Maximum-flow Problem

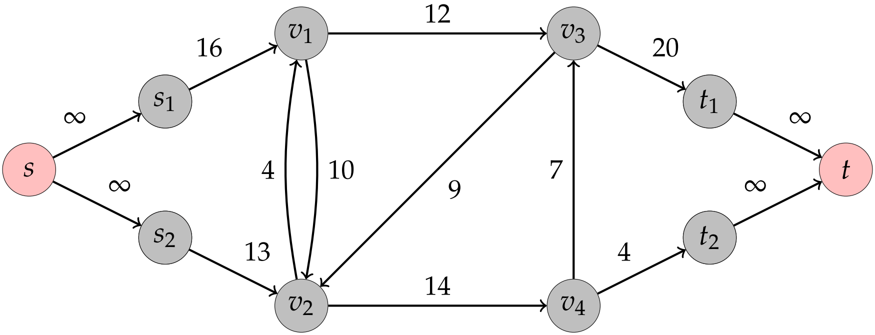
- We are given a flow network G with source s and sink t,
- we wish to find a flow of maximum value.

Networks with multiple sources and sinks



- How to deal with it?

Networks with multiple sources and sinks



- How to deal with it?
- Create a new supersource s and a new supersink and set the capacity to ∞ for these new edges.

Working with flows

- For $X, Y \subseteq V$, we define $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$.

Working with flows

- For $X, Y \subseteq V$, we define $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$.
- Then, the value of f is $|f| = f(s,V)$.

Working with flows

- For $X, Y \subseteq V$, we define $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$.
- Then, the value of f is $|f| = f(s,V)$.
- For all $X \subseteq V$, $f(X,X) = 0$ — with every $f(u,v)$ we sum in $f(v,u)$ as well.

Working with flows

- For $X, Y \subseteq V$, we define $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$.
- Then, the value of f is $|f| = f(s, V)$.
- For all $X \subseteq V$, $f(X, X) = 0$ — with every $f(u, v)$ we sum in $f(v, u)$ as well.
- For all $X, Y \subseteq V$, $f(X, Y) = -f(Y, X)$.

Working with flows

- For $X, Y \subseteq V$, we define $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$.
- Then, the value of f is $|f| = f(s,V)$.
- For all $X \subseteq V$, $f(X,X) = 0$ — with every $f(u,v)$ we sum in $f(v,u)$ as well.
- For all $X, Y \subseteq V$, $f(X,Y) = -f(Y,X)$.
- For all $X, Y, Z \subseteq V$, $X \cap Y = \emptyset$,

  $$f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$$

  and

  $$f(Z, X \cup Y) = f(Z,X) + f(Z,Y).$$

Working with flows – Example

Prove that $|f| = f(V, t)$.

Working with flows – Example

Prove that $|f| = f(V, t)$.

Proof.

- $|f| = f(s, V)$

Working with flows – Example

Prove that $|f| = f(V, t)$.

Proof.

- $|f| = f(s, V)$
- We know that $f(V, V) = f(s, V) + f(V - s, V)$ – see above.

Working with flows – Example

Prove that $|f| = f(V, t)$.

Proof.

- $|f| = f(s, V)$
- We know that $f(V, V) = f(s, V) + f(V - s, V)$ – see above.
- Therefore, $f(s, V) = f(V, V) - f(V - s, V)$.

Working with flows – Example

Prove that $|f| = f(V,t)$.

Proof.

- $|f| = f(s,V)$
- We know that $f(V,V) = f(s,V) + f(V-s,V)$ – see above.
- Therefore, $f(s,V) = f(V,V) - f(V-s,V)$.
- We know that $f(V,V) = 0$ – see above.

Working with flows – Example

Prove that $|f| = f(V,t)$.

Proof.

- $|f| = f(s,V)$
- We know that $f(V,V) = f(s,V) + f(V-s,V)$ – see above.
- Therefore, $f(s,V) = f(V,V) - f(V-s,V)$.
- We know that $f(V,V) = 0$ – see above.
- Therefore, $f(s,V) = -f(V-s,V) = f(V,V-s)$.

Working with flows – Example

Prove that $|f| = f(V,t)$.

Proof.

- $|f| = f(s,V)$
- We know that $f(V,V) = f(s,V) + f(V-s,V)$ – see above.
- Therefore, $f(s,V) = f(V,V) - f(V-s,V)$.
- We know that $f(V,V) = 0$ – see above.
- Therefore, $f(s,V) = -f(V-s,V) = f(V,V-s)$.
- We know that $f(V,V-s) = f(V,t) + f(V,V-s-t)$ – see above.

Working with flows – Example

Prove that $|f| = f(V, t)$.

Proof.

- $|f| = f(s, V)$
- We know that $f(V, V) = f(s, V) + f(V - s, V)$ – see above.
- Therefore, $f(s, V) = f(V, V) - f(V - s, V)$.
- We know that $f(V, V) = 0$ – see above.
- Therefore, $f(s, V) = -f(V - s, V) = f(V, V - s)$.
- We know that $f(V, V - s) = f(V, t) + f(V, V - s - t)$ – see above.
- From the previous and by flow conservation, $f(V, V - s - t) = -f(V - s - t, V) = -\sum_{u \in V - \{s,t\}} \sum_{v \in V} f(u, v) = -\sum_{u \in V - \{s,t\}} 0 = 0$.

Working with flows – Example

Prove that $|f| = f(V, t)$.

Proof.

- $|f| = f(s, V)$
- We know that $f(V, V) = f(s, V) + f(V - s, V)$ – see above.
- Therefore, $f(s, V) = f(V, V) - f(V - s, V)$.
- We know that $f(V, V) = 0$ – see above.
- Therefore, $f(s, V) = -f(V - s, V) = f(V, V - s)$.
- We know that $f(V, V - s) = f(V, t) + f(V, V - s - t)$ – see above.
- From the previous and by flow conservation, $f(V, V - s - t) = -f(V - s - t, V) = -\sum_{u \in V - \{s,t\}} \sum_{v \in V} f(u, v) = -\sum_{u \in V - \{s,t\}} 0 = 0.$
- Thus, $|f| = f(V, t)$.

□

The Ford-Fulkerson Method

The Ford-Fulkerson Method

- To find the maximum flow in the given network.

The Ford-Fulkerson Method

- To find the maximum flow in the given network.
- Not algorithm - there are several implementations with different complexity.

The Ford-Fulkerson Method

- To find the maximum flow in the given network.
- Not algorithm - there are several implementations with different complexity.

FORD-FULKERSON-METHOD(G, s, t)

inicialize $f(u,v) = 0$ for each $u, v \in V$
while there exists an augmenting path p
    do augment flow f along p
return f

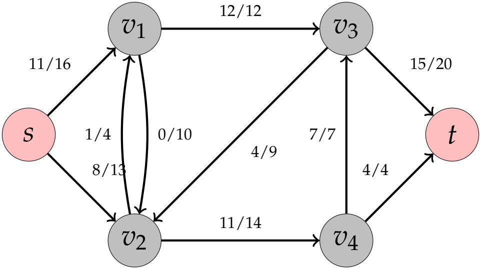
The Ford-Fulkerson Method

- To find the maximum flow in the given network.
- Not algorithm - there are several implementations with different complexity.

```
FORD-FULKERSON-METHOD(G, s, t)
  inicialize f(u, v) = 0 for each u, v ∈ V
  while there exists an augmenting path p
        do augment flow f along p
  return f
```

- Augmenting path is a simple path from s to t along which the flow can be increased.

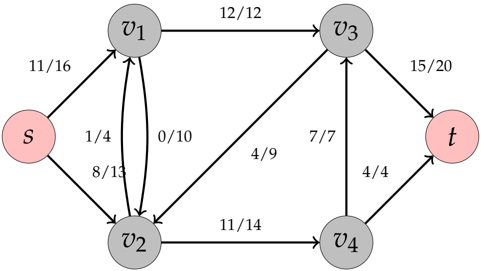
Residual Network(s)



- Residual capacity of (u, v) is

$$c_f(u,v) = c(u,v) - f(u,v)\,.$$

Residual Network(s)

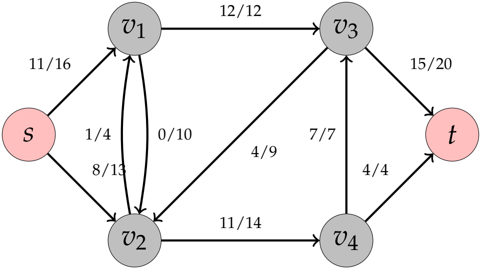


- Residual capacity of (u, v) is

$$c_f(u, v) = c(u, v) - f(u, v) .$$

- For example, $c_f(s, v_1) = 16 - 11 = 5$.

Residual Network(s)



- Residual capacity of (u, v) is

$$c_f(u, v) = c(u, v) - f(u, v)\,.$$

- For example, $c_f(s, v_1) = 16 - 11 = 5$.
- Flow $f(u, v)$ can be increased by 5 units.

Residual Network

- Let $G = (V, E)$ be a network and f be a flow in G.

Residual Network

- Let $G = (V, E)$ be a network and f be a flow in G.
- The residual network of G inducted by flow f is a network $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

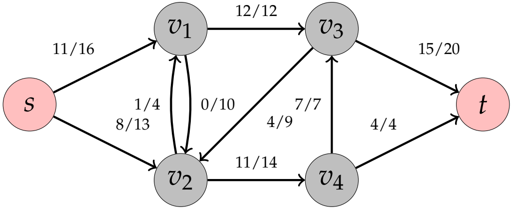
Residual Network

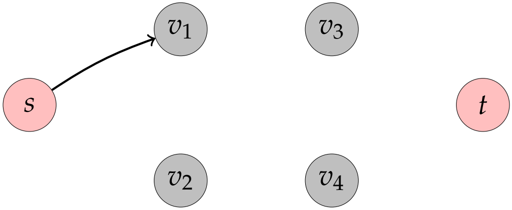
- Let $G = (V, E)$ be a network and f be a flow in G.
- The residual network of G inducted by flow f is a network $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} .$$

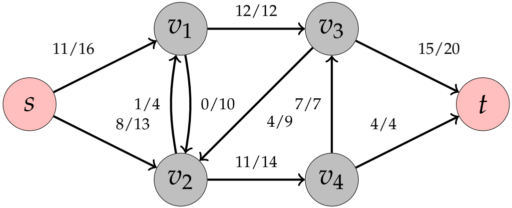
- It holds that $|E_f| \leq 2|E|$ – Think about it!

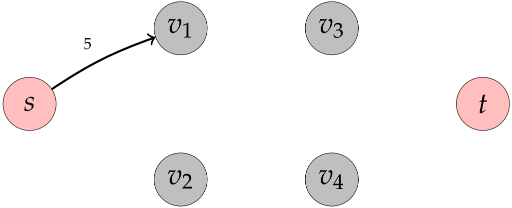
Network and its residual network



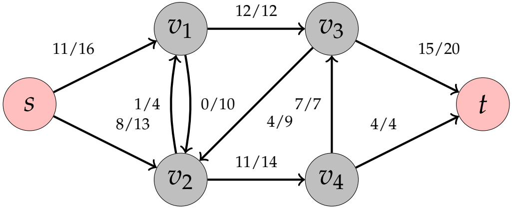


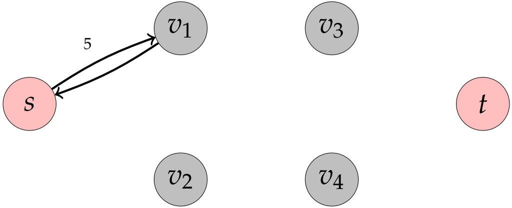
Network and its residual network



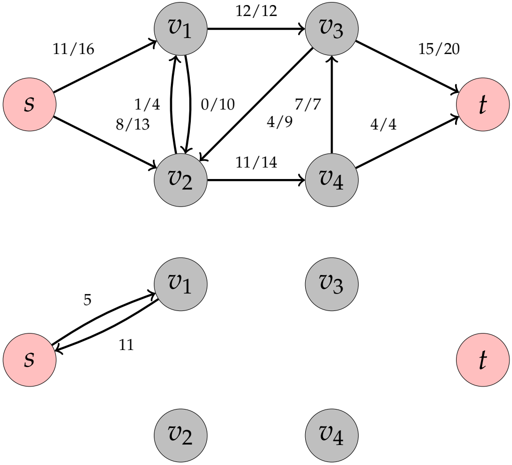


Network and its residual network

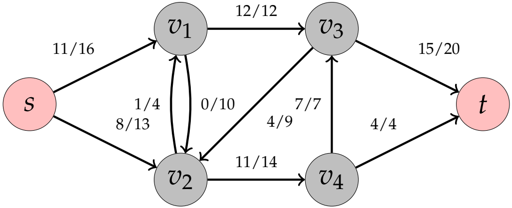


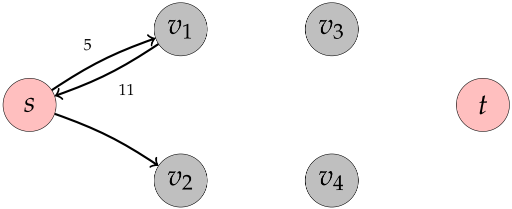


Network and its residual network

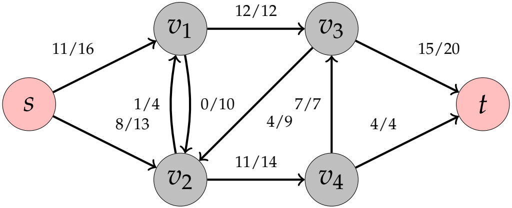


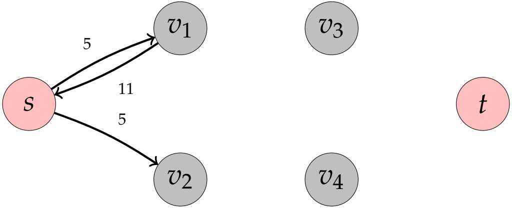
Network and its residual network



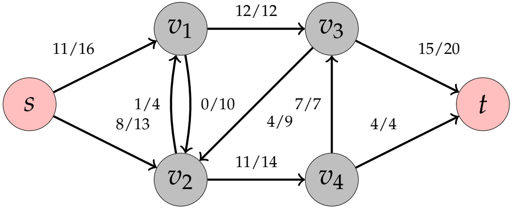


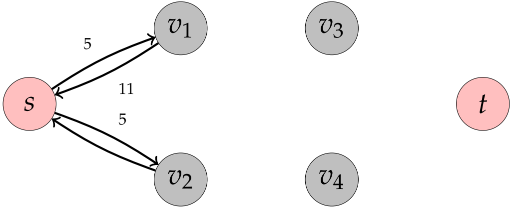
Network and its residual network



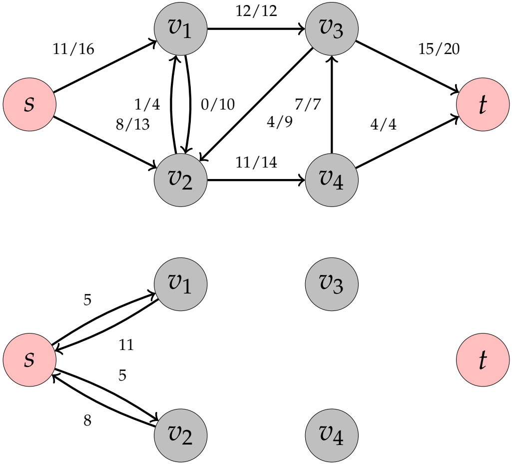


Network and its residual network

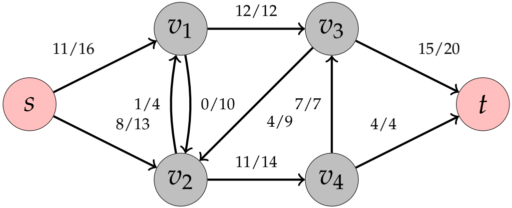


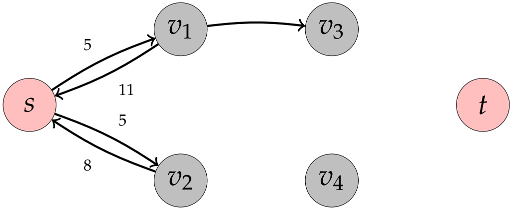


Network and its residual network

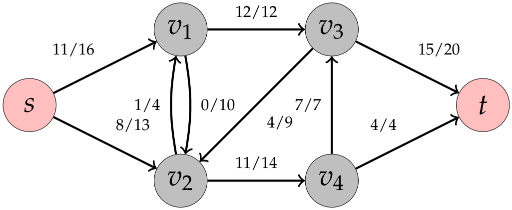


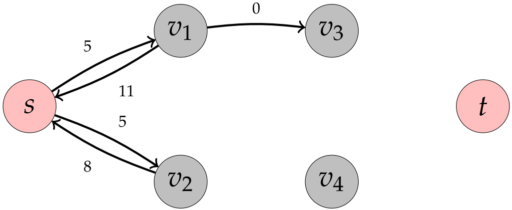
Network and its residual network



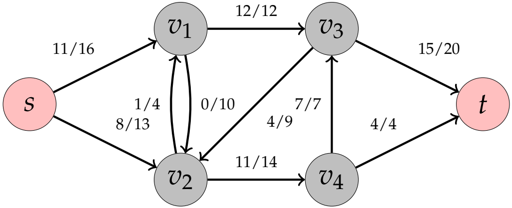


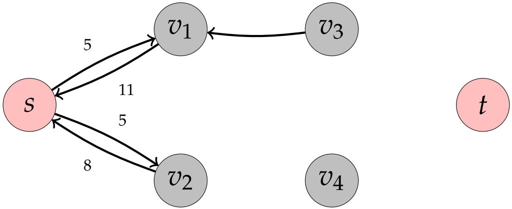
Network and its residual network



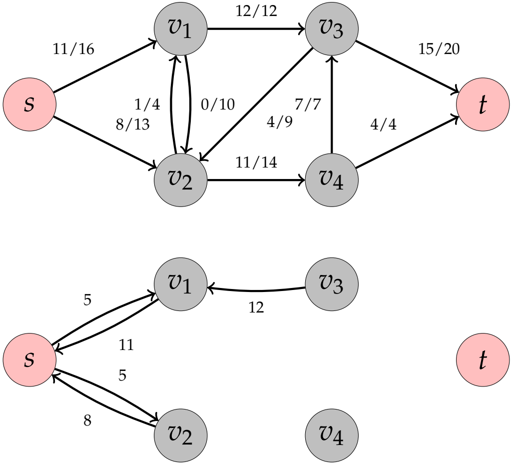


Network and its residual network

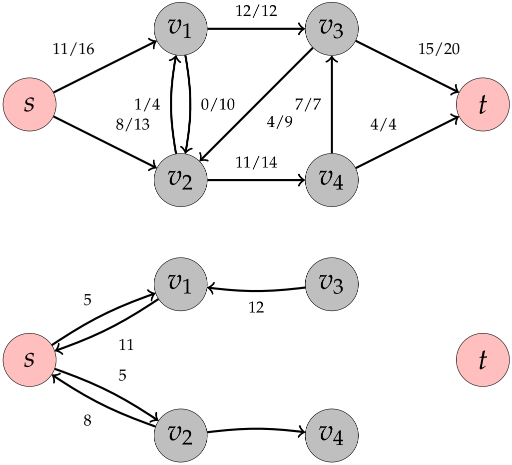




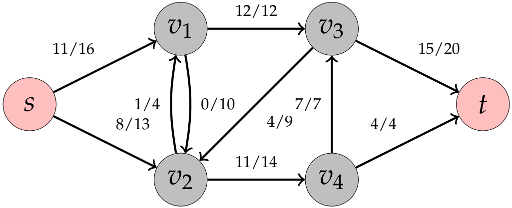
Network and its residual network

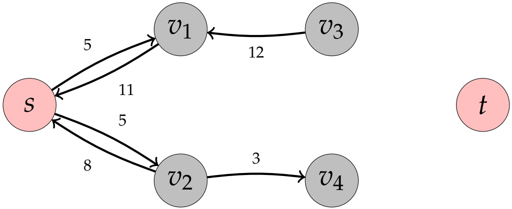


Network and its residual network

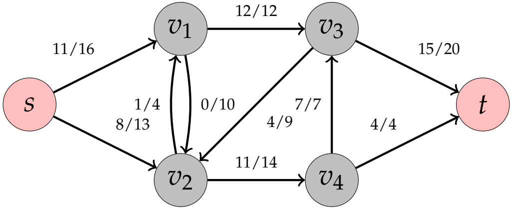


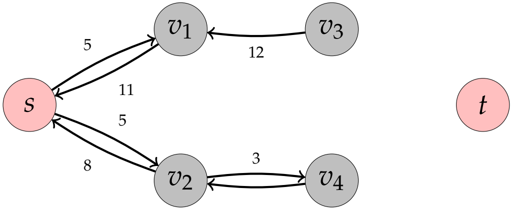
Network and its residual network



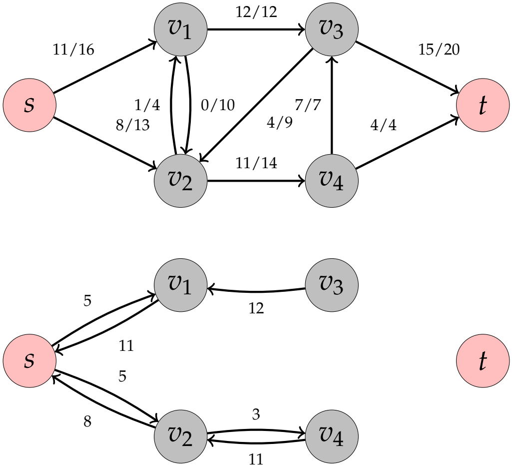


Network and its residual network

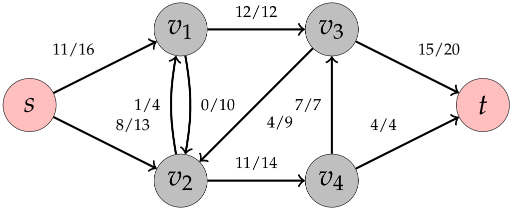


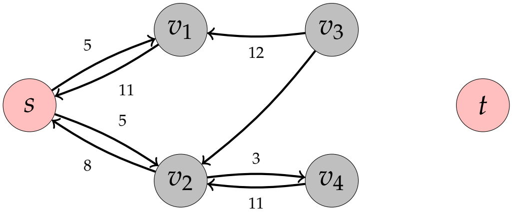


Network and its residual network

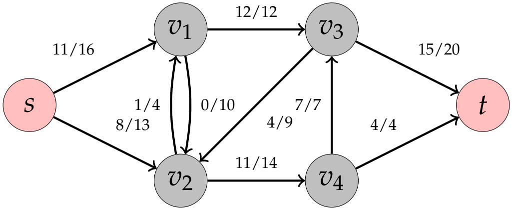


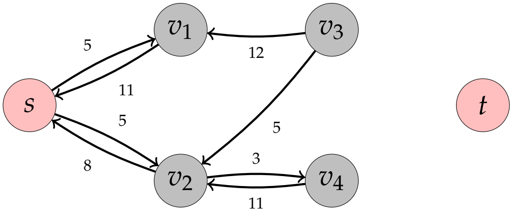
Network and its residual network



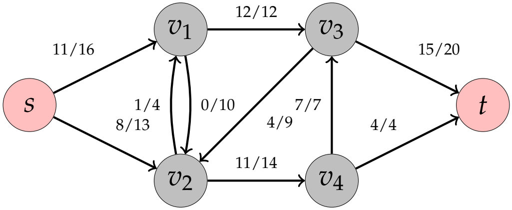


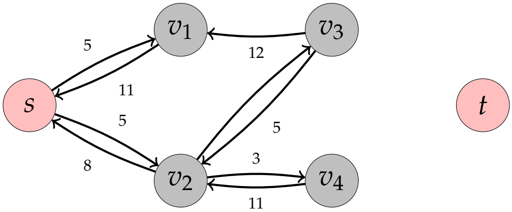
Network and its residual network



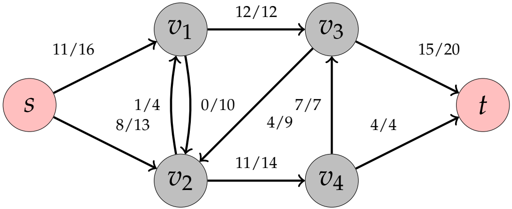


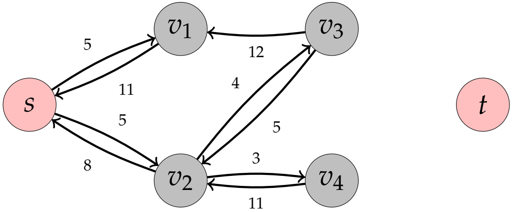
Network and its residual network



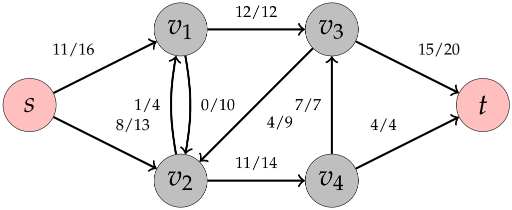


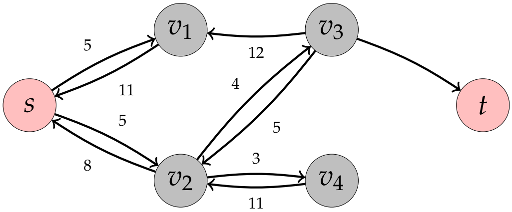
Network and its residual network



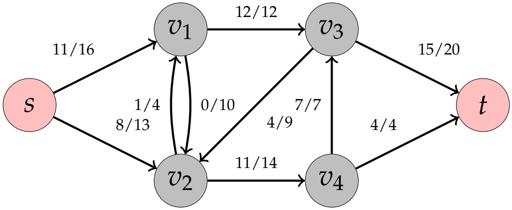


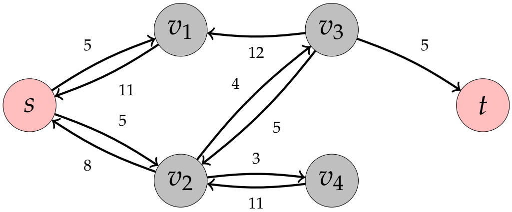
Network and its residual network



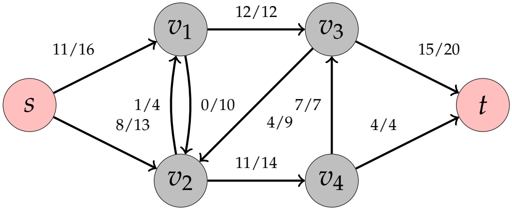


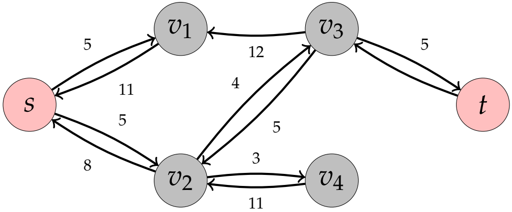
Network and its residual network



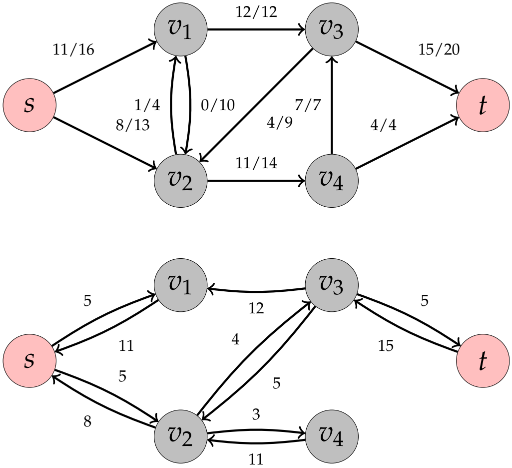


Network and its residual network

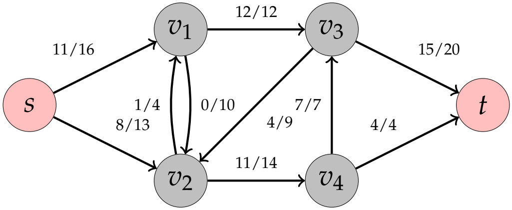


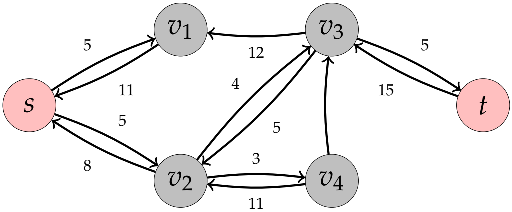


Network and its residual network

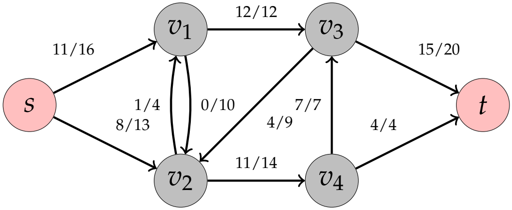


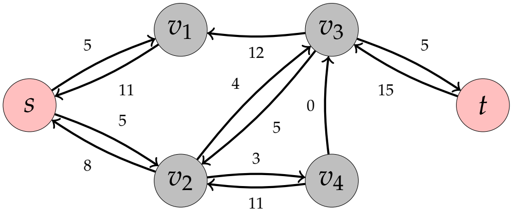
Network and its residual network



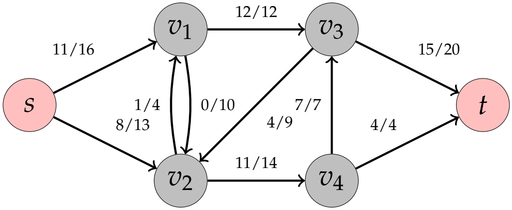


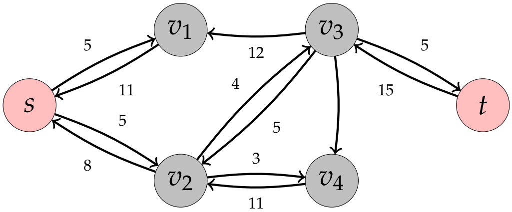
Network and its residual network



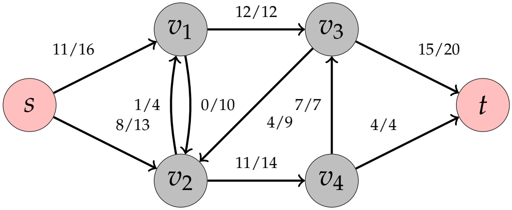


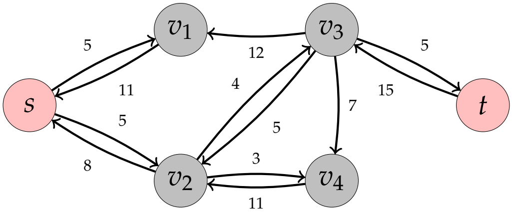
Network and its residual network



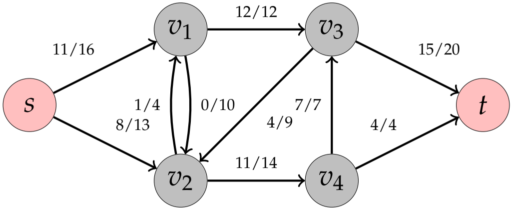


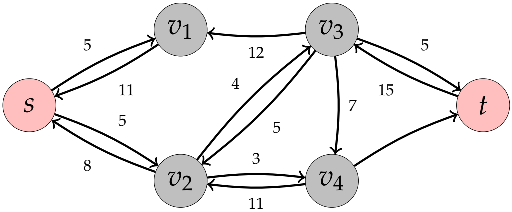
Network and its residual network



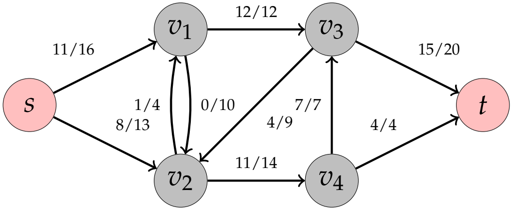


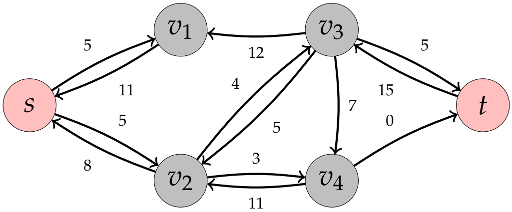
Network and its residual network



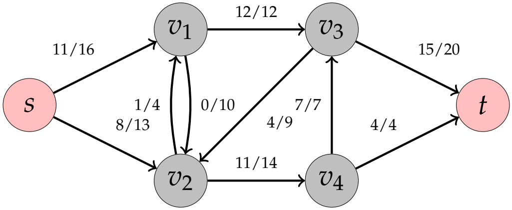


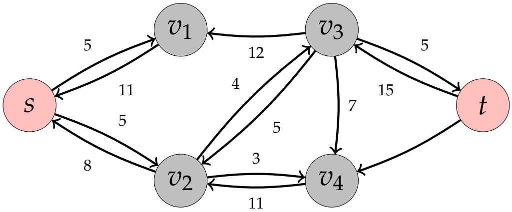
Network and its residual network



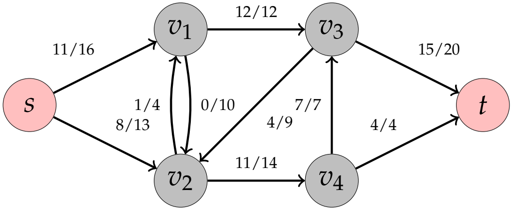


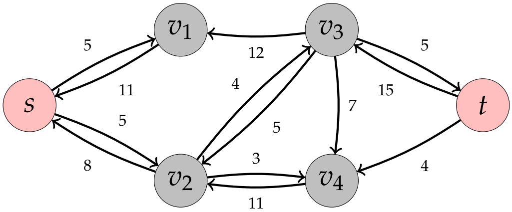
Network and its residual network



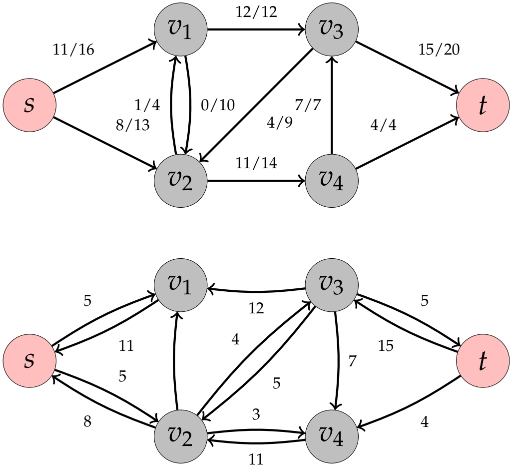


Network and its residual network

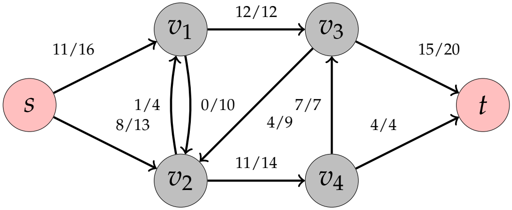


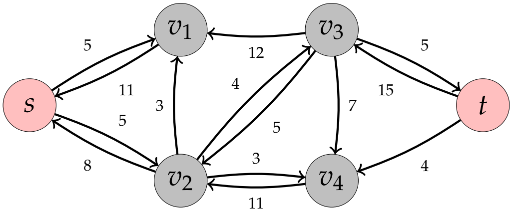


Network and its residual network

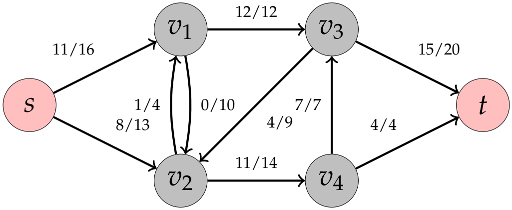


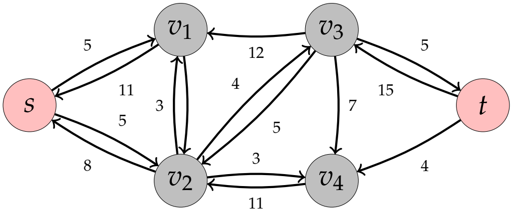
Network and its residual network



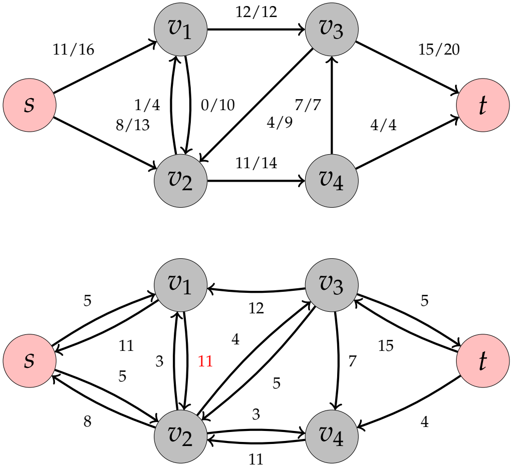


Network and its residual network

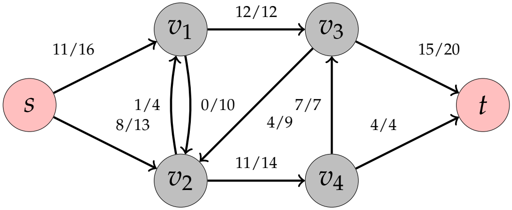


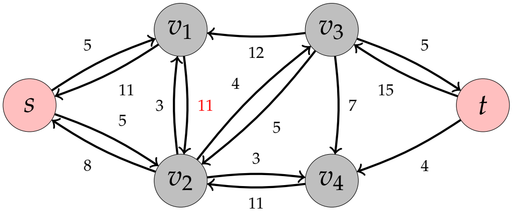


Network and its residual network



Network and its residual network





- Attention! $f(v_1, v_2) = 0 + (-1)$ so $c_f(v_1, v_2) = 10 - (-1) = 11$.

Residual network

Lemma 23.
Let $G = (V, E)$ *be a network and* f *be a flow in* G. *Let* G_f *be a residual network of* G *induced by* f *and let* f' *be a flow in* G_f.
Then, $f + f'$ *is a flow in* G *with the value of* $|f + f'| = |f| + |f'|$.

Proof.

- We must verify that tree conditions from the definition of a flow.

□

Condition 1: Capacity constraint

Demonstrate that $(f + f')(u,v) \leq c(u,v)$.

Proof.

- $f'(u,v) \leq c_f(u,v)$.

□

Condition 1: Capacity constraint

Demonstrate that $(f+f')(u,v) \leq c(u,v)$.

Proof.

- $f'(u,v) \leq c_f(u,v)$.
- $(f+f')(u,v) = f(u,v) + f'(u,v)$

□

Condition 1: Capacity constraint

Demonstrate that $(f+f')(u,v) \le c(u,v)$.

Proof.

- $f'(u,v) \le c_f(u,v)$.
- $$\begin{aligned}(f+f')(u,v) &= f(u,v) + f'(u,v) \\ &\le f(u,v) + (c(u,v) - f(u,v))\end{aligned}$$

□

Condition 1: Capacity constraint

Demonstrate that $(f + f')(u,v) \leq c(u,v)$.

Proof.

- $f'(u,v) \leq c_f(u,v)$.
- $$\begin{aligned}(f + f')(u,v) &= f(u,v) + f'(u,v) \\ &\leq f(u,v) + (c(u,v) - f(u,v)) \\ &= c(u,v).\end{aligned}$$

□

Condition 2: Skew symmetry

Demonstrate that $(f + f')(u, v) = -(f + f')(v, u)$.

Proof.

- $(f + f')(u, v) = f(u, v) + f'(u, v)$

□

Condition 2: Skew symmetry

Demonstrate that $(f + f')(u, v) = -(f + f')(v, u)$.

Proof.

- $$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u)\end{aligned}$$

□

Condition 2: Skew symmetry

Demonstrate that $(f+f')(u,v) = -(f+f')(v,u)$.

Proof.

- $$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &= -f(v,u)-f'(v,u)\\ &= -(f(v,u)+f'(v,u))\end{aligned}$$

□

Condition 2: Skew symmetry

Demonstrate that $(f+f')(u,v) = -(f+f')(v,u)$.

Proof.

- $$\begin{aligned}(f+f')(u,v) &= f(u,v) + f'(u,v) \\ &= -f(v,u) - f'(v,u) \\ &= -(f(v,u) + f'(v,u)) \\ &= -(f+f')(v,u).\end{aligned}$$

□

Condition 3: Flow conservation

Demonstrate that for $u \in V - \{s,t\}$, $\sum_{v \in V} (f + f')(u,v) = 0$.

Proof.

- $\sum_{v \in V} (f + f')(u,v) = \sum_{v \in V} (f(u,v) + f'(u,v))$

□

Condition 3: Flow conservation

Demonstrate that for $u \in V - \{s, t\}$, $\sum_{v \in V} (f + f')(u, v) = 0$.

Proof.

- $$\begin{aligned} \sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \end{aligned}$$

□

Condition 3: Flow conservation

Demonstrate that for $u \in V - \{s,t\}$, $\sum_{v \in V} (f + f')(u,v) = 0$.

Proof.

- $$\begin{aligned} \sum_{v \in V} (f + f')(u,v) &= \sum_{v \in V} (f(u,v) + f'(u,v)) \\ &= \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) \\ &= 0 + 0 = 0. \end{aligned}$$

□

Value of the resulting flow

- $|f + f'| = \sum_{v \in V} (f + f')(s, v)$

Value of the resulting flow

- $$|f + f'| = \sum_{v \in V} (f + f')(s, v) = \sum_{v \in V} (f(s, v) + f'(s, v))$$

Value of the resulting flow

- $$\begin{aligned} |f + f'| &= \sum_{v \in V} (f + f')(s, v) \\ &= \sum_{v \in V} (f(s, v) + f'(s, v)) \\ &= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \end{aligned}$$

Value of the resulting flow

- $$\begin{aligned} |f+f'| &= \sum_{v \in V} (f+f')(s,v) \\ &= \sum_{v \in V} (f(s,v) + f'(s,v)) \\ &= \sum_{v \in V} f(s,v) + \sum_{v \in V} f'(s,v) \\ &= |f| + |f'|. \end{aligned}$$

Augmenting path – Example

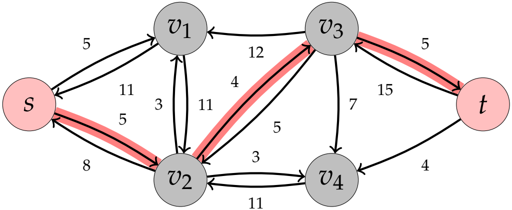
- Let $G = (V, E)$ be a network and f be a flow.

Augmenting path – Example

- Let $G = (V, E)$ be a network and f be a flow.
- Augmenting path p is a path from s to t along which flow f can be increased in G.

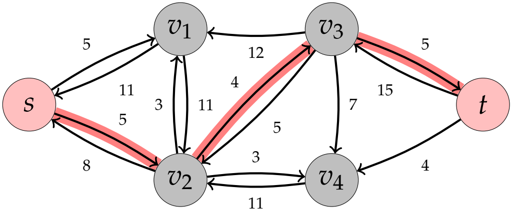
Augmenting path – Example

- Let $G = (V, E)$ be a network and f be a flow.
- Augmenting path p is a path from s to t along which flow f can be increased in G.



Augmenting path – Example

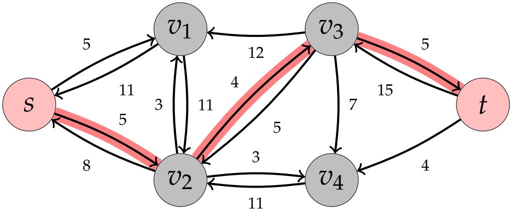
- Let $G = (V, E)$ be a network and f be a flow.
- Augmenting path p is a path from s to t along which flow f can be increased in G.



- Using this path, we can increase flow by 4 units.

Augmenting path – Example

- Let $G = (V, E)$ be a network and f be a flow.
- Augmenting path p is a path from s to t along which flow f can be increased in G.



- Using this path, we can increase flow by 4 units.
- Residual capacity of augmenting path p is

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ lies on path } p\}.$$

Lemma 24.
Let $G = (V, E)$ be a network, f be its flow and p be an augmenting path in G_f. Let define a function

$$f_p(u,v) = \begin{cases} c_f(p) & \text{for } (u,v) \text{ on } p \\ -c_f(p) & \text{for } (v,u) \text{ on } p \\ 0 & \text{otherwise} \end{cases}$$

Then, f_p is the flow in G_f of size $|f_p| = c_f(p) > 0$.

Proof.
Homework. □

Lemma 24.

Let $G = (V, E)$ be a network, f be its flow and p be an augmenting path in G_f. Let define a function

$$f_p(u, v) = \begin{cases} c_f(p) & \text{for } (u, v) \text{ on } p \\ -c_f(p) & \text{for } (v, u) \text{ on } p \\ 0 & \text{otherwise} \end{cases}$$

Then, f_p is the flow in G_f of size $|f_p| = c_f(p) > 0$.

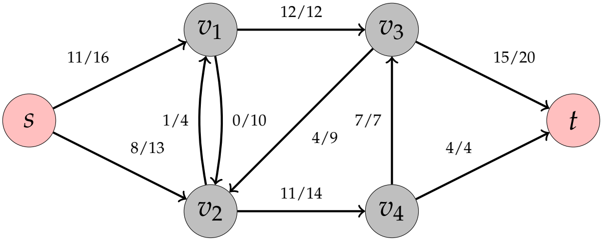
Proof.

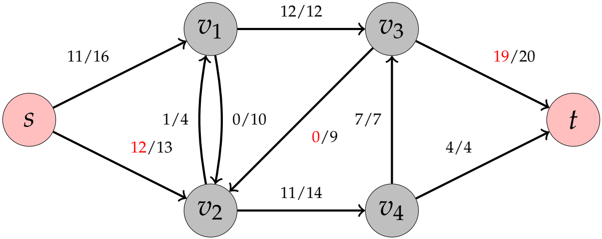
Homework. □

Corollary 25.

Let $f' = f + f_p$. Then, f' is a flow in G of size $|f'| = |f| + |f_p| > |f|$.

Residual network improved by 4 along $s \rightsquigarrow v_2 \rightsquigarrow v_3 \rightsquigarrow t$





Cut in Network

Cut in Flow Network

- Network cut in $G = (V, E)$ is a partition of V to S and $T = V - S$ such that $s \in S$ and $t \in T$.

Cut in Flow Network

- Network cut in $G = (V, E)$ is a partition of V to S and $T = V - S$ such that $s \in S$ and $t \in T$.
- Flow through a cut is defined as $f(S, T)$.

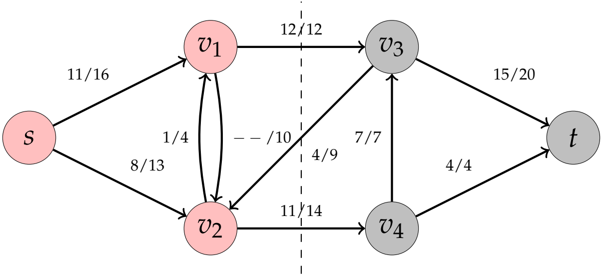
Cut in Flow Network

- Network cut in $G = (V, E)$ is a partition of V to S and $T = V - S$ such that $s \in S$ and $t \in T$.
- Flow through a cut is defined as $f(S, T)$.
- Cut capacity (S, T) is $c(S, T)$.

Cut in Flow Network

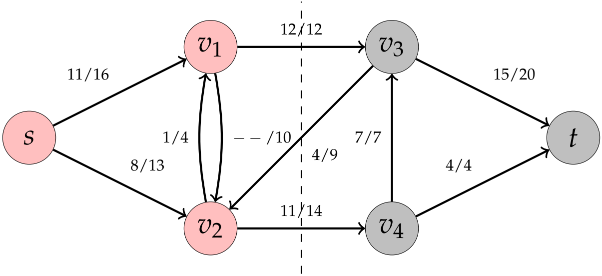
- Network cut in $G = (V, E)$ is a partition of V to S and $T = V - S$ such that $s \in S$ and $t \in T$.
- Flow through a cut is defined as $f(S, T)$.
- Cut capacity (S, T) is $c(S, T)$.
- Minimal cut is a cut with minimal capacity.

Cut in Network – Example



- Flow through a cut: $f(\{s, v_1, v_2\}, \{v_3, v_4, t\}) = f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) = 12 + (-4) + 11 = 19.$

Cut in Network – Example



- Flow through a cut: $f(\{s, v_1, v_2\}, \{v_3, v_4, t\}) = f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) = 12 + (-4) + 11 = 19.$
- Cut capacity: $c(\{s, v_1, v_2\}, \{v_3, v_4, t\}) = c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26.$

Properties

Lemma 26.
Let f be a flow in G with source s and sink t and let (S,T) be a cut of G. Then, $|f| = f(S,T)$.

Proof.

- $f(S,T) = f(S,V) - f(S,S)$

□

Properties

Lemma 26.
Let f be a flow in G with source s and sink t and let (S,T) be a cut of G. Then, $|f| = f(S,T)$.

Proof.

- $$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \end{aligned}$$

□

Properties

Lemma 26.
Let f be a flow in G with source s and sink t and let (S,T) be a cut of G. Then, $|f| = f(S,T)$.

Proof.

- $$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) + f(S - \{s\}, V) \end{aligned}$$

□

Properties

Lemma 26.
Let f be a flow in G with source s and sink t and let (S,T) be a cut of G. Then, $|f| = f(S,T)$.

Proof.

- $$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) + f(S-\{s\},V) \\ &= f(s,V) \end{aligned}$$

□

Properties

Lemma 26.
Let f be a flow in G with source s and sink t and let (S,T) be a cut of G. Then, $|f| = f(S,T)$.

Proof.

- $$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) + f(S - \{s\}, V) \\ &= f(s,V) \\ &= |f| \end{aligned}$$

□

Properties

Corollary 27.

The value of any flow f in G is bounded from above by the capacity of any cut of G.

Proof.

- $|f| = f(S,T)$

□

Properties

Corollary 27.

The value of any flow f in G is bounded from above by the capacity of any cut of G.

Proof.

- $$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \end{aligned}$$

□

Properties

Corollary 27.

The value of any flow f in G is bounded from above by the capacity of any cut of G.

Proof.

- $$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \end{aligned}$$

□

Properties

Corollary 27.

The value of any flow f in G is bounded from above by the capacity of any cut of G.

Proof.

- $$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) \end{aligned}$$

□

Properties

Corollary 27.

The value of any flow f in G is bounded from above by the capacity of any cut of G.

Proof.

- $$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) \end{aligned}$$

□

The value of a maximum flow is equal or less than the capacity of a minimum cut.

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(1) \Rightarrow (2)$:

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(1) \Rightarrow (2)$:
  - Let f is maximum flow and p is an augmenting path in G_f.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(1) \Rightarrow (2)$:
  - Let f is maximum flow and p is an augmenting path in G_f.
  - Then, $f + f_p$ is a flow in G and $|f + f_p| > |f|$. Contradition.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:
  - Let G_f contains no augmenting path, so no path from s to t in G_f.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:
  - Let G_f contains no augmenting path, so no path from s to t in G_f.
  - Let

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:
  - Let G_f contains no augmenting path, so no path from s to t in G_f.
  - Let

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

  - and let $T = V - S$.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:
  - Let G_f contains no augmenting path, so no path from s to t in G_f.
  - Let

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

  - and let $T = V - S$.
  - Since $s \in S$ and $t \in T$, (S,T) is a cut of G.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:
  - Let G_f contains no augmenting path, so no path from s to t in G_f.
  - Let

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

  - and let $T = V - S$.
  - Since $s \in S$ and $t \in T$, (S, T) is a cut of G.
  - For $u \in S$ and $v \in T$, we have $f(u, v) = c(u, v)$, otherwise $(u, v) \in E_f$, so $v \in S$.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(2) \Rightarrow (3)$:
  - Let G_f contains no augmenting path, so no path from s to t in G_f.
  - Let
$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$
  - and let $T = V - S$.
  - Since $s \in S$ and $t \in T$, (S,T) is a cut of G.
  - For $u \in S$ and $v \in T$, we have $f(u,v) = c(u,v)$, otherwise $(u,v) \in E_f$, so $v \in S$.
  - $|f| = f(S,T) = c(S,T)$.

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(3) \Rightarrow (1)$:

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(3) \Rightarrow (1)$:
  - $|f| \leq c(S,T)$ for any cut (S,T).

□

Max-flow min-cut Theorem

Let f be a flow in G with source s and sink t. Then, the following conditions are equivalent:

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

Proof.

- $(3) \Rightarrow (1)$:
  - $|f| \leq c(S, T)$ for any cut (S, T).
  - From $|f| = c(S, T)$, it follows that f is maximum.

□

The basic Ford-Fulkerson algorithm

The basic Ford-Fulkerson algorithm

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E
2       do f[u, v] ← 0
3          f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5          do c_f(p) ← min{c_f(u, v) : (u, v) is in p}
6             for each edge (u, v) in p
7                 do f[u, v] ← f[u, v] + c_f(p)
8                    f[v, u] ← −f[u, v]
```

- Time complexity depends on line 4.

The basic Ford-Fulkerson algorithm

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E
2       do f[u, v] ← 0
3          f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5         do c_f(p) ← min{c_f(u, v) : (u, v) is in p}
6            for each edge (u, v) in p
7                do f[u, v] ← f[u, v] + c_f(p)
8                   f[v, u] ← −f[u, v]
```

- Time complexity depends on line 4.
- Using BFS gives total complexity $O(nm^2)$ – so called Edmonds-Karp algorithm.

The basic Ford-Fulkerson algorithm – Example

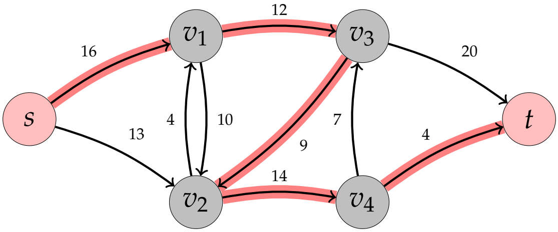


Figure: Residual network with an augmenting path from s to t.

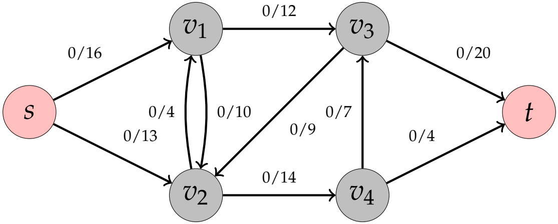


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

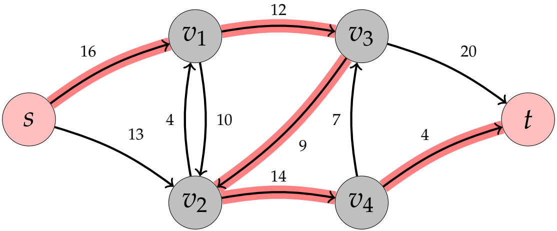


Figure: Residual network with an augmenting path from s to t.

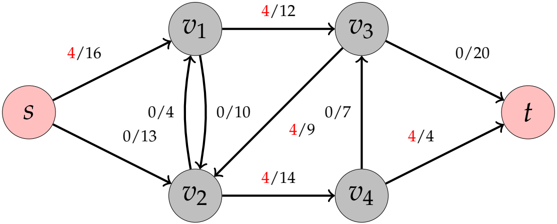


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

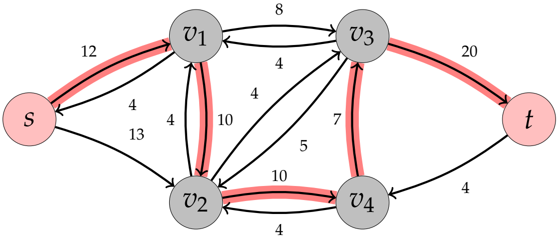


Figure: Residual network with an augmenting path from s to t.

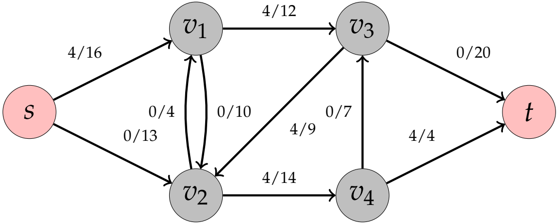


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

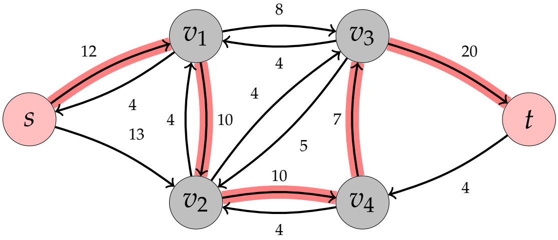


Figure: Residual network with an augmenting path from s to t.

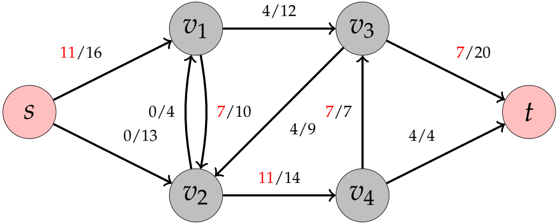


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

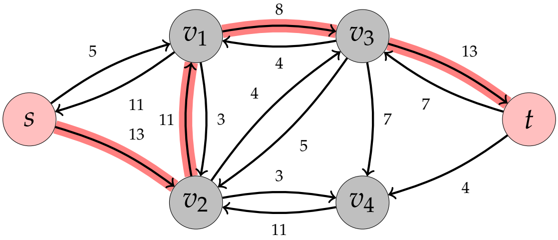


Figure: Residual network with an augmenting path from s to t.

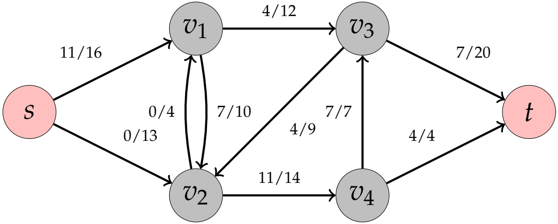


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

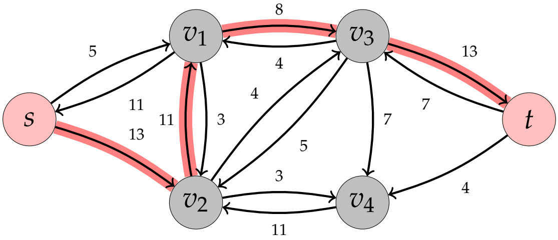


Figure: Residual network with an augmenting path from s to t.

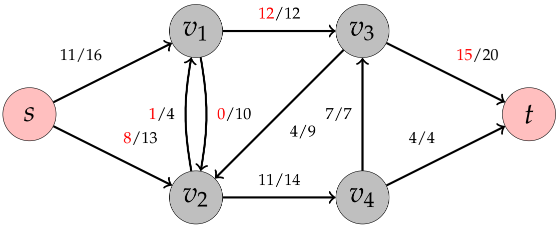


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

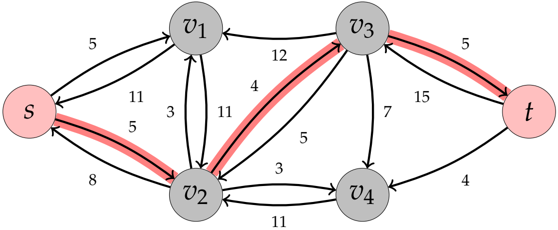


Figure: Residual network with an augmenting path from s to t.

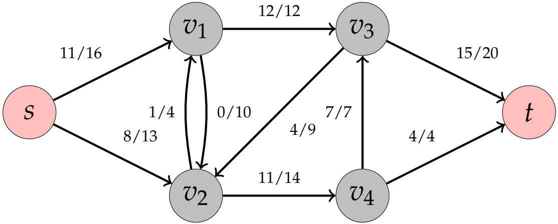


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

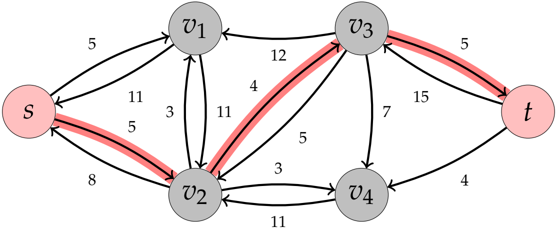


Figure: Residual network with an augmenting path from s to t.

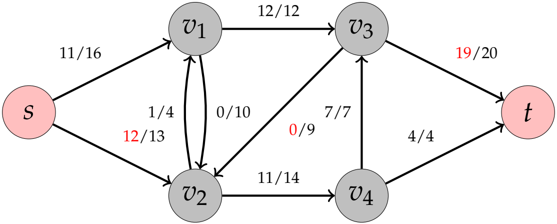


Figure: Network flow augmented along the path.

The basic Ford-Fulkerson algorithm – Example

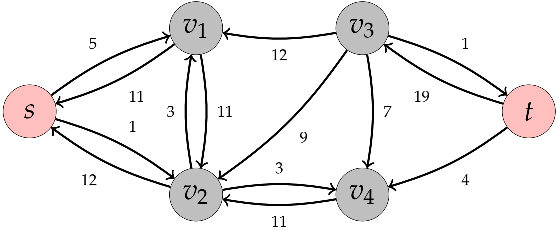


Figure: Residual network with an augmenting path from s to t.

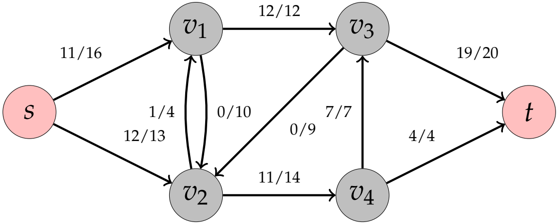


Figure: Network flow augmented along the path.

Maximum bipartite matching

Maximum bipartite matching

- Let $G = (V, E)$ be an undirected graph.

Maximum bipartite matching

- Let $G = (V, E)$ be an undirected graph.
- Matching in G is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.

Maximum bipartite matching

- Let $G = (V, E)$ be an undirected graph.
- Matching in G is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.
- A vertex is matched if some edge in M is incident on v; otherwise v is unmatched.

Maximum bipartite matching

- Let $G = (V, E)$ be an undirected graph.
- Matching in G is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.
- A vertex is matched if some edge in M is incident on v; otherwise v is unmatched.
- Maximum matching is a matching of maximum cardinality.

Maximum bipartite matching

- Let $G = (V, E)$ be an undirected graph.
- Matching in G is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.
- A vertex is matched if some edge in M is incident on v; otherwise v is unmatched.
- Maximum matching is a matching of maximum cardinality.
- We consider only connected bipartite graphs. That is, V can be partitioned into $V = L \cup R$, $R \cap L = \emptyset$ and $E \subseteq L \times R$.

Maximum bipartite matching

- Let $G = (V, E)$ be an undirected graph.
- Matching in G is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.
- A vertex is matched if some edge in M is incident on v; otherwise v is unmatched.
- Maximum matching is a matching of maximum cardinality.
- We consider only connected bipartite graphs. That is, V can be partitioned into $V = L \cup R$, $R \cap L = \emptyset$ and $E \subseteq L \times R$.
- We use the Ford-Fulkerson method to find maximum matching in a connected undirected bipartite graph.

Transformation to Maximum network flow problem

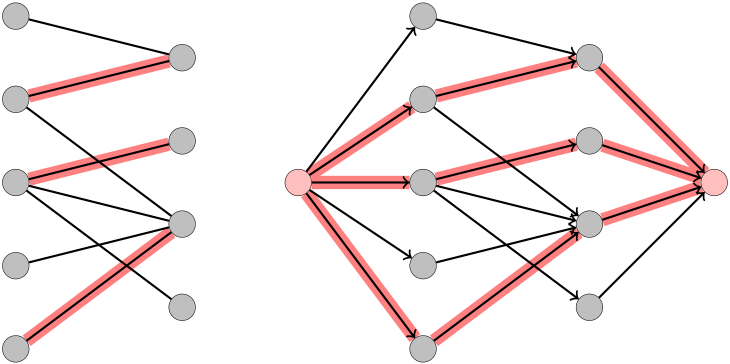


Figure: Bipartite graph and its flow network. Maximum matching and flow is highlighted (capacity of each edge is 1)

Transformation to Maximum network flow problem

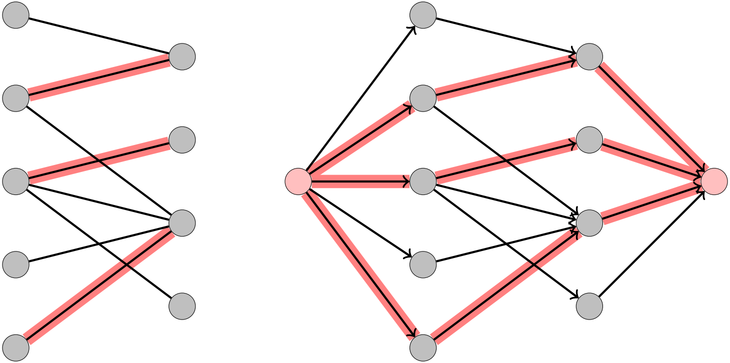


Figure: Bipartite graph and its flow network. Maximum matching and flow is highlighted (capacity of each edge is 1)

- Time complexity: $O(nm)$.

Graph Coloring

Notation

- Let $G = (V, E)$ be an undirected graph.

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.
- Formally, the coloring is a function

$$f : E \to B$$

($f : V \to B$), where B is a set of colors and $f(e_1) \neq f(e_2)$ for $e_1 \cap e_2 \neq \emptyset$ ($f(u) \neq f(v)$, if $\{u, v\}$ is an edge).

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.
- Formally, the coloring is a function

$$f : E \to B$$

  ($f : V \to B$), where B is a set of colors and $f(e_1) \neq f(e_2)$ for $e_1 \cap e_2 \neq \emptyset$ ($f(u) \neq f(v)$, if $\{u, v\}$ is an edge).
- Let $k \geq 0$. k-coloring is a coloring with $|B| = k$.

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.
- Formally, the coloring is a function

$$f : E \to B$$

($f : V \to B$), where B is a set of colors and $f(e_1) \neq f(e_2)$ for $e_1 \cap e_2 \neq \emptyset$ ($f(u) \neq f(v)$, if $\{u, v\}$ is an edge).
- Let $k \geq 0$. k-coloring is a coloring with $|B| = k$.
- $\psi_e(G)$ denotes the minimum number of colors necessary for edge coloring of G, called edge-chromatic index.

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.
- Formally, the coloring is a function

$$f : E \to B$$

  ($f : V \to B$), where B is a set of colors and $f(e_1) \neq f(e_2)$ for $e_1 \cap e_2 \neq \emptyset$ ($f(u) \neq f(v)$, if $\{u, v\}$ is an edge).
- Let $k \geq 0$. k-coloring is a coloring with $|B| = k$.
- $\psi_e(G)$ denotes the minimum number of colors necessary for edge coloring of G, called edge-chromatic index.
- $\psi_v(G)$ denotes the minimum number of colors necessary for (vertex) coloring of G, called vertex-chromatic index.

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.
- Formally, the coloring is a function

$$f : E \to B$$

($f : V \to B$), where B is a set of colors and $f(e_1) \neq f(e_2)$ for $e_1 \cap e_2 \neq \emptyset$ ($f(u) \neq f(v)$, if $\{u, v\}$ is an edge).
- Let $k \geq 0$. k-coloring is a coloring with $|B| = k$.
- $\psi_e(G)$ denotes the minimum number of colors necessary for edge coloring of G, called edge-chromatic index.
- $\psi_v(G)$ denotes the minimum number of colors necessary for (vertex) coloring of G, called vertex-chromatic index.
- Δ denotes the maximal degree of G.

Notation

- Let $G = (V, E)$ be an undirected graph.
- Goal: to colour edges (vertices) such that no two adjacent edges (adjacent vertices) has the same color.
- Formally, the coloring is a function

$$f : E \to B$$

($f : V \to B$), where B is a set of colors and $f(e_1) \neq f(e_2)$ for $e_1 \cap e_2 \neq \emptyset$ ($f(u) \neq f(v)$, if $\{u, v\}$ is an edge).
- Let $k \geq 0$. k-coloring is a coloring with $|B| = k$.
- $\psi_e(G)$ denotes the minimum number of colors necessary for edge coloring of G, called edge-chromatic index.
- $\psi_v(G)$ denotes the minimum number of colors necessary for (vertex) coloring of G, called vertex-chromatic index.
- Δ denotes the maximal degree of G.
- Graph-coloring problem: Determine $\psi_X(G)$ for a given graph, $X \in \{v, e\}$.

Edge Graph Coloring

Edge Graph Coloring

- Basic observation:

Edge Graph Coloring

- Basic observation:
- $\Delta \leq \psi_e(G)$.

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.
- $|E| = 1$ – obvious.

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.
- $|E| = 1$ – obvious.
- Assume that all edges but one are coloured using at most Δ colors.

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.
- $|E| = 1$ – obvious.
- Assume that all edges but one are coloured using at most Δ colors.
- The uncolored edge is (u, v).

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.
- $|E| = 1$ – obvious.
- Assume that all edges but one are coloured using at most Δ colors.
- The uncolored edge is (u, v).
- Since we can use Δ colors, at least one color is not incident to u and one is no incident to v.

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.
- $|E| = 1$ – obvious.
- Assume that all edges but one are coloured using at most Δ colors.
- The uncolored edge is (u, v).
- Since we can use Δ colors, at least one color is not incident to u and one is no incident to v.
- If they are the same, we are done.

Edge Coloring of Bipartite Graph

Theorem 28.
If G is bipartite, then $\psi_e(G) = \Delta$.

Proof

- By induction on the cardinality of set of edges.
- $|E| = 1$ – obvious.
- Assume that all edges but one are coloured using at most Δ colors.
- The uncolored edge is (u, v).
- Since we can use Δ colors, at least one color is not incident to u and one is no incident to v.
- If they are the same, we are done.
- If they differ, we label these colors by C_1 and C_2.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.
- Let $H_u(C_1, C_2)$ be a subgraph containing u and all edges reachable from u that are coloured only by C_1 and C_2.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.
- Let $H_u(C_1, C_2)$ be a subgraph containing u and all edges reachable from u that are coloured only by C_1 and C_2.
- Since (u, v) is an edge, u and v belongs to the different partite sets.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.
- Let $H_u(C_1, C_2)$ be a subgraph containing u and all edges reachable from u that are coloured only by C_1 and C_2.
- Since (u, v) is an edge, u and v belongs to the different partite sets.
- Then, every path from u to v in $H_u(C_1, C_2)$ must have the last edge coloured by C_2.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.
- Let $H_u(C_1, C_2)$ be a subgraph containing u and all edges reachable from u that are coloured only by C_1 and C_2.
- Since (u, v) is an edge, u and v belongs to the different partite sets.
- Then, every path from u to v in $H_u(C_1, C_2)$ must have the last edge coloured by C_2.
- But an edge with color C_2 is not incident to v, so v is not in $H_u(C_1, C_2)$.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.
- Let $H_u(C_1, C_2)$ be a subgraph containing u and all edges reachable from u that are coloured only by C_1 and C_2.
- Since (u, v) is an edge, u and v belongs to the different partite sets.
- Then, every path from u to v in $H_u(C_1, C_2)$ must have the last edge coloured by C_2.
- But an edge with color C_2 is not incident to v, so v is not in $H_u(C_1, C_2)$.
- By the exchange of C_1 and C_2 in $H_u(C_1, C_2)$ we get that C_2 is not incident to u.

Edge Coloring of Bipartite Graph

- The colors not incident to u and v are denoted by C_1 and C_2, respectively.
- Let $H_u(C_1, C_2)$ be a subgraph containing u and all edges reachable from u that are coloured only by C_1 and C_2.
- Since (u, v) is an edge, u and v belongs to the different partite sets.
- Then, every path from u to v in $H_u(C_1, C_2)$ must have the last edge coloured by C_2.
- But an edge with color C_2 is not incident to v, so v is not in $H_u(C_1, C_2)$.
- By the exchange of C_1 and C_2 in $H_u(C_1, C_2)$ we get that C_2 is not incident to u.
- Then, we can paint (u, v) by C_2. □

Edge Coloring of Complete Graph

Theorem 29.

If G is complete with n vertices, then $\psi_e(G) = \begin{cases} \Delta & n \text{ even} \\ \Delta + 1 & n \text{ odd} \end{cases}$

Proof

- Case 1: If n is odd, draw a graph as regular polygon (see below).

Edge Coloring of Complete Graph

Theorem 29.

If G is complete with n vertices, then $\psi_e(G) = \begin{cases} \Delta & n \text{ even} \\ \Delta + 1 & n \text{ odd} \end{cases}$

Proof

- Case 1: If n is odd, draw a graph as regular polygon (see below).
- We paint border edges by colors $1, 2, \ldots, n = \Delta + 1$.

Edge Coloring of Complete Graph

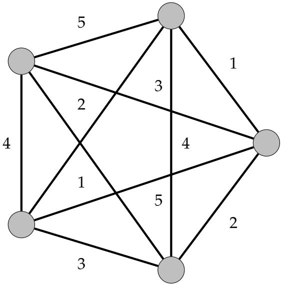
Theorem 29.

If G is complete with n vertices, then $\psi_e(G) = \begin{cases} \Delta & n \text{ even} \\ \Delta + 1 & n \text{ odd} \end{cases}$

Proof

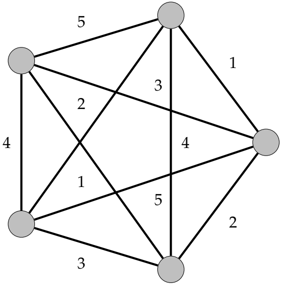
- Case 1: If n is odd, draw a graph as regular polygon (see below).
- We paint border edges by colors $1, 2, \ldots, n = \Delta + 1$.
- Paint every inner edge to the same color as its parallel border edge.

Edge Coloring of Complete Graph



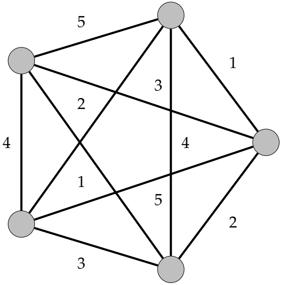
Edge Coloring of Complete Graph

- No Δ-coloring for a complete graph with odd n $(\Delta = n - 1)$.



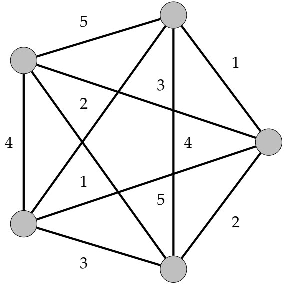
Edge Coloring of Complete Graph

- No Δ-coloring for a complete graph with odd n $(\Delta = n - 1)$.
- Assume it is possible. Then, if G has $\frac{1}{2}n(n-1)$ edges, we have at least $\frac{1}{2}n$ edges of the same color.



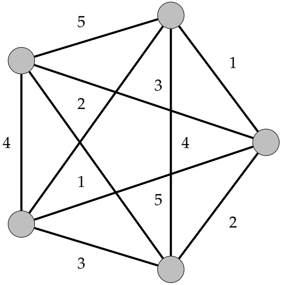
Edge Coloring of Complete Graph

- No Δ-coloring for a complete graph with odd n $(\Delta = n-1)$.
- Assume it is possible. Then, if G has $\frac{1}{2}n(n-1)$ edges, we have at least $\frac{1}{2}n$ edges of the same color.
- Let $M \subseteq E$ such that no two edges from M are incident to the same vertex.



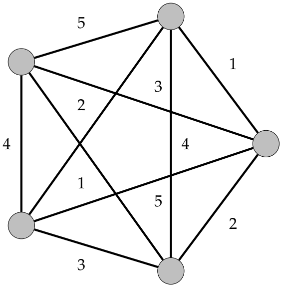
Edge Coloring of Complete Graph

- No Δ-coloring for a complete graph with odd n $(\Delta = n - 1)$.
- Assume it is possible. Then, if G has $\frac{1}{2}n(n-1)$ edges, we have at least $\frac{1}{2}n$ edges of the same color.
- Let $M \subseteq E$ such that no two edges from M are incident to the same vertex.
- Therefore, $|M| \leq \frac{1}{2}(n-1)$ – (prove as a homework).



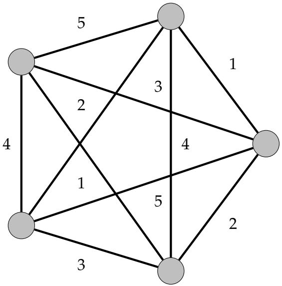
Edge Coloring of Complete Graph

- Case 2: Let n be even.



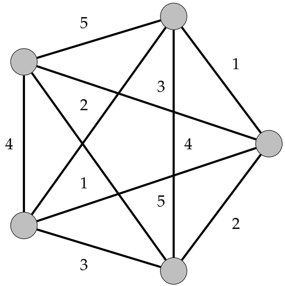
Edge Coloring of Complete Graph

- Case 2: Let n be even.
- Describe G as the complete graph G' with $n-1$ vertices + one more vertex connected to all others.



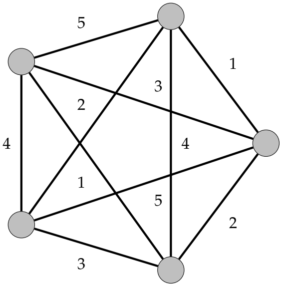
Edge Coloring of Complete Graph

- Case 2: Let n be even.
- Describe G as the complete graph G' with $n-1$ vertices + one more vertex connected to all others.
- Use the procedure from Case 1 on G'.



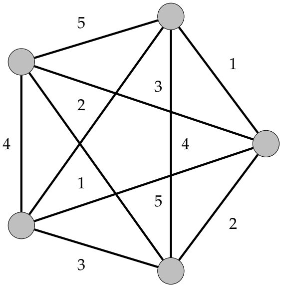
Edge Coloring of Complete Graph

- Case 2: Let n be even.
- Describe G as the complete graph G' with $n-1$ vertices + one more vertex connected to all others.
- Use the procedure from Case 1 on G'.
- There is one unused color in each vertex.



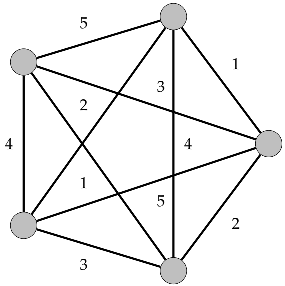
Edge Coloring of Complete Graph

- Case 2: Let n be even.
- Describe G as the complete graph G' with $n-1$ vertices + one more vertex connected to all others.
- Use the procedure from Case 1 on G'.
- There is one unused color in each vertex.
- All these colors are mutually different, so we can use them to paint the edges of "$G - G'$".



Edge Coloring of Complete Graph

- Case 2: Let n be even.
- Describe G as the complete graph G' with $n-1$ vertices + one more vertex connected to all others.
- Use the procedure from Case 1 on G'.
- There is one unused color in each vertex.
- All these colors are mutually different, so we can use them to paint the edges of "$G - G'$".
- In the end, we used at most $\Delta = n - 1$ colors. □



Edge Coloring of Undirected Graph

Theorem 30.
Let G is simple graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We need to show that $\psi_e(G) \leq \Delta + 1$.

Edge Coloring of Undirected Graph

Theorem 30.
Let G is simple graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We need to show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.

Edge Coloring of Undirected Graph

Theorem 30.
Let G is simple graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We need to show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.
- The principle is similar to the proof for bipartite graphs.

Edge Coloring of Undirected Graph

Theorem 30.
Let G is simple graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We need to show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.
- The principle is similar to the proof for bipartite graphs.
- See Chapter 7 in [Gibbons, 1985].

Edge Coloring of Undirected Graph

Theorem 31.
Let G be an undirected graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$*.*

Proof

- We must show that $\psi_e(G) \leq \Delta + 1$.

Edge Coloring of Undirected Graph

Theorem 31.
Let G be an undirected graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We must show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.

Edge Coloring of Undirected Graph

Theorem 31.
Let G be an undirected graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We must show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.
- Induction basis: For one edge, it holds trivially.

Edge Coloring of Undirected Graph

Theorem 31.
Let G be an undirected graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We must show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.
- Induction basis: For one edge, it holds trivially.
- Let all edges except an edge (v_0, v_1) are colored by at most $\Delta + 1$ colors.

Edge Coloring of Undirected Graph

Theorem 31.
Let G be an undirected graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We must show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.
- Induction basis: For one edge, it holds trivially.
- Let all edges except an edge (v_0, v_1) are colored by at most $\Delta + 1$ colors.
- At least one color is missing in v_0 and one is missing in v_1.

Edge Coloring of Undirected Graph

Theorem 31.
Let G be an undirected graph. Then, $\Delta \leq \psi_e(G) \leq \Delta + 1$.

Proof

- We must show that $\psi_e(G) \leq \Delta + 1$.
- By induction on the number of edges.
- Induction basis: For one edge, it holds trivially.
- Let all edges except an edge (v_0, v_1) are colored by at most $\Delta + 1$ colors.
- At least one color is missing in v_0 and one is missing in v_1.
- If both missing colors are the same, we are done.

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and
  - (v_0, v_{i+1}) is colored by C_i.

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and
  - (v_0, v_{i+1}) is colored by C_i.
- So we have sequence $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_i)$ and $C_1, C_2, C_3, \ldots, C_i$, for some $i \geq 0$.

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and
  - (v_0, v_{i+1}) is colored by C_i.
- So we have sequence $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_i)$ and $C_1, C_2, C_3, \ldots, C_i$, for some $i \geq 0$.
- Notice that there is at most one edge, (v_0, v), colored by C_i.

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and
  - (v_0, v_{i+1}) is colored by C_i.
- So we have sequence $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_i)$ and $C_1, C_2, C_3, \ldots, C_i$, for some $i \geq 0$.
- Notice that there is at most one edge, (v_0, v), colored by C_i.
  - If there is such v and $v \notin \{v_1, v_2, \ldots, v_i\}$, then add (v_0, v_{i+1}) to the sequence, where $v_{i+1} = v$ and C_{i+1} is missing in v_{i+1}.

Edge Coloring of Undirected Graph

- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and
  - (v_0, v_{i+1}) is colored by C_i.
- So we have sequence $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_i)$ and $C_1, C_2, C_3, \ldots, C_i$, for some $i \geq 0$.
- Notice that there is at most one edge, (v_0, v), colored by C_i.
  - If there is such v and $v \notin \{v_1, v_2, \ldots, v_i\}$, then add (v_0, v_{i+1}) to the sequence, where $v_{i+1} = v$ and C_{i+1} is missing in v_{i+1}.
  - Otherwise, the sequence is finished.

Edge Coloring of Undirected Graph

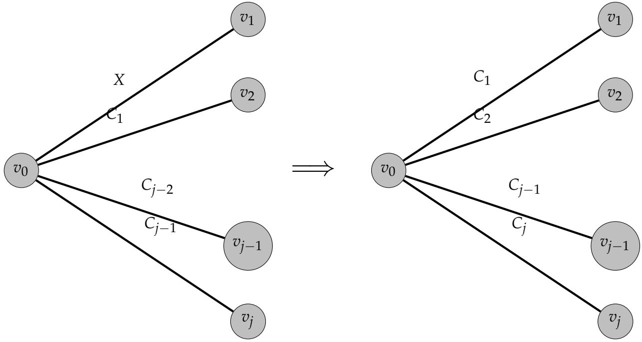
- Let C_0, C_1 be the colors missing in v_0, v_1, respectively.
- Construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots$ such that
  - C_i is missing in v_i and
  - (v_0, v_{i+1}) is colored by C_i.
- So we have sequence $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_i)$ and $C_1, C_2, C_3, \ldots, C_i$, for some $i \geq 0$.
- Notice that there is at most one edge, (v_0, v), colored by C_i.
  - If there is such v and $v \notin \{v_1, v_2, \ldots, v_i\}$, then add (v_0, v_{i+1}) to the sequence, where $v_{i+1} = v$ and C_{i+1} is missing in v_{i+1}.
  - Otherwise, the sequence is finished.
- Such sequence has always at most Δ edges.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.
  - i) If there is no (v_0, v) colored by C_j, so we do the recoloring $(X \neq C_j)$:



Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.
  - ii) If there is $k < j$ such that (v_0, v_k) is colored by C_j.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \dots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \dots, C_j$, for some $j \geq 0$.
  ii) If there is $k < j$ such that (v_0, v_k) is colored by C_j.
  - Then, for $i < k$, we recolor edges (see above), so (v_0, v_i) is colored by C_i.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.
  - ii) If there is $k < j$ such that (v_0, v_k) is colored by C_j.
  - Then, for $i < k$, we recolor edges (see above), so (v_0, v_i) is colored by C_i.
  - (v_0, v_k) remains uncolored.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.
  - ii) If there is $k < j$ such that (v_0, v_k) is colored by C_j.
  - Then, for $i < k$, we recolor edges (see above), so (v_0, v_i) is colored by C_i.
  - (v_0, v_k) remains uncolored.
- Every component of $H(C_0, C_j)$ – subgraph with all edges of colors C_0 and C_j – is either a path, or a cycle, because every vertex is adjacent to at most one edge of color C_0 and one of C_j.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \dots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \dots, C_j$, for some $j \geq 0$.
  ii) If there is $k < j$ such that (v_0, v_k) is colored by C_j.
  - Then, for $i < k$, we recolor edges (see above), so (v_0, v_i) is colored by C_i.
  - (v_0, v_k) remains uncolored.
- Every component of $H(C_0, C_j)$ – subgraph with all edges of colors C_0 and C_j – is either a path, or a cycle, because every vertex is adjacent to at most one edge of color C_0 and one of C_j.
- At least one of C_0, C_j is not in v_0, v_k, v_j.

Edge Coloring of Undirected Graph

- Let $(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_j)$ be the built sequence and $C_1, C_2, C_3, \ldots, C_j$, for some $j \geq 0$.
  - ii) If there is $k < j$ such that (v_0, v_k) is colored by C_j.
  - Then, for $i < k$, we recolor edges (see above), so (v_0, v_i) is colored by C_i.
  - (v_0, v_k) remains uncolored.
- Every component of $H(C_0, C_j)$ – subgraph with all edges of colors C_0 and C_j – is either a path, or a cycle, because every vertex is adjacent to at most one edge of color C_0 and one of C_j.
- At least one of C_0, C_j is not in v_0, v_k, v_j.
- So not all can be in a single component of $H(C_0, C_j)$:
  $v_0 \overset{C_j}{\to} x \overset{X}{\to} y \ldots \overset{C_0}{\to} v_k$ and we do not reach v_j.

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

b) $v_0 \notin H_{v_j}(C_0, C_j)$, so we do recolor

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

b) $v_0 \notin H_{v_j}(C_0, C_j)$, so we do recolor

  - (v_0, v_i) by C_i, $k \leq i < j$,

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

b) $v_0 \notin H_{v_j}(C_0, C_j)$, so we do recolor

  - (v_0, v_i) by C_i, $k \leq i < j$,
  - (v_0, v_j) remains uncolored.

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

b) $v_0 \notin H_{v_j}(C_0, C_j)$, so we do recolor

  - (v_0, v_i) by C_i, $k \leq i < j$,
  - (v_0, v_j) remains uncolored.

- In the recoloring, neither C_0, nor C_j was used, so $H(C_0, C_j)$ is unchanged.

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

b) $v_0 \notin H_{v_j}(C_0, C_j)$, so we do recolor
   - (v_0, v_i) by C_i, $k \le i < j$,
   - (v_0, v_j) remains uncolored.

- In the recoloring, neither C_0, nor C_j was used, so $H(C_0, C_j)$ is unchanged.
- Again, $C_0 \leftrightarrow C_j$ v $H_{v_j}(C_0, C_j)$ and C_0 is missing in v_j.

Edge Coloring of Undirected Graph

a) $v_0 \notin H_{v_k}(C_0, C_j)$ – component of $H(C_0, C_j)$ contains v_k – then $C_0 \leftrightarrow C_j$ in $H_{v_k}(C_0, C_j)$, therefore C_0 is missing in v_k.

- C_0 is missing in v_0 as well, so we color (v_0, v_k) by C_0.

b) $v_0 \notin H_{v_j}(C_0, C_j)$, so we do recolor

  - (v_0, v_i) by C_i, $k \leq i < j$,
  - (v_0, v_j) remains uncolored.

- In the recoloring, neither C_0, nor C_j was used, so $H(C_0, C_j)$ is unchanged.
- Again, $C_0 \leftrightarrow C_j$ v $H_{v_j}(C_0, C_j)$ and C_0 is missing in v_j.
- So color (v_0, v_j) by C_0. □

Edge Coloring of Undirected Graph

- Based on the proof, we can introduce a polynomial algorithm.

Edge Coloring of Undirected Graph

- Based on the proof, we can introduce a polynomial algorithm.
- But problem whether $\psi_e(G) = \Delta$ is NP-complete.

Approximation for Edge Coloring

1. Add edges to G to get $K_{|V|}$.

Approximation for Edge Coloring

1. Add edges to G to get $K_{|V|}$.
2. Find proper edge-coloring for the complete graph (Δ or $\Delta + 1$ colors needed).

Approximation for Edge Coloring

1. Add edges to G to get $K_{|V|}$.
2. Find proper edge-coloring for the complete graph (Δ or $\Delta + 1$ colors needed).
3. Delete edges added to G in step 1.

Approximation for Edge Coloring

1. Add edges to G to get $K_{|V|}$.
2. Find proper edge-coloring for the complete graph (Δ or $\Delta + 1$ colors needed).
3. Delete edges added to G in step 1.

- We get k-edge-coloring with $k \leq n$, but $\psi_e(G)$ can be significantly smaller than k.

Approximation for Edge Coloring

1. Add edges to G to get $K_{|V|}$.
2. Find proper edge-coloring for the complete graph (Δ or $\Delta + 1$ colors needed).
3. Delete edges added to G in step 1.

- We get k-edge-coloring with $k \leq n$, but $\psi_e(G)$ can be significantly smaller than k.
- Time complexity: $O(n^2)$

(Vertex) Graph Coloring

Graph Coloring

- NP-Complete problem: Can we find a proper k-coloring of G?

Graph Coloring

Theorem 32.
Any (simple) graph G has $\Delta + 1$-coloring.

Proof.

- By induction on n.

□

Graph Coloring

Theorem 32.
Any (simple) graph G has $\Delta + 1$-coloring.

Proof.

- By induction on n.
- $n = 1$, obvious.

□

Graph Coloring

Theorem 32.
Any (simple) graph G has $\Delta + 1$-coloring.

Proof.

- By induction on n.
- $n = 1$, obvious.
- If we add vertex u, then it is connected with at most Δ other vertices.

□

Graph Coloring

Theorem 32.
Any (simple) graph G has $\Delta + 1$-coloring.

Proof.

- By induction on n.
- $n = 1$, obvious.
- If we add vertex u, then it is connected with at most Δ other vertices.
- Since we have $\Delta + 1$ colors, we have one spare color to paint u.

□

Graph Coloring

- In most cases: $\psi_v(G) < \Delta + 1$.

Graph Coloring

- In most cases: $\psi_v(G) < \Delta + 1$.
- Example:

Graph Coloring

- In most cases: $\psi_v(G) < \Delta + 1$.
- Example:
- If G is planar, then $\psi_v(G) \leq 4$, but Δ can be arbitrary.

Graph Coloring

- In most cases: $\psi_v(G) < \Delta + 1$.
- Example:
- If G is planar, then $\psi_v(G) \leq 4$, but Δ can be arbitrary.

- Homework: Design your own algorithm to find some proper coloring of a given graph?

Chromatic polynomial

Chromatic polynomial

- $P_k(G)$ – chromatic polynomial of G;
  determines the number of ways of proper vertex-coloring of G with k colors.

Chromatic polynomial

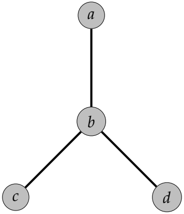


Figure: Graph G_1.

- b ... picks up one of k colors.

Chromatic polynomial

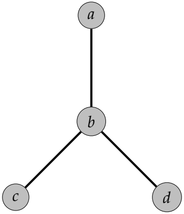


Figure: Graph G_1.

- b ... picks up one of k colors.
- a, c, d ... pick up any of $k-1$ remaining colors.

Chromatic polynomial

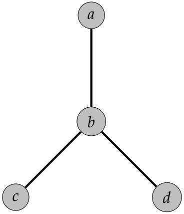


Figure: Graph G_1.

- b ... picks up one of k colors.
- a, c, d ... pick up any of $k - 1$ remaining colors.
- $P_k(G_1) = k(k-1)^3$

Chromatic polynomial

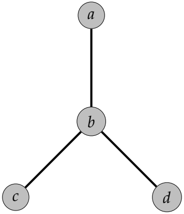


Figure: Graph G_1.

- b ... picks up one of k colors.
- a, c, d ... pick up any of $k-1$ remaining colors.
- $P_k(G_1) = k(k-1)^3$
- In general, let T_n be a tree with n vertices. Then, $P_k(T_n) = k(k-1)^{n-1}$.

Chromatic polynomial

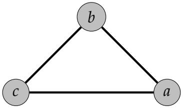


Figure: Graph G_2.

- a ... paint it to any of k colors.

Chromatic polynomial

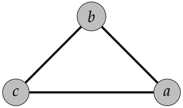


Figure: Graph G_2.

- a ... paint it to any of k colors.
- b ... paint it to any of $k-1$ remaining colors.

Chromatic polynomial

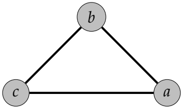


Figure: Graph G_2.

- a . . . paint it to any of k colors.
- b . . . paint it to any of $k-1$ remaining colors.
- c . . . paint it to any of $k-2$ remaining colors.

Chromatic polynomial

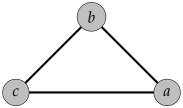


Figure: Graph G_2.

- a ... paint it to any of k colors.
- b ... paint it to any of $k-1$ remaining colors.
- c ... paint it to any of $k-2$ remaining colors.
- $P_k(G_2) = k(k-1)(k-2)$

Chromatic polynomial

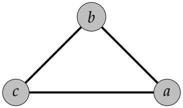


Figure: Graph G_2.

- a ... paint it to any of k colors.
- b ... paint it to any of $k-1$ remaining colors.
- c ... paint it to any of $k-2$ remaining colors.
- $P_k(G_2) = k(k-1)(k-2)$
- In general, let K_n be a complete graph with n vertices.

Chromatic polynomial

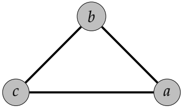


Figure: Graph G_2.

- a ... paint it to any of k colors.
- b ... paint it to any of $k-1$ remaining colors.
- c ... paint it to any of $k-2$ remaining colors.
- $P_k(G_2) = k(k-1)(k-2)$
- In general, let K_n be a complete graph with n vertices.
- Then, $P_k(K_n) = \frac{k!}{(k-n)!}$

Chromatic polynomial

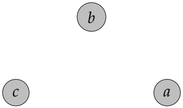


Figure: Graph G_2'.

- a . . . gets arbitrary one of k colors.

Chromatic polynomial

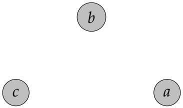


Figure: Graph G_2'.

- a . . . gets arbitrary one of k colors.
- b . . . gets arbitrary one of k colors.

Chromatic polynomial

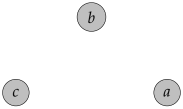


Figure: Graph G_2'.

- a ... gets arbitrary one of k colors.
- b ... gets arbitrary one of k colors.
- c ... gets arbitrary one of k colors.

Chromatic polynomial

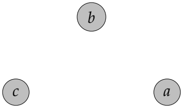


Figure: Graph G_2'.

- a ... gets arbitrary one of k colors.
- b ... gets arbitrary one of k colors.
- c ... gets arbitrary one of k colors.
- $P_k(G_2') = k^3$

Chromatic polynomial

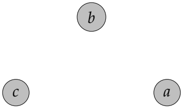


Figure: Graph G_2'.

- a ... gets arbitrary one of k colors.
- b ... gets arbitrary one of k colors.
- c ... gets arbitrary one of k colors.
- $P_k(G_2') = k^3$
- In general, let Φ_n be an isolated graph with n vertices; that is, $deg(v) = 0$ for all $v \in V$.

Chromatic polynomial

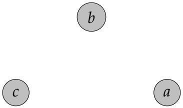


Figure: Graph G_2'.

- a ... gets arbitrary one of k colors.
- b ... gets arbitrary one of k colors.
- c ... gets arbitrary one of k colors.
- $P_k(G_2') = k^3$
- In general, let Φ_n be an isolated graph with n vertices; that is, $deg(v) = 0$ for all $v \in V$.
- Then, $P_k(\Phi_n) = k^n$

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.
- Let G be an undirected graph.

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.

- Let G be an undirected graph.
- How to construct $P_k(G)$?

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.

- Let G be an undirected graph.
- How to construct $P_k(G)$?

- Notation:

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.

- Let G be an undirected graph.
- How to construct $P_k(G)$?

- Notation:
- $G - (u, v)$ ... subgraph of G where just edge (u, v) was removed.

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.

- Let G be an undirected graph.
- How to construct $P_k(G)$?

- Notation:
- $G - (u,v)$ ... subgraph of G where just edge (u,v) was removed.
- $G + (u,v)$ ... graph created by adding (u,v) to G.

Chromatic polynomial

- Observation: If $k < \psi_v(G)$, then $P_k(G) = 0$.

- Let G be an undirected graph.
- How to construct $P_k(G)$?

- Notation:
- $G - (u, v)$ ... subgraph of G where just edge (u, v) was removed.
- $G + (u, v)$ ... graph created by adding (u, v) to G.
- $G \circ (u, v)$ ... graph created from G by contracting (u, v).

Chromatic polynomial – Subtracting Recursion Formula

Theorem 33.
Let (u, v) *be an edge in* G*, then*

$$P_k(G) = P_k(G - (u, v)) - P_k(G \circ (u, v))\,.$$

Proof.

- $P_k(G)$ denotes the number of colorings where u and v has different color.

□

Chromatic polynomial – Subtracting Recursion Formula

Theorem 33.
Let (u, v) *be an edge in* G, *then*

$$P_k(G) = P_k(G - (u, v)) - P_k(G \circ (u, v)) .$$

Proof.

- $P_k(G)$ denotes the number of colorings where u and v has different color.
- All these colorings are also covered by $P_k(G - (u, v))$.

□

Chromatic polynomial – Subtracting Recursion Formula

Theorem 33.
Let (u,v) *be an edge in* G*, then*

$$P_k(G) = P_k(G - (u,v)) - P_k(G \circ (u,v)) \, .$$

Proof.

- $P_k(G)$ denotes the number of colorings where u and v has different color.
- All these colorings are also covered by $P_k(G - (u,v))$.
- In addition, $P_k(G - (u,v))$ covers also the colorings where u and v has the same color.

□

Chromatic polynomial – Subtracting Recursion Formula

Theorem 33.
Let (u, v) be an edge in G, then

$$P_k(G) = P_k(G - (u, v)) - P_k(G \circ (u, v)).$$

Proof.

- $P_k(G)$ denotes the number of colorings where u and v has different color.
- All these colorings are also covered by $P_k(G - (u, v))$.
- In addition, $P_k(G - (u, v))$ covers also the colorings where u and v has the same color.
- So, we subtract them using polynomial $P_k(G \circ (u, v))$.

□

Chromatic polynomial – Example

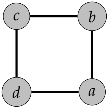


Figure: Graph G_3.

- $P_k(G_3) = P_k(\Phi_4) - 4P_k(\Phi_3) + 6P_k(\Phi_2) - 3P_k(\Phi_1)$

Chromatic polynomial – Example

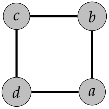


Figure: Graph G_3.

- $$P_k(G_3) = P_k(\Phi_4) - 4P_k(\Phi_3) + 6P_k(\Phi_2) - 3P_k(\Phi_1)$$
  $$= k(k-1)(k^2 - 3k + 3)$$

Chromatic polynomial – Adding Recursive Formula

- If G is dense, there is better variant of the construction:

Chromatic polynomial – Adding Recursive Formula

- If G is dense, there is better variant of the construction:
- $P_k(G) = P_k(G + (u, v)) + P_k((G + (u, v)) \circ (u, v))$

Chromatic polynomial – Adding Recursive Formula

- If G is dense, there is better variant of the construction:
- $P_k(G) = P_k(G + (u,v)) + P_k((G + (u,v)) \circ (u,v))$
- That is, we add new edges until we reach complete graphs as addends.

Chromatic polynomial – Example

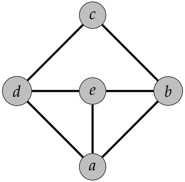


Figure: Graph G_4.

- $P_k(G_4) = P_k(K_5) + 3P_k(K_4) + 2P_k(K_3)$

Chromatic polynomial – Example

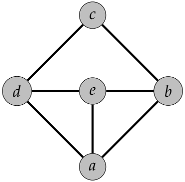


Figure: Graph G_4.

- $$\begin{aligned} P_k(G_4) &= P_k(K_5) + 3P_k(K_4) + 2P_k(K_3) \\ &= k(k-1)(k-2)(k^2-4k+5) \end{aligned}$$

Chromatic polynomial and vertex-chromatic index

- From $P_k(G)$, we can determine $\psi_v(G)$ as minimum k such that $P_k(G) > 0$.

Chromatic polynomial and vertex-chromatic index

- From $P_k(G)$, we can determine $\psi_v(G)$ as minimum k such that $P_k(G) > 0$.
- $\psi_v(G_3) = 2$

Chromatic polynomial and vertex-chromatic index

- From $P_k(G)$, we can determine $\psi_v(G)$ as minimum k such that $P_k(G) > 0$.

- $\psi_v(G_3) = 2$
- $\psi_v(G_4) =?$

Chromatic polynomial and vertex-chromatic index

- From $P_k(G)$, we can determine $\psi_v(G)$ as minimum k such that $P_k(G) > 0$.

- $\psi_v(G_3) = 2$
- $\psi_v(G_4) = ?$

- What is the time complexity of building chromatic polynomial?

Chromatic polynomial and vertex-chromatic index

- From $P_k(G)$, we can determine $\psi_v(G)$ as minimum k such that $P_k(G) > 0$.
- $\psi_v(G_3) = 2$
- $\psi_v(G_4) = ?$
- What is the time complexity of building chromatic polynomial?
  For $k > 3$, $O(2^n n^r)$ for some constant r.

Approximate Sequential Vertex Coloring

- Lawler Algorithm for Vertex-coloring – $O(n\, m\, k^n)$, where $k = 1 + \sqrt[3]{3}$

Approximate Sequential Vertex Coloring

- Lawler Algorithm for Vertex-coloring – $O(n\,m\,k^n)$, where $k = 1 + \sqrt[3]{3}$
- What about an approximate algorithm?

Approximate Sequential Vertex Coloring

- Lawler Algorithm for Vertex-coloring – $O(n\,m\,k^n)$, where $k = 1 + \sqrt[3]{3}$
- What about an approximate algorithm?

Approximate Sequential Vertex Coloring

- Lawler Algorithm for Vertex-coloring – $O(n\,m\,k^n)$, where $k = 1 + \sqrt[3]{3}$
- What about an approximate algorithm?

```
APPROXIMATE-SEQUENTIAL-VERTEX-COLORING(G)
for each vertex u ∈ V
    do for c ← 1 to Δ + 1
           do N[u, c] ← FALSE
for each vertex u ∈ V
    do c ← 1
       while N[u, c] = TRUE
          do c ← c + 1
       for each v ∈ Adj[u]
          do N[v, c] ← TRUE
       color[u] ← c
```

- Time Complexity: $O(n^2)$

Approximate Sequential Vertex Coloring

- Lawler Algorithm for Vertex-coloring – $O(n\,m\,k^n)$, where $k = 1 + \sqrt[3]{3}$
- What about an approximate algorithm?

```
APPROXIMATE-SEQUENTIAL-VERTEX-COLORING(G)
for each vertex u ∈ V
     do for c ← 1 to Δ + 1
            do N[u, c] ← FALSE
for each vertex u ∈ V
     do c ← 1
        while N[u, c] = TRUE
           do c ← c + 1
        for each v ∈ Adj[u]
           do N[v, c] ← TRUE
        color[u] ← c
```

- Time Complexity: $O(n^2)$
- Performance ratio $\text{A-S-V-C}(G)/\psi_v(G)$ is non-constant.

Exercises

1. Consider 3×3 chessboard represented as a graph with 9 vertices where an undirected edge (u, v) represents that a chess piece placed at u dominates v (it can attack the other piece at v) and vice versa. Use graph coloring to determine how many queens we can place on this chessboard so they do not attack each other.
2. Derive chromatic polynomial using subtracting formula for the complete graph with 4 vertices.
3. Derive chromatic polynomial using adding formula for the isolated graph with 4 vertices.
4. Use approximate vertex coloring algorithm for a bipartite graph with $L = \{u_1, u_2, \ldots, u_k\}$, $R = \{v_1, v_2, \ldots, v_k\}$, and $E = \{(u_i, v_j) : i \neq j\}$, $k \geq 2$. First, consider the vertices are colored in the order u_1, u_2, ..., u_k, v_1, v_2, ..., v_k. Second, apply the algorithm in the other order u_1, v_1, u_2, v_2, ..., u_k, v_k. Compare the results.

Eulerian Tours

L. Euler and W. R. Hamilton

- Leonhard Euler (1707 – 1783, Swiss mathematician)

L. Euler and W. R. Hamilton

- Leonhard Euler (1707 – 1783, Swiss mathematician)
  - The Königsberge bridges problem

L. Euler and W. R. Hamilton

- Leonhard Euler (1707 – 1783, Swiss mathematician)
  - The Königsberge bridges problem
  - Graph exploration that walks every edge exactly once.

L. Euler and W. R. Hamilton

- Leonhard Euler (1707 – 1783, Swiss mathematician)
  - The Königsberge bridges problem
  - Graph exploration that walks every edge exactly once.
- William Rowan Hamilton (1805 – 1865, British mathematician)

L. Euler and W. R. Hamilton

- Leonhard Euler (1707 – 1783, Swiss mathematician)
  - The Königsberge bridges problem
  - Graph exploration that walks every edge exactly once.
- William Rowan Hamilton (1805 – 1865, British mathematician)
  - a game how to plan a journey through 20 cities as the tips on the regular dodecahedron

L. Euler and W. R. Hamilton

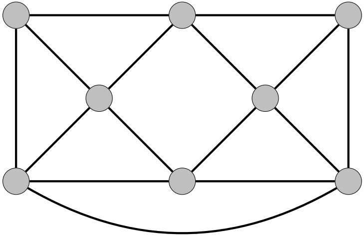
- Leonhard Euler (1707 – 1783, Swiss mathematician)
  - The Königsberge bridges problem
  - Graph exploration that walks every edge exactly once.
- William Rowan Hamilton (1805 – 1865, British mathematician)
  - a game how to plan a journey through 20 cities as the tips on the regular dodecahedron
  - Graph exploration that walks through every vertex exactly once.

L. Euler and W. R. Hamilton

- Leonhard Euler (1707 – 1783, Swiss mathematician)
  - The Königsberge bridges problem
  - Graph exploration that walks every edge exactly once.
- William Rowan Hamilton (1805 – 1865, British mathematician)
  - a game how to plan a journey through 20 cities as the tips on the regular dodecahedron
  - Graph exploration that walks through every vertex exactly once.
- Definition note: Tour = path or circuit; Cycle/Circuit = closed path

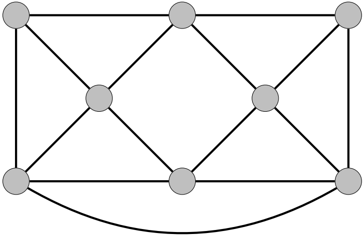
Eulerian graph

- Eulerian graph is a graph that contains an Eulerian circuit; that is, a closed path that visits all edges exactly once.



Eulerian graph

- Eulerian graph is a graph that contains an Eulerian circuit; that is, a closed path that visits all edges exactly once.
- Note that Eulerian path does not have to be closed, but then the graph is not Eulerian.



Theorem: Existence of an Eulerian tour

Theorem 34.

An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.

Theorem: Existence of an Eulerian tour

Theorem 34.
An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.
- *Sufficient condition*: By induction on the number of edges in $|E|$.

Theorem: Existence of an Eulerian tour

Theorem 34.
An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.
- *Sufficient condition*: By induction on the number of edges in $|E|$.
- Assume that $G = (V_G, E_G)$ with $|E_G| > 2$ satisfies this theorem.

Theorem: Existence of an Eulerian tour

Theorem 34.
An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.
- *Sufficient condition*: By induction on the number of edges in $|E|$.
- Assume that $G = (V_G, E_G)$ with $|E_G| > 2$ satisfies this theorem.
- If there are odd-degree vertices in G, denote them v_1, v_2.

Theorem: Existence of an Eulerian tour

Theorem 34.
An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.
- *Sufficient condition*: By induction on the number of edges in $|E|$.
- Assume that $G = (V_G, E_G)$ with $|E_G| > 2$ satisfies this theorem.
- If there are odd-degree vertices in G, denote them v_1, v_2.
- Consider any exploration of G by closed (or open) tour $T = (V_G, E_T)$ from vertex v_i (or v_1) until we reach vertex v_j from which we cannot continue without repeating an edge (no unused incident edge).

Theorem: Existence of an Eulerian tour

Theorem 34.
An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.
- *Sufficient condition*: By induction on the number of edges in $|E|$.
- Assume that $G = (V_G, E_G)$ with $|E_G| > 2$ satisfies this theorem.
- If there are odd-degree vertices in G, denote them v_1, v_2.
- Consider any exploration of G by closed (or open) tour $T = (V_G, E_T)$ from vertex v_i (or v_1) until we reach vertex v_j from which we cannot continue without repeating an edge (no unused incident edge).
  (a) If no odd-degree vertex then $v_i = v_j$;

Theorem: Existence of an Eulerian tour

Theorem 34.
An undirected graph G, has an Eulerian tour if and only if it is connected and the number of odd-degree vertices is 0 or 2.

Proof

- *Necessary condition*: If an Eulerian path exists in G then G must be connected and only vertices on the ends of the path can be of odd-degree.
- *Sufficient condition*: By induction on the number of edges in $|E|$.
- Assume that $G = (V_G, E_G)$ with $|E_G| > 2$ satisfies this theorem.
- If there are odd-degree vertices in G, denote them v_1, v_2.
- Consider any exploration of G by closed (or open) tour $T = (V_G, E_T)$ from vertex v_i (or v_1) until we reach vertex v_j from which we cannot continue without repeating an edge (no unused incident edge).
  (a) If no odd-degree vertex then $v_i = v_j$;
  (b) otherwise, $v_j = v_2$.

Theorem: Existence of an Eulerian tour

Proof (continued)

- Let $G' = G - T = (V_{G'} = \{u, v | (u, v) \in E_G - E_T\}, E_G - E_T)$. G' can be unconnected, but contains only even-degree vertices.

Theorem: Existence of an Eulerian tour

Proof (continued)

- Let $G' = G - T = (V_{G'} = \{u, v | (u, v) \in E_G - E_T\}, E_G - E_T)$. G' can be unconnected, but contains only even-degree vertices.
- From IH, G' has an Eulerian tour for every its component.

Theorem: Existence of an Eulerian tour

Proof (continued)

- Let $G' = G - T = (V_{G'} = \{u, v | (u, v) \in E_G - E_T\}, E_G - E_T)$. G' can be unconnected, but contains only even-degree vertices.
- From IH, G' has an Eulerian tour for every its component.
- Since G is connected and if G' is nonempty, then $V_T \cap V_{G'} \neq \emptyset$.

Theorem: Existence of an Eulerian tour

Proof (continued)

- Let $G' = G - T = (V_{G'} = \{u, v | (u, v) \in E_G - E_T\}, E_G - E_T)$. G' can be unconnected, but contains only even-degree vertices.
- From IH, G' has an Eulerian tour for every its component.
- Since G is connected and if G' is nonempty, then $V_T \cap V_{G'} \neq \emptyset$.
- Now, we inject Eulerian tours from G' into T using any of these common vertices. □

Example: Draw a house by a tour

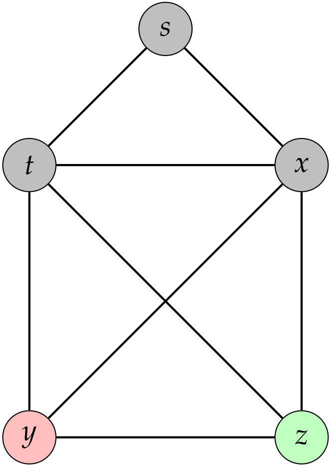


Figure: Eulerian House

Example: Draw a house by a tour

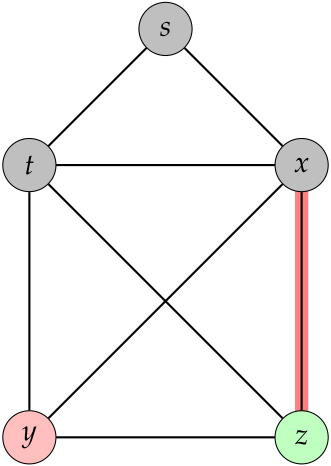


Figure: Eulerian House

Example: Draw a house by a tour

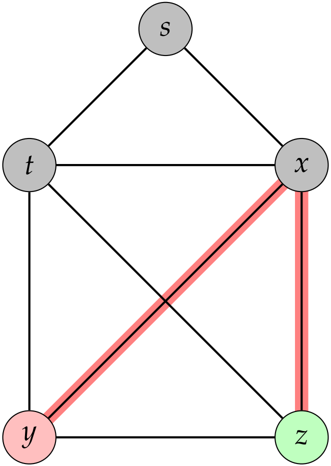


Figure: Eulerian House

Example: Draw a house by a tour

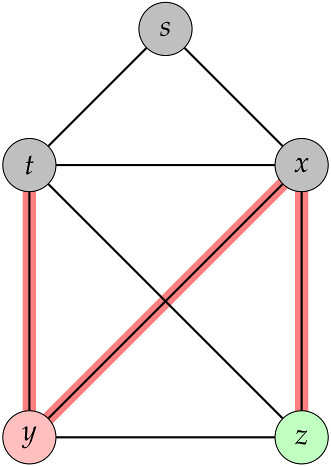


Figure: Eulerian House

Example: Draw a house by a tour

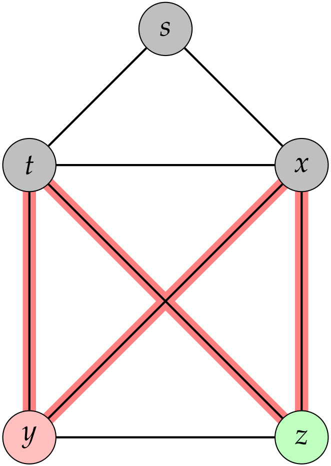


Figure: Eulerian House

Example: Draw a house by a tour

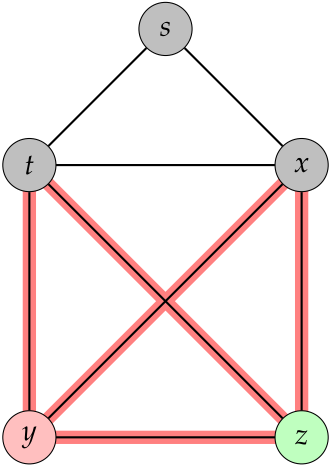


Figure: Eulerian House

Example: Draw a house by a tour

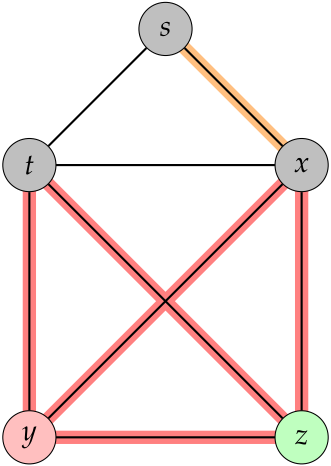


Figure: Eulerian House

Example: Draw a house by a tour

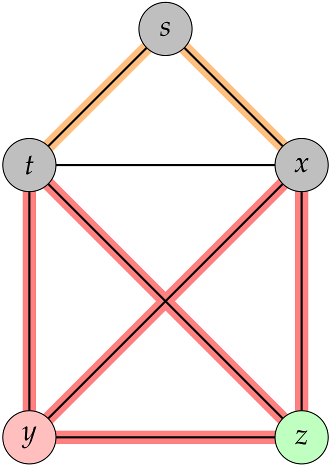


Figure: Eulerian House

Example: Draw a house by a tour

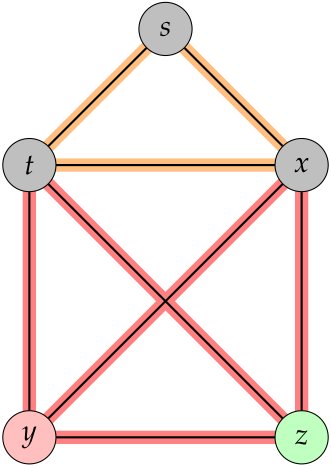


Figure: Eulerian House

Example: Draw a house by a tour

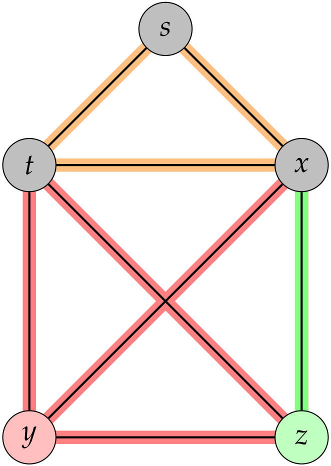


Figure: Eulerian House

Example: Draw a house by a tour

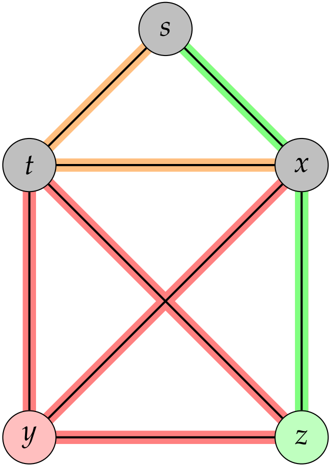


Figure: Eulerian House

Example: Draw a house by a tour

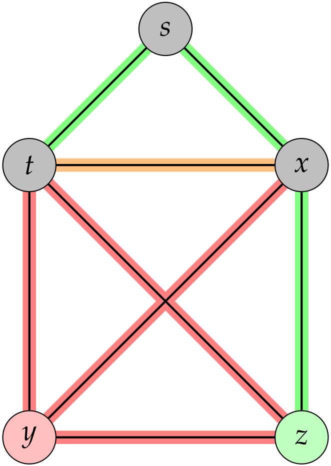


Figure: Eulerian House

Example: Draw a house by a tour

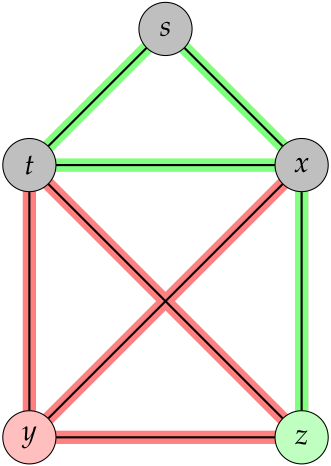


Figure: Eulerian House

Example: Draw a house by a tour

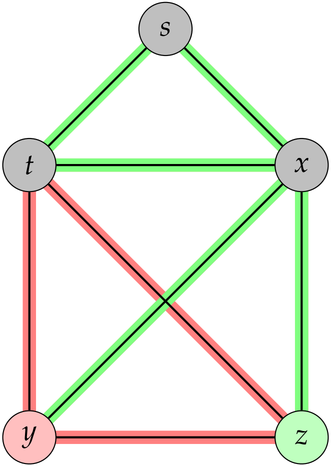


Figure: Eulerian House

Example: Draw a house by a tour

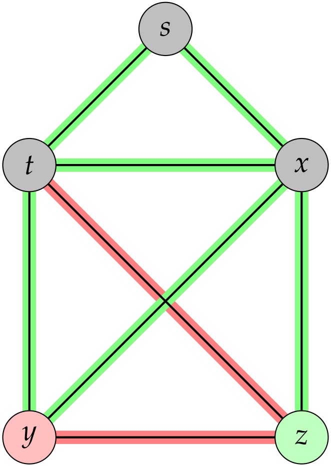


Figure: Eulerian House

Example: Draw a house by a tour

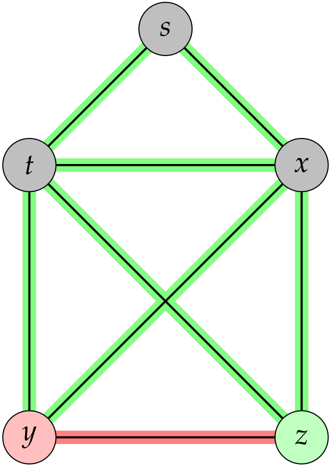


Figure: Eulerian House

Example: Draw a house by a tour

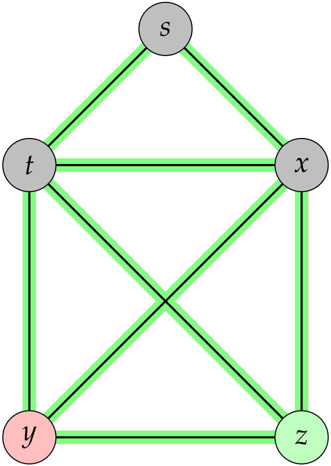


Figure: Eulerian House

Eulerian tour in digraphs

Out-tree of a graph $G = (V, E)$ is a directed subgraph (spanning tree) $T = (V, E')$ with root $u \in V$ where $E' \subseteq E$ and $d_+(u) = 0$ and $d_+(v) = 1$ for every $v \in V - \{u\}$.

Eulerian tour in digraphs

Out-tree of a graph $G = (V, E)$ is a directed subgraph (spanning tree) $T = (V, E')$ with root $u \in V$ where $E' \subseteq E$ and $d_+(u) = 0$ and $d_+(v) = 1$ for every $v \in V - \{u\}$.
Balanced graph $G = (V, E)$ is a digraph with $d_+(u) = d_-(u)$ for every $u \in V$.

Eulerian tour in digraphs

Out-tree of a graph $G = (V, E)$ is a directed subgraph (spanning tree) $T = (V, E')$ with root $u \in V$ where $E' \subseteq E$ and $d_+(u) = 0$ and $d_+(v) = 1$ for every $v \in V - \{u\}$.
Balanced graph $G = (V, E)$ is a digraph with $d_+(u) = d_-(u)$ for every $u \in V$.

Theorem 35.
A digraph $G = (V, E)$ is Eulerian if and only if G is connected (after making symmetric) and balanced. G has an Eulerian path if and only if G is connected and the degrees of V satisfy:

$$d_-(v_1) = d_+(v_1) + 1 \text{ and } d_+(v_2) = d_-(v_2) + 1 \text{ and}$$

$$\text{for every } v \in V - \{v_1, v_2\},\ d_-(v) = d_+(v)$$

Eulerian tour in digraphs

Out-tree of a graph $G = (V, E)$ is a directed subgraph (spanning tree) $T = (V, E')$ with root $u \in V$ where $E' \subseteq E$ and $d_+(u) = 0$ and $d_+(v) = 1$ for every $v \in V - \{u\}$.
Balanced graph $G = (V, E)$ is a digraph with $d_+(u) = d_-(u)$ for every $u \in V$.

Theorem 35.
A digraph $G = (V, E)$ is Eulerian if and only if G is connected (after making symmetric) and balanced. G has an Eulerian path if and only if G is connected and the degrees of V satisfy:

$$d_-(v_1) = d_+(v_1) + 1 \quad \text{and} \quad d_+(v_2) = d_-(v_2) + 1 \quad \text{and}$$

$$\text{for every } v \in V - \{v_1, v_2\},\ d_-(v) = d_+(v)$$

Proof. The first part in analogy to undirected Eulerian graph.

Directed Eulerian Tour – Examples

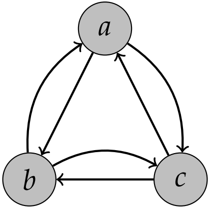


Figure: Eulerian digraph

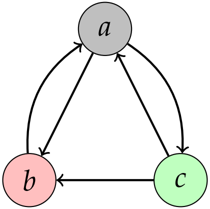


Figure: Eulerian path that is not a circuit

Theorem: Spanning out-tree of Eulerian digraph

Theorem 36.
Let $G = (V, E)$ be an Eulerian digraph and T its subgraph created by Eulerian tour from any vertex u in the following way: for every $v \neq u$, we add the first edge leading to v. Then, T is a spanning out-tree of digraph G rooted at u.

Proof

- From the construction of T, it holds that $d_+(u) = 0$ and $d_+(v) = 1$ for every $u \neq v$, $u, v \in V$.

Theorem: Spanning out-tree of Eulerian digraph

Theorem 36.
Let $G = (V, E)$ be an Eulerian digraph and T its subgraph created by Eulerian tour from any vertex u in the following way: for every $v \neq u$, we add the first edge leading to v. Then, T is a spanning out-tree of digraph G rooted at u.

Proof

- From the construction of T, it holds that $d_+(u) = 0$ and $d_+(v) = 1$ for every $u \neq v$, $u, v \in V$.
- Observe that T has $n - 1$ edges. Now, we prove that T is acyclic (by contradiction):

Theorem: Spanning out-tree of Eulerian digraph

Theorem 36.
Let $G = (V, E)$ be an Eulerian digraph and T its subgraph created by Eulerian tour from any vertex u in the following way: for every $v \neq u$, we add the first edge leading to v. Then, T is a spanning out-tree of digraph G rooted at u.

Proof

- From the construction of T, it holds that $d_+(u) = 0$ and $d_+(v) = 1$ for every $u \neq v$, $u, v \in V$.
- Observe that T has $n - 1$ edges. Now, we prove that T is acyclic (by contradiction):
- Assume that T contains a cycle finished by (v_i, v_j).

Theorem: Spanning out-tree of Eulerian digraph

Theorem 36.
Let $G = (V, E)$ be an Eulerian digraph and T its subgraph created by Eulerian tour from any vertex u in the following way: for every $v \neq u$, we add the first edge leading to v. Then, T is a spanning out-tree of digraph G rooted at u.

Proof

- From the construction of T, it holds that $d_+(u) = 0$ and $d_+(v) = 1$ for every $u \neq v$, $u, v \in V$.
- Observe that T has $n - 1$ edges. Now, we prove that T is acyclic (by contradiction):
- Assume that T contains a cycle finished by (v_i, v_j).
- Since $d_+(u) = 0$, $v_j \neq u$.

Theorem: Spanning out-tree of Eulerian digraph

Theorem 36.
Let $G = (V, E)$ be an Eulerian digraph and T its subgraph created by Eulerian tour from any vertex u in the following way: for every $v \neq u$, we add the first edge leading to v. Then, T is a spanning out-tree of digraph G rooted at u.

Proof

- From the construction of T, it holds that $d_+(u) = 0$ and $d_+(v) = 1$ for every $u \neq v$, $u, v \in V$.
- Observe that T has $n - 1$ edges. Now, we prove that T is acyclic (by contradiction):
- Assume that T contains a cycle finished by (v_i, v_j).
- Since $d_+(u) = 0$, $v_j \neq u$.
- Since (v_i, v_j) closes a cycle, so v_j was already processed, which is a contradiction! □

Theorem about directed Eulerian tour

Theorem 37.
If G is connected and balanced digraph with a directed spanning tree T rooted at u, then we can find Eulerian circuit in the reverse order in the following way:

(a) *Start with any edge incident to u.*
(b) *Next edges are chosen as incident to the current vertex such that:*
    (i) *the edge was not visited yet,*
    (ii) *the edges from T are chosen as the last ones.*
(c) *The search ends if the current vertex has no incident unvisited edges.*

Proof

- The balanced property guarantees that it ends back in root u.

Theorem about directed Eulerian tour

Theorem 37.
If G is connected and balanced digraph with a directed spanning tree T rooted at u, then we can find Eulerian circuit in the reverse order in the following way:

(a) *Start with any edge incident to u.*
(b) *Next edges are chosen as incident to the current vertex such that:*
  (i) *the edge was not visited yet,*
  (ii) *the edges from T are chosen as the last ones.*
(c) *The search ends if the current vertex has no incident unvisited edges.*

Proof

- The balanced property guarantees that it ends back in root u.
- Assume that the circuit does not contain an edge (v_i, v_j).

Theorem about directed Eulerian tour

Proof

- Assume that the circuit does not contain an edge (v_i, v_j).

Theorem about directed Eulerian tour

Proof

- Assume that the circuit does not contain an edge (v_i, v_j).
- Since the balanced graph, v_i must be the end vertex for the next unvisited edge (v_k, v_i).

Theorem about directed Eulerian tour

Proof

- Assume that the circuit does not contain an edge (v_i, v_j).
- Since the balanced graph, v_i must be the end vertex for the next unvisited edge (v_k, v_i).
- Let edge (v_k, v_i) be from T, so it will not be used because of step (b(ii)).

Theorem about directed Eulerian tour

Proof

- Assume that the circuit does not contain an edge (v_i, v_j).
- Since the balanced graph, v_i must be the end vertex for the next unvisited edge (v_k, v_i).
- Let edge (v_k, v_i) be from T, so it will not be used because of step (b(ii)).
- Now, traverse the sequence of edges reversely back to u.

Theorem about directed Eulerian tour

Proof

- Assume that the circuit does not contain an edge (v_i, v_j).
- Since the balanced graph, v_i must be the end vertex for the next unvisited edge (v_k, v_i).
- Let edge (v_k, v_i) be from T, so it will not be used because of step (b(ii)).
- Now, traverse the sequence of edges reversely back to u.
- Since G is balanced, we find unvisited edge that is incident to u, which is a contradiction with step (c). □

Algorithm for searching directed Eulerian path

```
EULER-CIRCUIT(G)
Find an oriented spanning out-tree T = (V, E_T) of G = (V, E) (root u)
for every vertex v ∈ V
    do  A[v] ← ∅
        I[v] ← 0
for every edge (v_i, v_j) ∈ E
    do if (v_i, v_j) ∈ E_T
          then add v_i to the tail of list A[v_j]
          else add v_i to the head of list A[v_j]
EC ← ∅
CV ← u
while I[CV] ≤ d_+(CV)
    do  add CV to the head of list EC
        I[CV] ← I[CV] + 1
        CV ← A[CV][I[CV]]
Print EC
```

Algorithm for searching directed Eulerian path

Analysis of time complexity

- Eulerian graph has always $m \geq n$ (more edges then vertices).

Algorithm for searching directed Eulerian path

Analysis of time complexity

- Eulerian graph has always $m \geq n$ (more edges then vertices).
- Line 1: DFS, get the highest f and then DFS from vertex with the highest $f \Rightarrow O(m)$.

Algorithm for searching directed Eulerian path

Analysis of time complexity

- Eulerian graph has always $m \geq n$ (more edges then vertices).
- Line 1: DFS, get the highest f and then DFS from vertex with the highest $f \Rightarrow O(m)$.
- In *while* cycle, we always increment $I[CV]$, so $\sum_{v \in V} d_+(v) = \Theta(m)$.

Algorithm for searching directed Eulerian path

Analysis of time complexity

- Eulerian graph has always $m \geq n$ (more edges then vertices).
- Line 1: DFS, get the highest f and then DFS from vertex with the highest $f \Rightarrow O(m)$.
- In *while* cycle, we always increment $I[CV]$, so $\sum_{v \in V} d_+(v) = \Theta(m)$.
- Therefore, the total time complexity $O(m)$.

Application of Eulerian tours

- de Bruijn sequence

Application of Eulerian tours

- de Bruijn sequence
  - Given an alphabet, find cycle-string where are no two same substrings of length k.

Application of Eulerian tours

- de Bruijn sequence
  - Given an alphabet, find cycle-string where are no two same substrings of length k.
- Chinese postman problem: traverse all the streets of the district effectively and get back to post office.

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.
- Find the shortest (non-simple) circuit that contains all edges of G.

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.
- Find the shortest (non-simple) circuit that contains all edges of G.
  - Given an alphabet, find cycle-string where are no two same substrings of length k.

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.
- Find the shortest (non-simple) circuit that contains all edges of G.
  - Given an alphabet, find cycle-string where are no two same substrings of length k.
- Chinese postman problem: traverse all the streets of the district effectively and get back to post office.

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.
- Find the shortest (non-simple) circuit that contains all edges of G.
  - Given an alphabet, find cycle-string where are no two same substrings of length k.
- Chinese postman problem: traverse all the streets of the district effectively and get back to post office.
  - Given connected positively-weighted digraph,

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.
- Find the shortest (non-simple) circuit that contains all edges of G.
  - Given an alphabet, find cycle-string where are no two same substrings of length k.
- Chinese postman problem: traverse all the streets of the district effectively and get back to post office.
  - Given connected positively-weighted digraph,
  - find the shortest circuit that contains all edges of such digraph.

Chinese postman problem for undirected graphs

- Let $G = (V, E)$ be a connected positively-weighted non-Eulerian undirected graph.
- Find the shortest (non-simple) circuit that contains all edges of G.
  - Given an alphabet, find cycle-string where are no two same substrings of length k.
- Chinese postman problem: traverse all the streets of the district effectively and get back to post office.
  - Given connected positively-weighted digraph,
  - find the shortest circuit that contains all edges of such digraph.
  - Optimal solution for non-Eulerian graph: $O(m + n^3)$

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'
3. Find a minimum-weight perfect matching of G'

Algorithm for Chinese postman problem

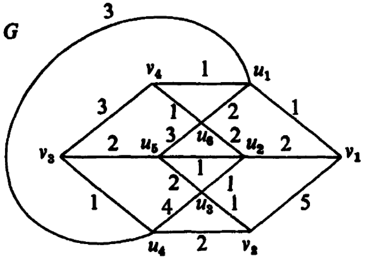
1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'
3. Find a minimum-weight perfect matching of G'
4. Construct G''

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'
3. Find a minimum-weight perfect matching of G'
4. Construct G''
5. Find an Eulerian circuit of G'' and thus a minimum-weight postman's circuit of G.

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'
3. Find a minimum-weight perfect matching of G'
4. Construct G''
5. Find an Eulerian circuit of G'' and thus a minimum-weight postman's circuit of G.

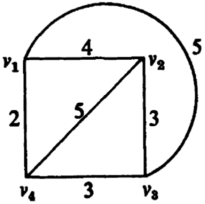


$d(v_1, v_2) = 4$ along (v_1, u_2, u_3, v_2)
$d(v_1, v_3) = 5$ along (v_1, u_2, u_5, v_3)
$d(v_1, v_4) = 2$ along (v_1, u_1, v_4)
$d(v_2, v_3) = 3$ along (v_2, u_4, v_3)
$d(v_2, v_4) = 5$ along $(v_2, u_3, u_2, u_6, v_4)$
$d(v_3, v_4) = 3$ along (v_3, v_4)

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'
3. Find a minimum-weight perfect matching of G'
4. Construct G''
5. Find an Eulerian circuit of G'' and thus a minimum-weight postman's circuit of G.

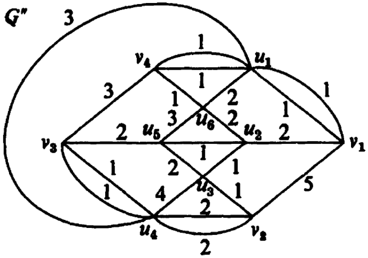
G'.



A minimum-weight perfect matching consists of the edges (v_1, v_4) and (v_2, v_3).

Algorithm for Chinese postman problem

1. Find the set of shortest paths between all pairs of vertices of odd-degree in G.
2. Construct G'
3. Find a minimum-weight perfect matching of G'
4. Construct G''
5. Find an Eulerian circuit of G'' and thus a minimum-weight postman's circuit of G.



An Eulerian circuit of G'' and a solution to the Chinese postman problem for G is (v_1, u_1, v_4, v_3, u_4, v_2, v_1, u_2, u_3, v_2, u_4, u_3, u_5, v_3, u_4, u_1, v_4, u_6, u_5, u_2, u_6, u_1, v_1).

Hamiltonian Paths and Cycles

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.
- Types of Hamiltonian tasks/problems

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.
- Types of Hamiltonian tasks/problems
  - Existence problems - does a Hamiltonian tour exist (solution: yes/no; or the path itself)

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.
- Types of Hamiltonian tasks/problems
  - Existence problems - does a Hamiltonian tour exist (solution: yes/no; or the path itself)
  - Optimization problems - find the best Hamiltonian tour in a weighted graph

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.
- Types of Hamiltonian tasks/problems
  - Existence problems - does a Hamiltonian tour exist (solution: yes/no; or the path itself)
  - Optimization problems - find the best Hamiltonian tour in a weighted graph
- All tasks here are NP-Complete (very hard).

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.
- Types of Hamiltonian tasks/problems
  - Existence problems - does a Hamiltonian tour exist (solution: yes/no; or the path itself)
  - Optimization problems - find the best Hamiltonian tour in a weighted graph
- All tasks here are NP-Complete (very hard).
- Necessary condition = Each Hamiltonian graph satisfies but some non-Hamiltonian as well.

Hamiltonian path and cycles

- Hamiltonian graph is a graph that contains Hamiltonian circuit. That is, closed path going through all vertices exactly once.
- Types of Hamiltonian tasks/problems
  - Existence problems - does a Hamiltonian tour exist (solution: yes/no; or the path itself)
  - Optimization problems - find the best Hamiltonian tour in a weighted graph
- All tasks here are NP-Complete (very hard).
- Necessary condition = Each Hamiltonian graph satisfies but some non-Hamiltonian as well.
- Sufficient condition = Only Hamiltonian graphs satisfies but not all of them.

Sufficient conditions for special graphs

Theorem 38.
Every complete graph is Hamiltonian.

Sufficient conditions for special graphs

Theorem 38.
Every complete graph is Hamiltonian.

Proof

- Take any permutation of vertices.

Sufficient conditions for special graphs

Theorem 38.
Every complete graph is Hamiltonian.

Proof

- Take any permutation of vertices.

Theorem 39.
Every digraph with complete symmetric graph contains a Hamiltonian path.

Sufficient conditions for special graphs

Theorem 38.
Every complete graph is Hamiltonian.

Proof

- Take any permutation of vertices.

Theorem 39.
Every digraph with complete symmetric graph contains a Hamiltonian path.

Theorem 40.
Every strongly-connected digraph with complete symmetric graph is Hamiltonian graph.

Sufficient conditions for special graphs

Theorem 38.
Every complete graph is Hamiltonian.

Proof

- Take any permutation of vertices.

Theorem 39.
Every digraph with complete symmetric graph contains a Hamiltonian path.

Theorem 40.
Every strongly-connected digraph with complete symmetric graph is Hamiltonian graph.

Theorem 41.
If $G = (V, E)$ is a graph such that $|V| > 3$ and $\min_{v \in V}(d(v)) > \frac{n}{2}$ then G is Hamiltonian.

Chvátal theorem (1972)

Theorem 42.
Let G be undirected graph with $n \geq 3$ vertices. If $d(v_1) \leq d(v_2) \leq \cdots \leq d(v_n)$ is a non-descending sequence of degrees of vertices and, in addition, the following holds:

$$\text{if for some } k \leq \frac{n}{2} \text{ is } d(v_k) \leq k \text{ then } d(v_{n-k}) \geq n - k$$

then G is Hamiltonian.

- First part of the proof guarantees the existence of a Hamiltonian circuit for sufficiently high degrees.

Chvátal theorem (1972)

Theorem 42.
Let G be undirected graph with $n \geq 3$ vertices. If $d(v_1) \leq d(v_2) \leq \cdots \leq d(v_n)$ is a non-descending sequence of degrees of vertices and, in addition, the following holds:

$$\text{if for some } k \leq \frac{n}{2} \text{ is } d(v_k) \leq k \text{ then } d(v_{n-k}) \geq n-k$$

then G is Hamiltonian.

- First part of the proof guarantees the existence of a Hamiltonian circuit for sufficiently high degrees.
- Second part proves that this is the best sufficient condition based on the degrees of vertices.

Chvátal theorem (1972)

Theorem 42.
Let G be undirected graph with $n \geq 3$ vertices. If $d(v_1) \leq d(v_2) \leq \cdots \leq d(v_n)$ is a non-descending sequence of degrees of vertices and, in addition, the following holds:

$$\text{if for some } k \leq \frac{n}{2} \text{ is } d(v_k) \leq k \text{ then } d(v_{n-k}) \geq n - k$$

then G is Hamiltonian.

- First part of the proof guarantees the existence of a Hamiltonian circuit for sufficiently high degrees.
- Second part proves that this is the best sufficient condition based on the degrees of vertices.
- The proof by contradiction is very complex and non-constructive.

Necessary conditions for special graphs

Biconnected graph is a graph where any one vertex can be removed and graph remains connected. That is, there is no articulation vertex.

Necessary conditions for special graphs

Biconnected graph is a graph where any one vertex can be removed and graph remains connected. That is, there is no articulation vertex.

Theorem 43.
All Hamiltonian graphs are biconnected.

Necessary conditions for special graphs

Biconnected graph is a graph where any one vertex can be removed and graph remains connected. That is, there is no articulation vertex.

Theorem 43.
All Hamiltonian graphs are biconnected.

- But a biconnected graph need not be Hamiltonian.

Necessary conditions for special graphs

Biconnected graph is a graph where any one vertex can be removed and graph remains connected. That is, there is no articulation vertex.

Theorem 43.
All Hamiltonian graphs are biconnected.

- But a biconnected graph need not be Hamiltonian.
- See, for example, the Petersen graph

Necessary conditions for special graphs

Biconnected graph is a graph where any one vertex can be removed and graph remains connected. That is, there is no articulation vertex.

Theorem 43.
All Hamiltonian graphs are biconnected.

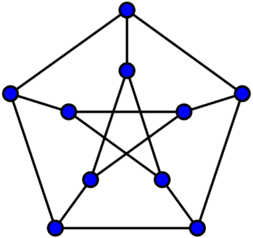
- But a biconnected graph need not be Hamiltonian.
- See, for example, the Petersen graph

Necessary conditions for special graphs

Biconnected graph is a graph where any one vertex can be removed and graph remains connected. That is, there is no articulation vertex.

Theorem 43.
All Hamiltonian graphs are biconnected.

- But a biconnected graph need not be Hamiltonian.
- See, for example, the Petersen graph



Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.

Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.
- Corresponds to Hamiltonian graphs: Find the shortest Hamiltonian circuit in weighted complete graph.

Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.
- Corresponds to Hamiltonian graphs: Find the shortest Hamiltonian circuit in weighted complete graph.
- Technique: Optimization task $\rightarrow$ problem over complete graph:

Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.
- Corresponds to Hamiltonian graphs: Find the shortest Hamiltonian circuit in weighted complete graph.
- Technique: Optimization task $\rightarrow$ problem over complete graph:
  - Add edges to the general graph G to get complete graph K, weight the edges by M.

Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.
- Corresponds to Hamiltonian graphs: Find the shortest Hamiltonian circuit in weighted complete graph.
- Technique: Optimization task $\rightarrow$ problem over complete graph:
  - Add edges to the general graph G to get complete graph K, weight the edges by M.
  - M is big enough (e.g. the sum of all original weights).

Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.
- Corresponds to Hamiltonian graphs: Find the shortest Hamiltonian circuit in weighted complete graph.
- Technique: Optimization task $\rightarrow$ problem over complete graph:
  - Add edges to the general graph G to get complete graph K, weight the edges by M.
  - M is big enough (e.g. the sum of all original weights).
  - Solve the problem in K. If the result contains edge with M, there is no solution in G.

Travel Salesman Problem

- Salesman want to visit n cities without repetition and with the shortest circuit return to the starting city.
- Corresponds to Hamiltonian graphs: Find the shortest Hamiltonian circuit in weighted complete graph.
- Technique: Optimization task $\rightarrow$ problem over complete graph:
  - Add edges to the general graph G to get complete graph K, weight the edges by M.
  - M is big enough (e.g. the sum of all original weights).
  - Solve the problem in K. If the result contains edge with M, there is no solution in G.
- Applications: Transportation tasks, Process scheduling, ...

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$
    1. Find minimum spanning tree T in G;

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$
    1. Find minimum spanning tree T in G;
    2. If $w(T) \geq$ bound then skip this branch;

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$
  1. Find minimum spanning tree T in G;
  2. If $w(T) \geq$ bound then skip this branch;
  3. Is T Hamiltonian path? Yes, bound $\leftarrow w(T)$;

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$
  1. Find minimum spanning tree T in G;
  2. If $w(T) \geq$ bound then skip this branch;
  3. Is T Hamiltonian path? Yes, bound $\leftarrow w(T)$;
  4. Take some vertex v with $d(v) = k \geq 3$.

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$
  1. Find minimum spanning tree T in G;
  2. If $w(T) \geq$ bound then skip this branch;
  3. Is T Hamiltonian path? Yes, bound $\leftarrow w(T)$;
  4. Take some vertex v with $d(v) = k \geq 3$.
  5. Remove some edge e incident with v and execute the search recursively in $G - e$ (k new branches).

Finding minimum-length Hamiltonian path

- Observe: Every Hamiltonian path is a spanning tree of G (vertices with degree ≤ 2)
- *Branch and Bound* technique: Let bound $\leftarrow \infty$
  1. Find minimum spanning tree T in G;
  2. If $w(T) \geq$ bound then skip this branch;
  3. Is T Hamiltonian path? Yes, bound $\leftarrow w(T)$;
  4. Take some vertex v with $d(v) = k \geq 3$.
  5. Remove some edge e incident with v and execute the search recursively in $G - e$ (k new branches).
- Intractable/ineffective since enumeration grows with $n!$.