Department of
Computer Science and Engineering
计算机科学与工程系

CS215 DISCRETE MATH

Dr. Qi Wang

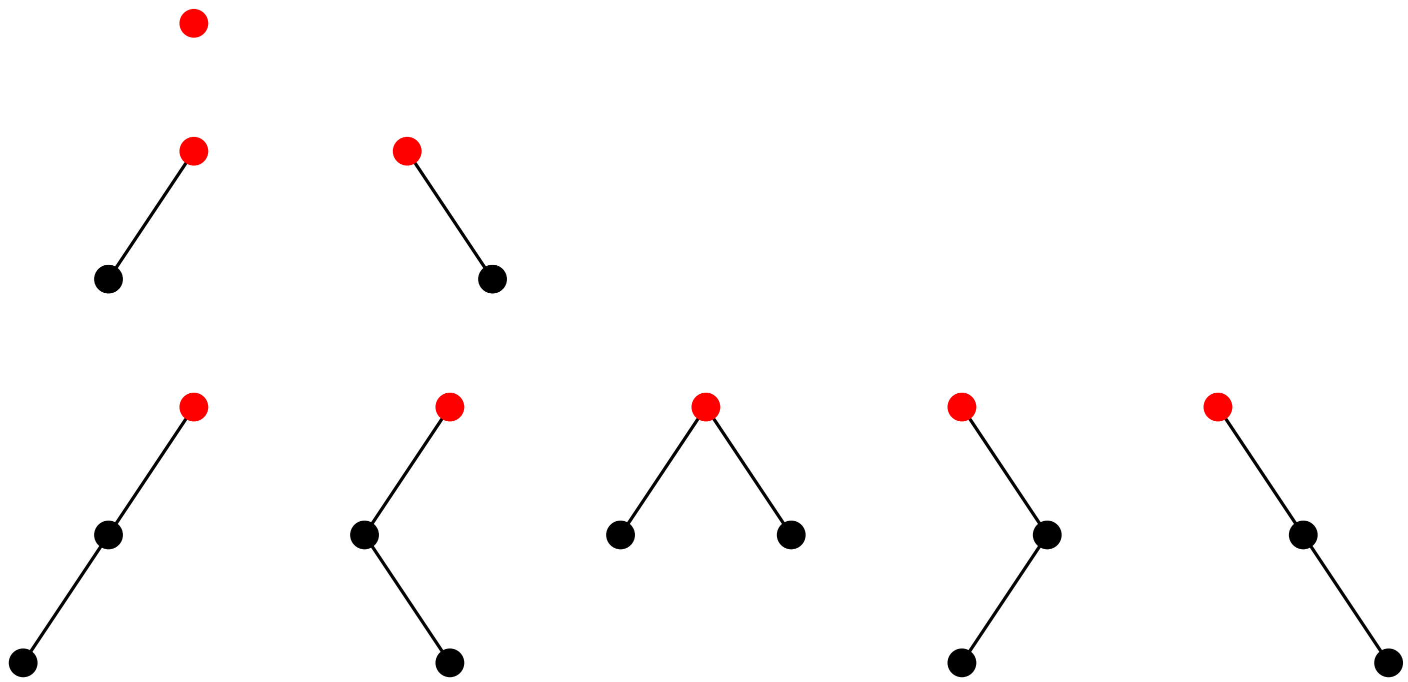
Department of Computer Science and Engineering
Office: Room413, CoE South Tower
Email: wangqi@sustech.edu.cn

Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

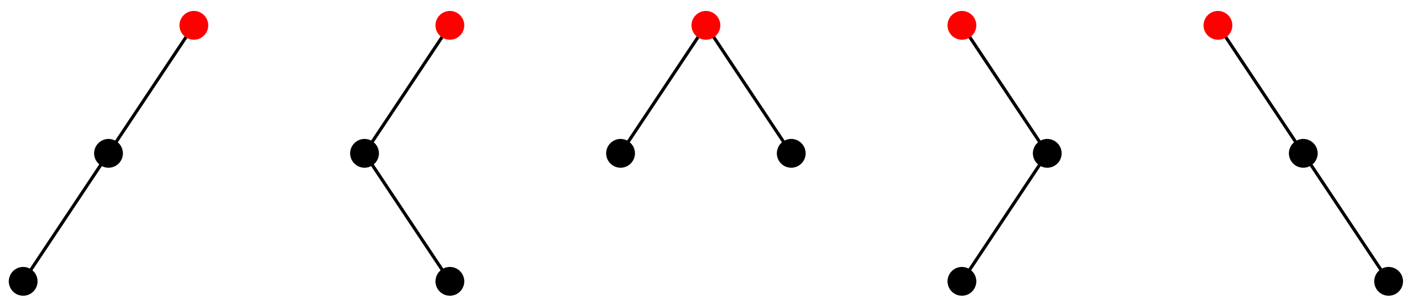


Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

$$C_0 = C_1 = 1$$

$$C_2 = 2$$



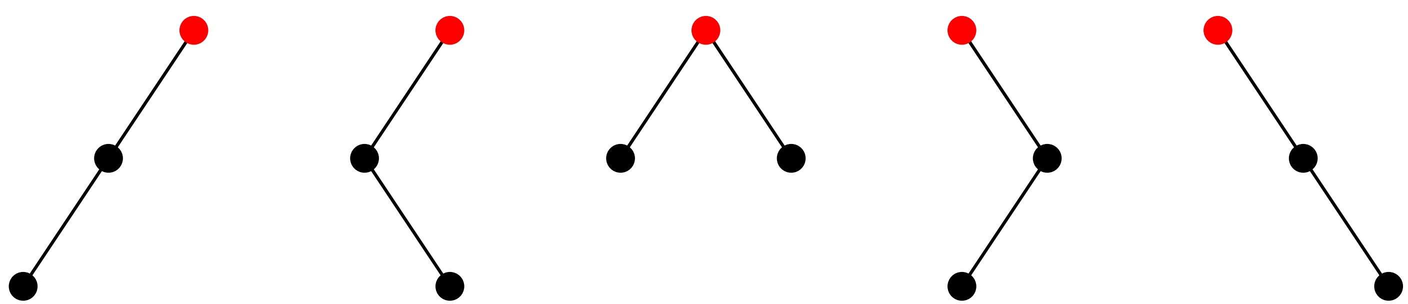
$$C_3 = 5$$

Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

$$C_0 = C_1 = 1$$

$$C_2 = 2$$



$$C_3 = 5$$

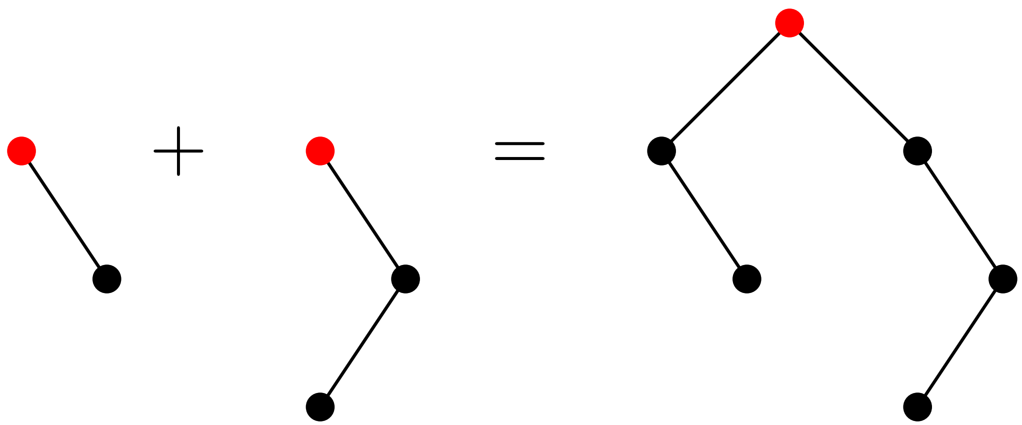
How to find a formula for C_n?

Catalan Numbers

- We first give an important *observation* on the recursive relation.

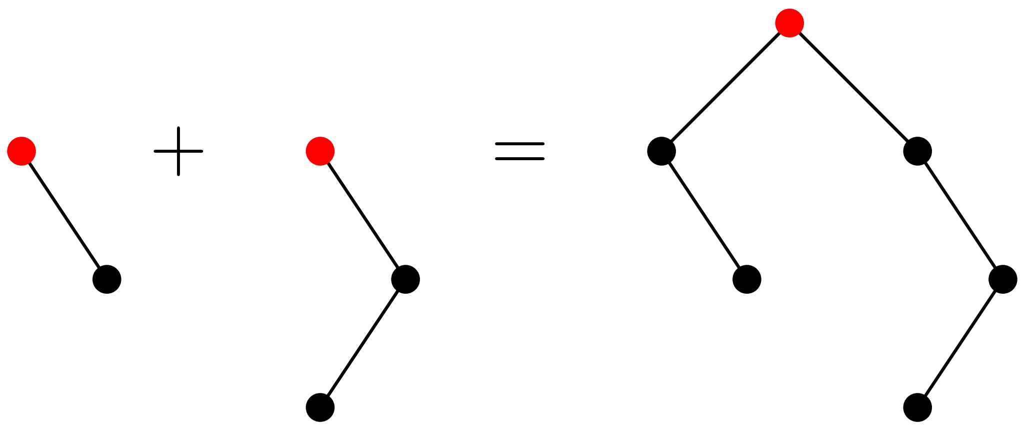
Catalan Numbers

- We first give an important *observation* on the recursive relation.
  Any nonempty rooted binary tree
  $=$ two smaller binary trees (possibly empty) $+$ one extra root



Catalan Numbers

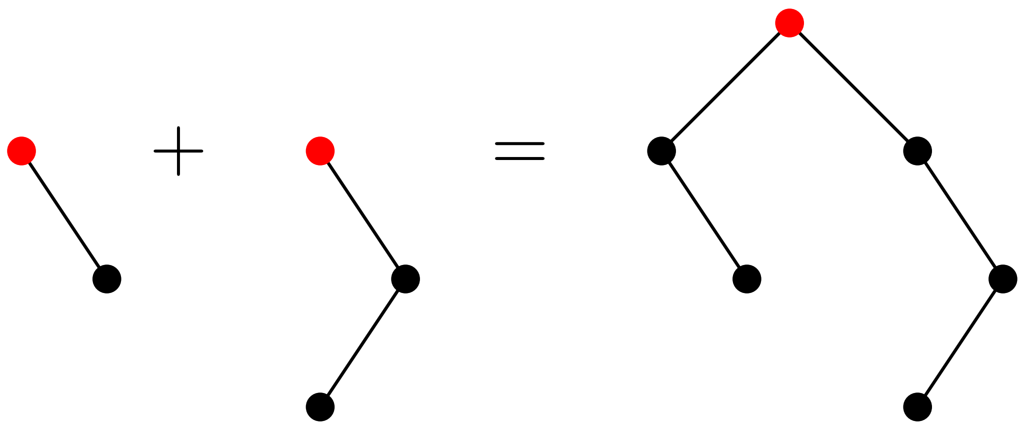
- We first give an important *observation* on the recursive relation.
  Any nonempty rooted binary tree
  $=$ two smaller binary trees (possibly empty) $+$ one extra root



We have $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$

Catalan Numbers

- We first give an important *observation* on the recursive relation.
  Any nonempty rooted binary tree
  $=$ two smaller binary trees (possibly empty) $+$ one extra root



We have $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$

For example, $C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0 = 1 * 2 + 1 * 1 + 2 * 1 = 5.$

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

  Then we have $xf^2 + 1 = f$, which gives $f = \frac{1 \pm \sqrt{1-4x}}{2x}$ for $x \neq 0$.

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

  Then we have $xf^2 + 1 = f$, which gives $f = \frac{1 \pm \sqrt{1-4x}}{2x}$ for $x \neq 0$.

  When $x \to 0$, $\frac{1+\sqrt{1-4x}}{2x} \to \infty$ and $\frac{1-\sqrt{1-4x}}{2x} \to 1$.

  Since $f(0) = 1 = C_0$, we know $f = \frac{1-\sqrt{1-4x}}{2x}$.

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

  Then we have $xf^2 + 1 = f$, which gives $f = \frac{1 \pm \sqrt{1-4x}}{2x}$ for $x \neq 0$.

  When $x \to 0$, $\frac{1+\sqrt{1-4x}}{2x} \to \infty$ and $\frac{1-\sqrt{1-4x}}{2x} \to 1$.

  Since $f(0) = 1 = C_0$, we know $f = \frac{1-\sqrt{1-4x}}{2x}$.

  C_n – the coefficient of x^n in the expansion of f.

Catalan Numbers

- $f = \frac{1-\sqrt{1-4x}}{2x}$, by the extended Binomial Theorem,

$$\sqrt{1-4x} = (1+(-4x))^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n.$$

Catalan Numbers

- $f = \frac{1-\sqrt{1-4x}}{2x}$, by the extended Binomial Theorem,

$$\sqrt{1-4x} = (1+(-4x))^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n.$$

$$\frac{1-\sqrt{1-4x}}{2x} = \sum_{n=1}^{\infty} -\frac{1}{2}\binom{1/2}{n}(-4)^n x^{n-1} = \sum_{n=0}^{\infty} -\frac{1}{2}\binom{1/2}{n+1}(-4)^{n+1} x^n.$$

where $\binom{1/2}{n} = \frac{\frac{1}{2}(-\frac{1}{2})(-\frac{3}{2})\cdots(-\frac{2n-3}{2})}{n!}$.

Catalan Numbers

- $f = \frac{1-\sqrt{1-4x}}{2x}$, by the extended Binomial Theorem,

$$\sqrt{1-4x} = (1+(-4x))^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n.$$

$$\frac{1-\sqrt{1-4x}}{2x} = \sum_{n=1}^{\infty} -\frac{1}{2}\binom{1/2}{n}(-4)^n x^{n-1} = \sum_{n=0}^{\infty} -\frac{1}{2}\binom{1/2}{n+1}(-4)^{n+1}x^n.$$

where $\binom{1/2}{n} = \frac{\frac{1}{2}(-\frac{1}{2})(-\frac{3}{2})\cdots(-\frac{2n-3}{2})}{n!}$.

Then we have $C_n = \frac{1}{n+1}\binom{2n}{n}$.

This is called the n-th *Catalan number*.

Catalan Numbers: Related Problems

- **Theorem** The number of sequences $a_1, \ldots, a_{2n}$ of $2n$ terms that can be formed using exactly n $+1$'s and exactly n -1's whose partial sums are always nonnegative, i.e., $a_1 + a_2 + \cdots + a_k \geq 0$ for any $1 \leq k \leq 2n$, equals the n-th Catalan number C_n.

Catalan Numbers: Related Problems

- **Theorem** The number of sequences $a_1, \ldots, a_{2n}$ of $2n$ terms that can be formed using exactly n $+1$'s and exactly n -1's whose partial sums are always nonnegative, i.e., $a_1 + a_2 + \cdots + a_k \geq 0$ for any $1 \leq k \leq 2n$, equals the n-th Catalan number C_n.

- Many others! For example, the number of *full binary trees* with $2n+1$ vertices...

Catalan Numbers: Related Problems

- **Theorem** The number of sequences $a_1, \ldots, a_{2n}$ of $2n$ terms that can be formed using exactly n $+1$'s and exactly n -1's whose partial sums are always nonnegative, i.e., $a_1 + a_2 + \cdots + a_k \geq 0$ for any $1 \leq k \leq 2n$, equals the n-th Catalan number C_n.

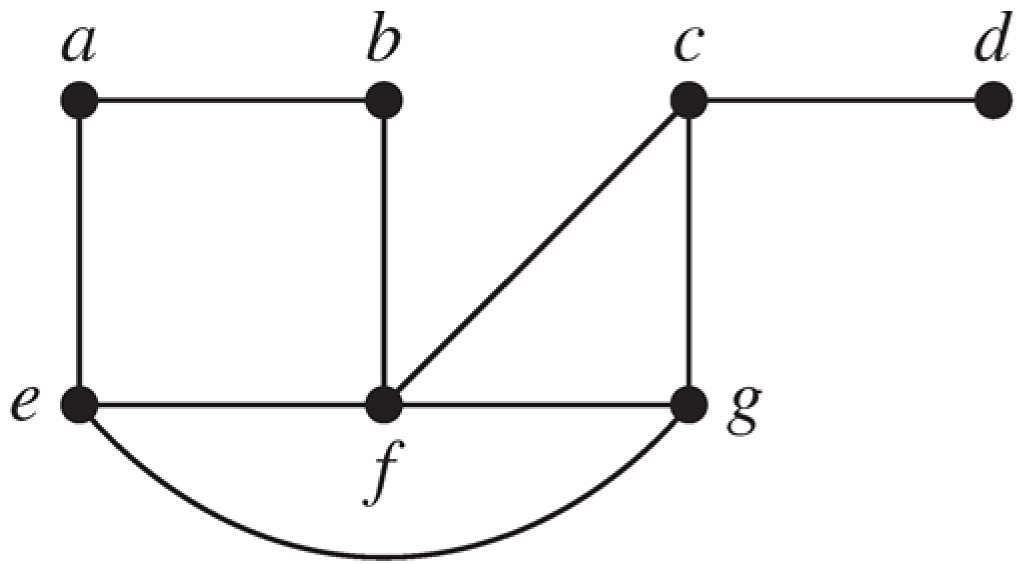
- Many others! For example, the number of *full binary trees* with $2n+1$ vertices...

R. Stanley, *Catalan Numbers*, Cambridge University Press, 2015. Includes 214 combinatorial interpretations of C_n, and 68 additional problems!

Spanning Trees

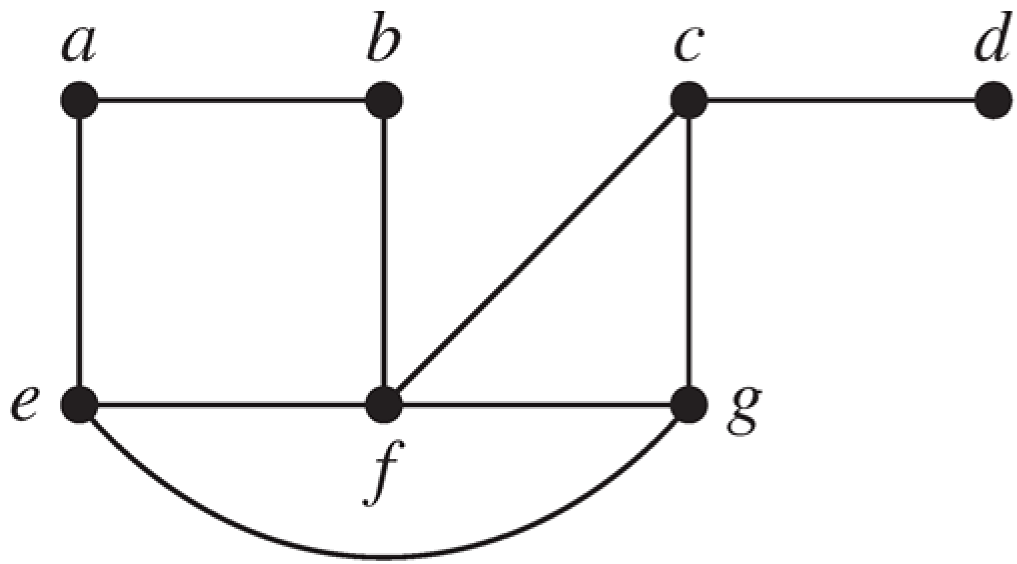
- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G.

- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G.



Spanning Trees

- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G.



remove edges to avoid circuits

Spanning Trees

- **Theorem** A simple graph is connected if and only if it has a spanning tree.

Spanning Trees

- **Theorem** A simple graph is connected if and only if it has a spanning tree.

Proof

- **Theorem** A simple graph is connected if and only if it has a spanning tree.

Proof

"only if " part

- **Theorem** A simple graph is connected if and only if it has a spanning tree.

  Proof

  "only if " part

  The spanning tree can be obtained by removing edges from simple circuits.

- **Theorem** A simple graph is connected if and only if it has a spanning tree.

  Proof

  "only if " part

  The spanning tree can be obtained by removing edges from simple circuits.

  "if " part

- **Theorem** A simple graph is connected if and only if it has a spanning tree.

Proof

"only if" part

The spanning tree can be obtained by removing edges from simple circuits.

"if" part

easy

- We can find spanning trees by removing edges from simple circuits.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.
  - ◇ First arbitrarily choose a vertex of the graph as the root.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ Form a path by successively adding vertices and edges.
  Continue adding to this path as long as possible.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ Form a path by successively adding vertices and edges. Continue adding to this path as long as possible.

  ◇ If the path goes through all vertices of the graph, the tree is a spanning tree.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

  ◇ First arbitrarily choose a vertex of the graph as the root.

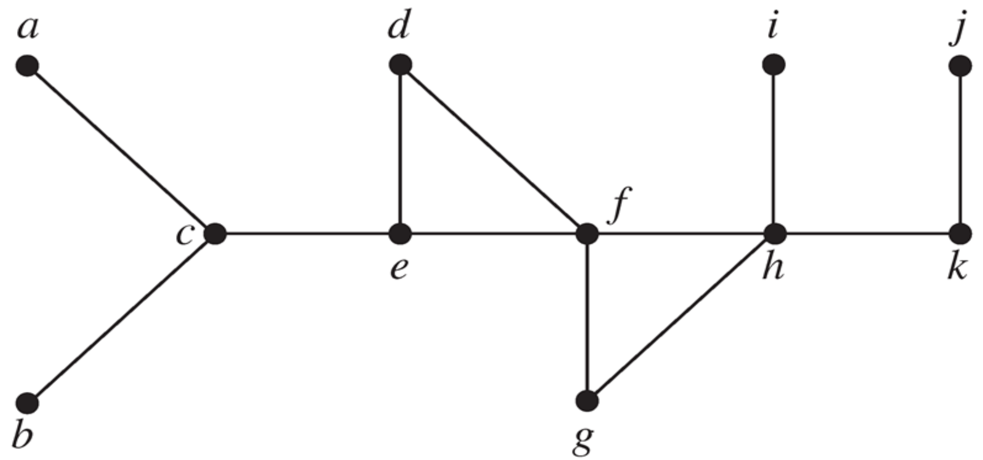
  ◇ Form a path by successively adding vertices and edges.
  Continue adding to this path as long as possible.

  ◇ If the path goes through all vertices of the graph, the tree is a spanning tree.

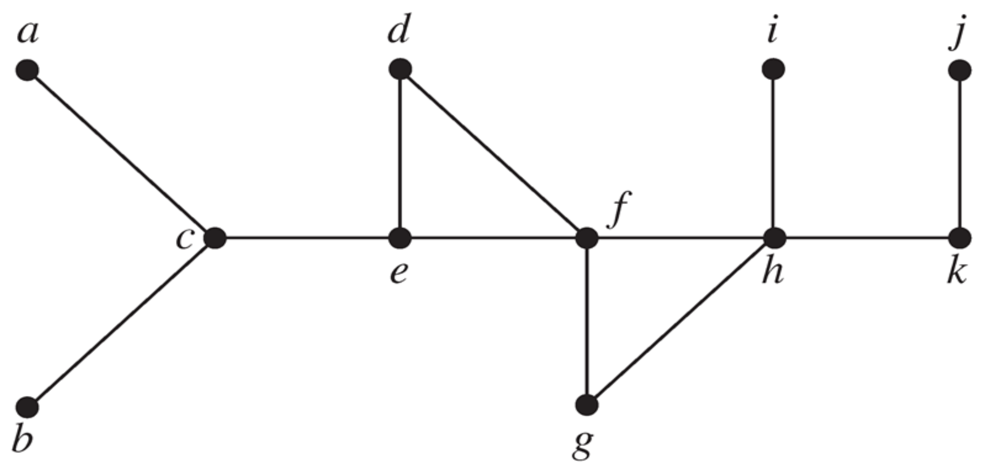
  ◇ Otherwise, move back to some vertex to repeat this procedure (*backtracking*)

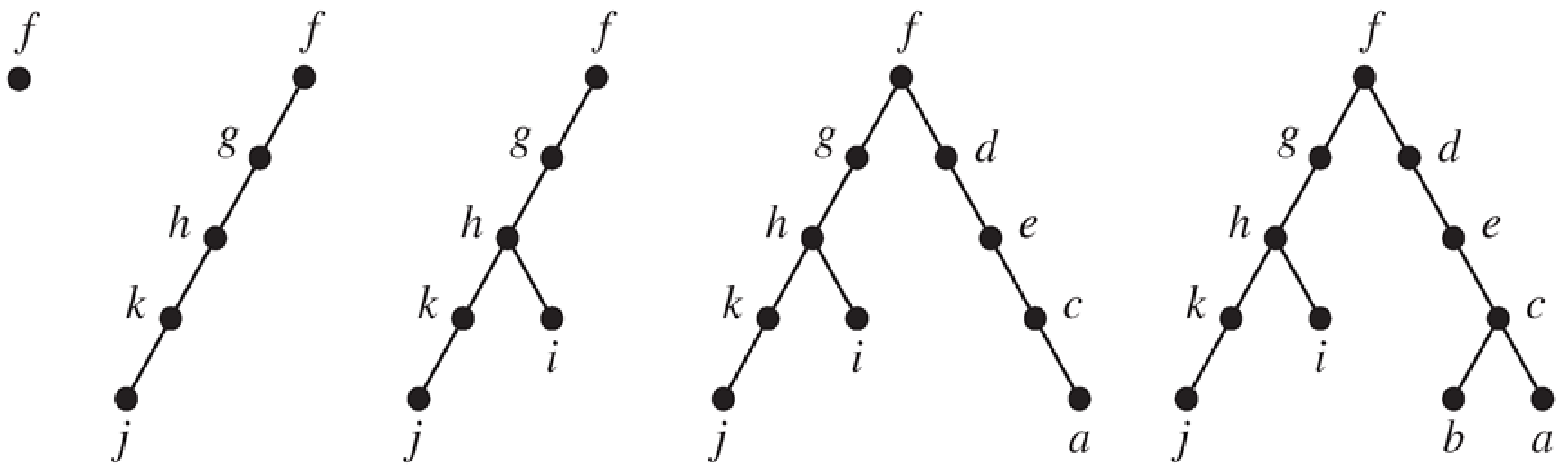
Depth-First Search

- **Example**



- **Example**





Depth-First Search Algorithm

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(v: vertex of G)
for each vertex w adjacent to v and not yet in T
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

Depth-First Search Algorithm

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(v: vertex of G)
for each vertex w adjacent to v and not yet in T
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

time complexity: $O(e)$

Breadth-First Search

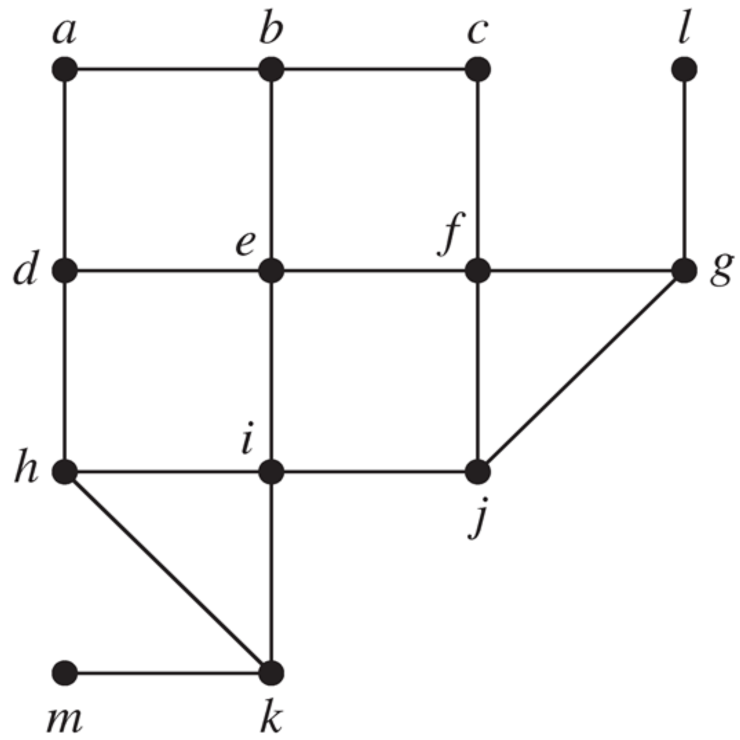
- This is the second algorithm that we build up spanning trees by successively adding edges.

Breadth-First Search

- This is the second algorithm that we build up spanning trees by successively adding edges.
  - ◇ First arbitrarily choose a vertex of the graph as the root.
  - ◇ Form a path by adding all edges incident to this vertex and the other endpoint of each of these edges
  - ◇ For each vertex added at the previous level, add edge incident to this vertex, as long as it does not produce a simple circuit.
  - ◇ Continue in this manner until all vertices have been added.

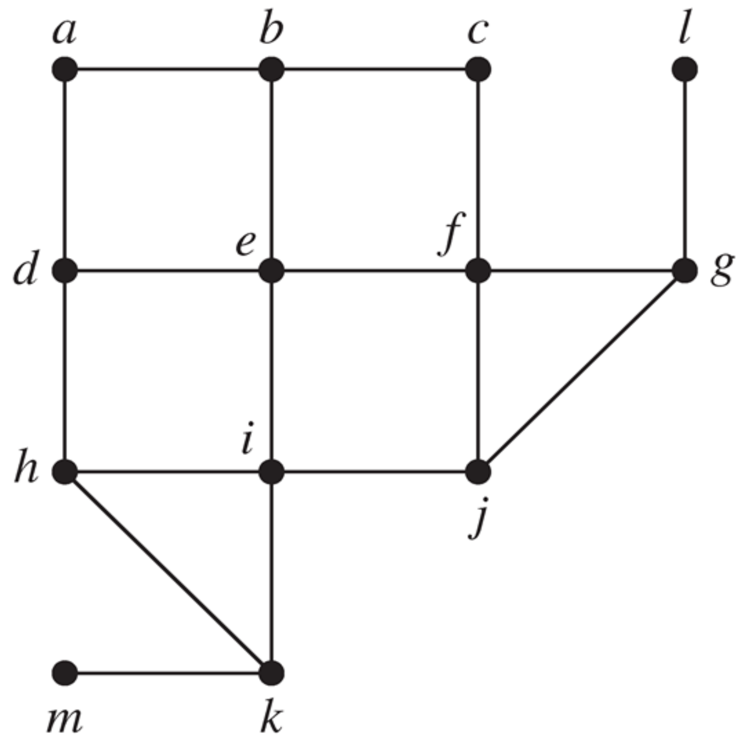
Breadth-First Search

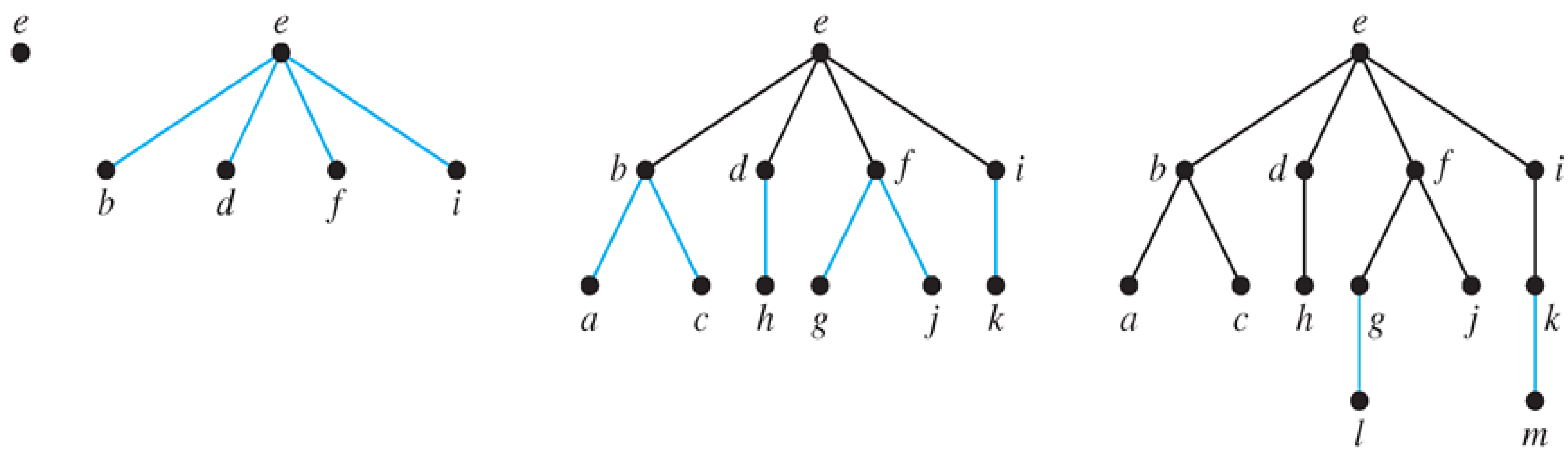
- **Example**



Breadth-First Search

- **Example**





Breadth-First Search

procedure *BFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
L := empty list *visit*(v_1)
put v_1 in the list L of unprocessed vertices
while L is not empty
    remove the first vertex, v, from L
    for each neighbor w of v
        if w is not in L and not in T **then**
            add w to the end of the list L
            add w and edge $\{v,w\}$ to T

Breadth-First Search

procedure *BFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
L := empty list *visit*(v_1)
put v_1 in the list L of unprocessed vertices
while L is not empty
    remove the first vertex, v, from L
    for each neighbor w of v
        if w is not in L and not in T **then**
            add w to the end of the list L
            add w and edge $\{v,w\}$ to T

time complexity: $O(e)$

Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...

Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...

  find shortest paths, determine whether bipartite, ...

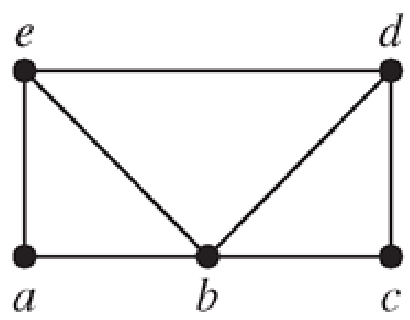
Applications of DFS, BFS

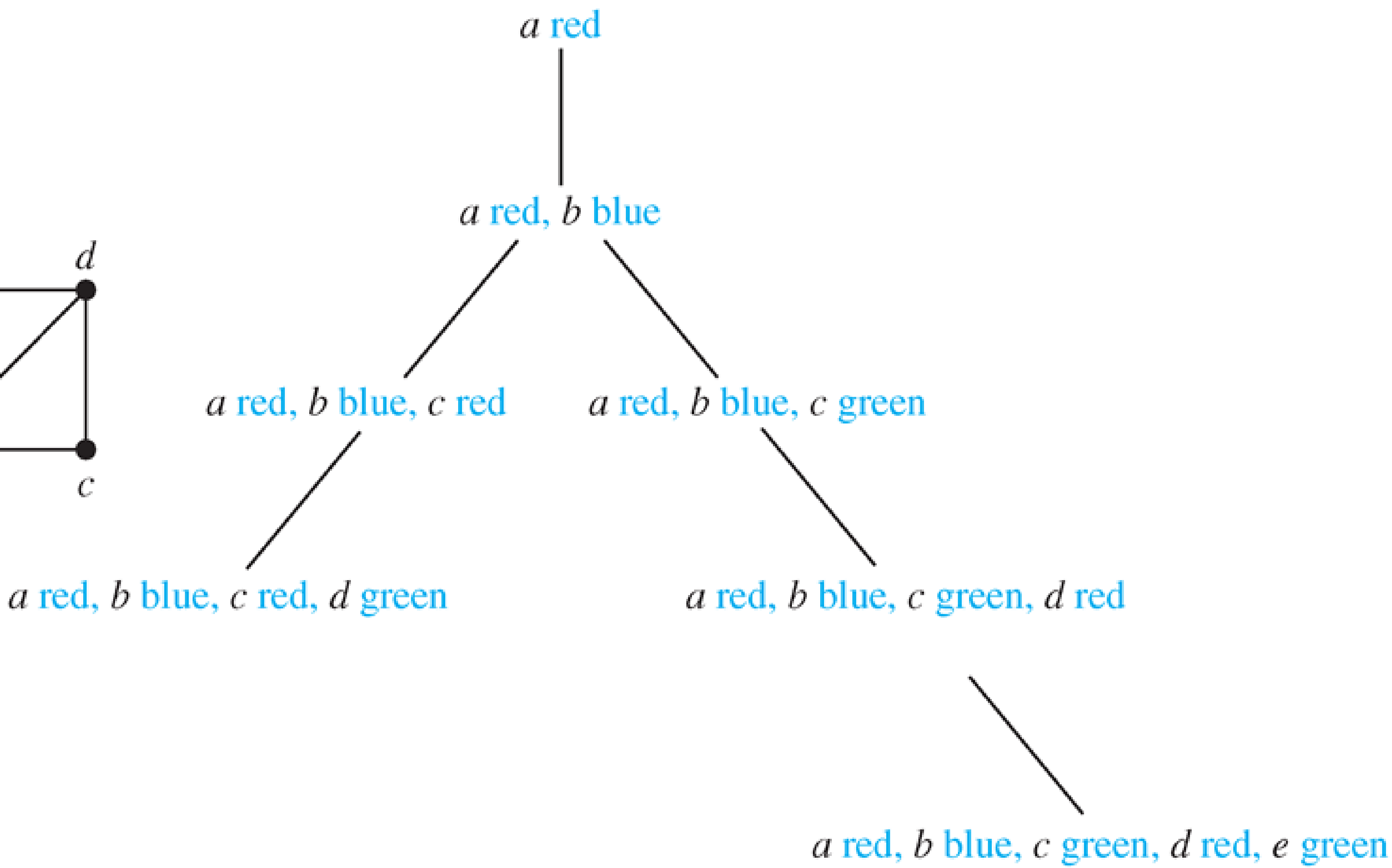
- find paths, circuits, connected components, cut vertices, ...

  find shortest paths, determine whether bipartite, ...

  graph coloring, sums of subsets, ...

Applications of DFS, BFS



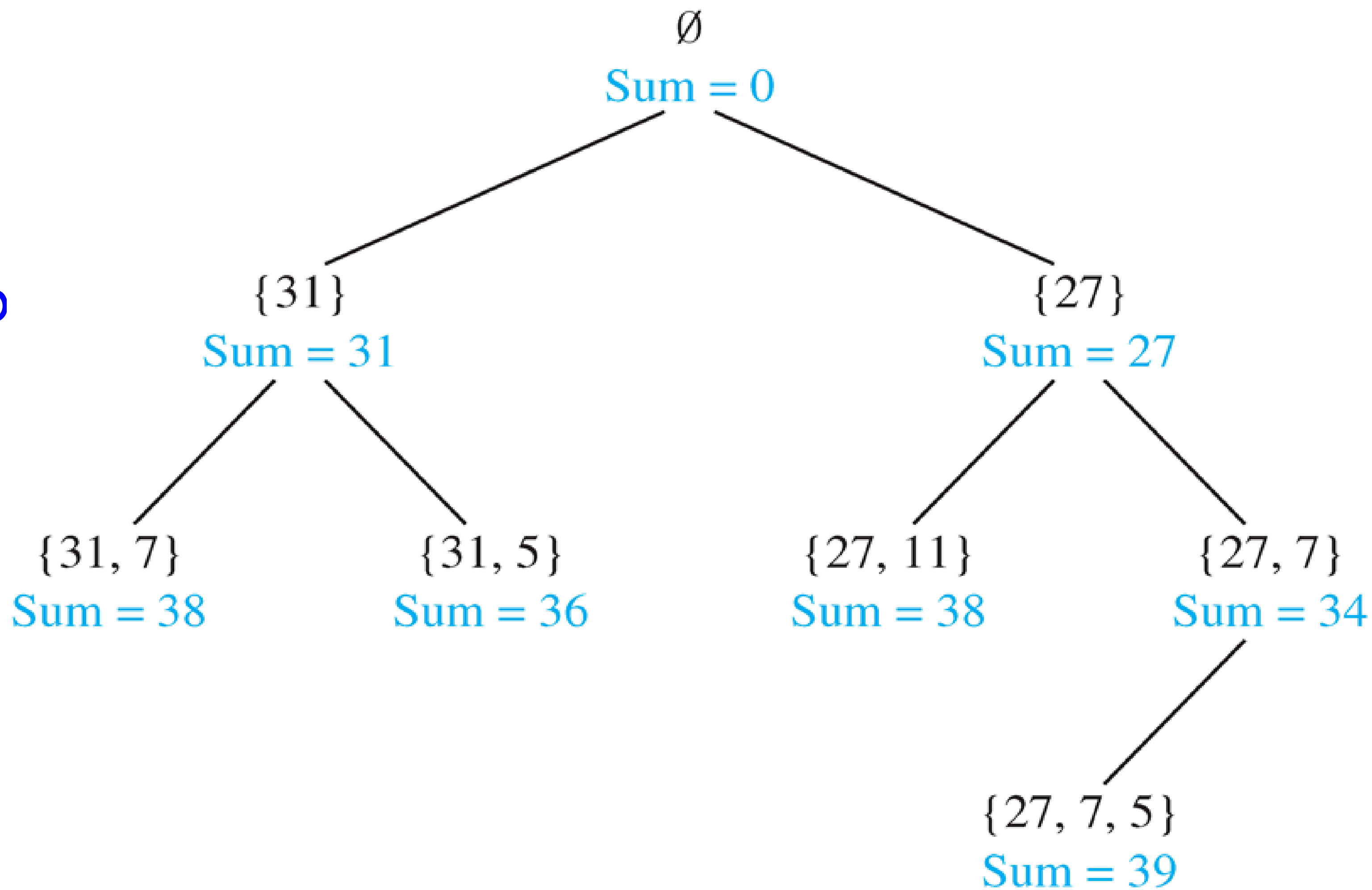


Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...

  find

  grap



find a subset of $\{31, 27, 15, 11, 7, 5\}$ with the sum 39

Minimum Spanning Trees

- **Definition** A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

- **Definition** A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

  two greedy algorithms:
  Prim's Algorithm, Kruscal's Algorithm

Prim's Algorithm

ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
    e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
    $T := T$ with e added
return T {T is a minimum spanning tree of G}

Prim's Algorithm

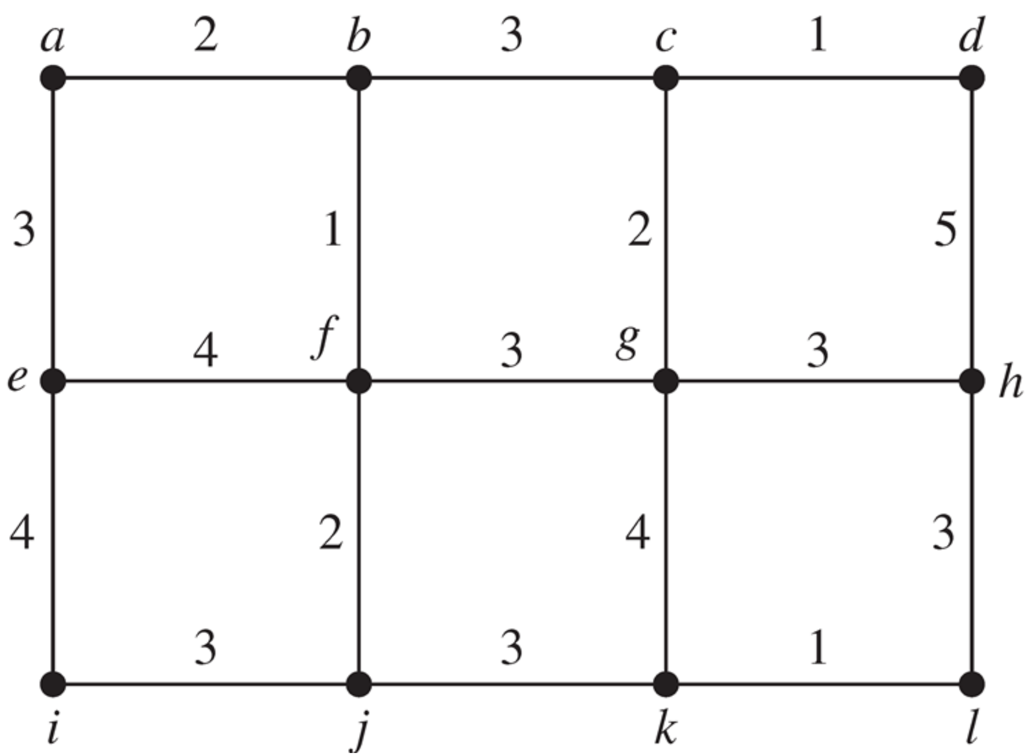
ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
    e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
    $T := T$ with e added
return T {T is a minimum spanning tree of G}

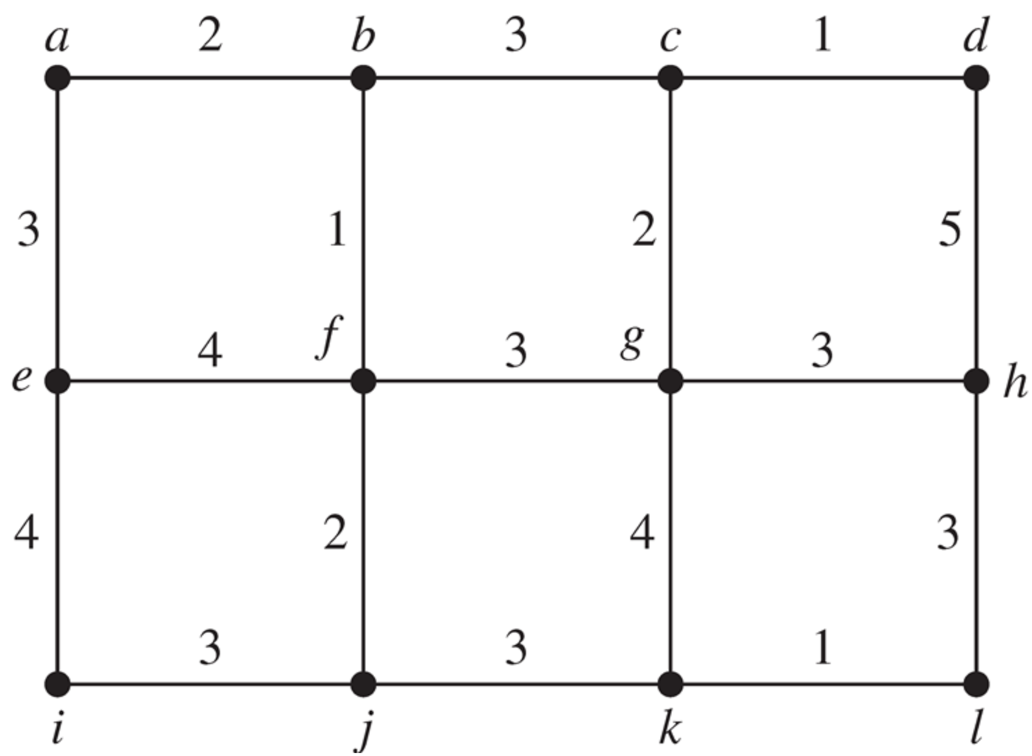
We can maintain a *heap* of all the edges with at least one endpoint in T, and in each iteration, we do Extract-Mins until we see an edge that has one endpoint in T and one endpoint not in T.

time complexity: $e \log v$

- **Example**



- **Example**



Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

Prim's Algorithm: Correctness

- **Proof** by *induction*.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

  If $e \in M$, this is clearly true. Since by i.h., $T \subset M$ and $e \in M$, then $T \cup \{e\} \subset M$. Now suppose that $e \notin M$.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

  If $e \in M$, this is clearly true. Since by i.h., $T \subset M$ and $e \in M$, then $T \cup \{e\} \subset M$. Now suppose that $e \notin M$.

  If we add e to M, then a circuit will be created in M. Since e has one endpoint in T and the other endpoint not in T, there has to be some other edge e' in this circuit that has exactly one endpoint in T.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

  If $e \in M$, this is clearly true. Since by i.h., $T \subset M$ and $e \in M$, then $T \cup \{e\} \subset M$. Now suppose that $e \notin M$.

  If we add e to M, then a circuit will be created in M. Since e has one endpoint in T and the other endpoint not in T, there has to be some other edge e' in this circuit that has exactly one endpoint in T.

  Since Prim's algorithm has chosen to add e, we have $w(e) \leq w(e')$. So if we add e to M and remove e' from M, we will have a new tree M' whose total weight $\leq$ that of M, and $T \cup \{e\} \subset M'$.

Kruscal's Algorithm

ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
  e := any edge in G with smallest weight that does not form a simple circuit when added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

Kruscal's Algorithm

ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
    e := any edge in G with smallest weight that does not form a simple circuit
        when added to T
    $T := T$ with e added
return T {T is a minimum spanning tree of G}

time complexity: $e \log e$ *Union-Find*

Kruscal's Algorithm

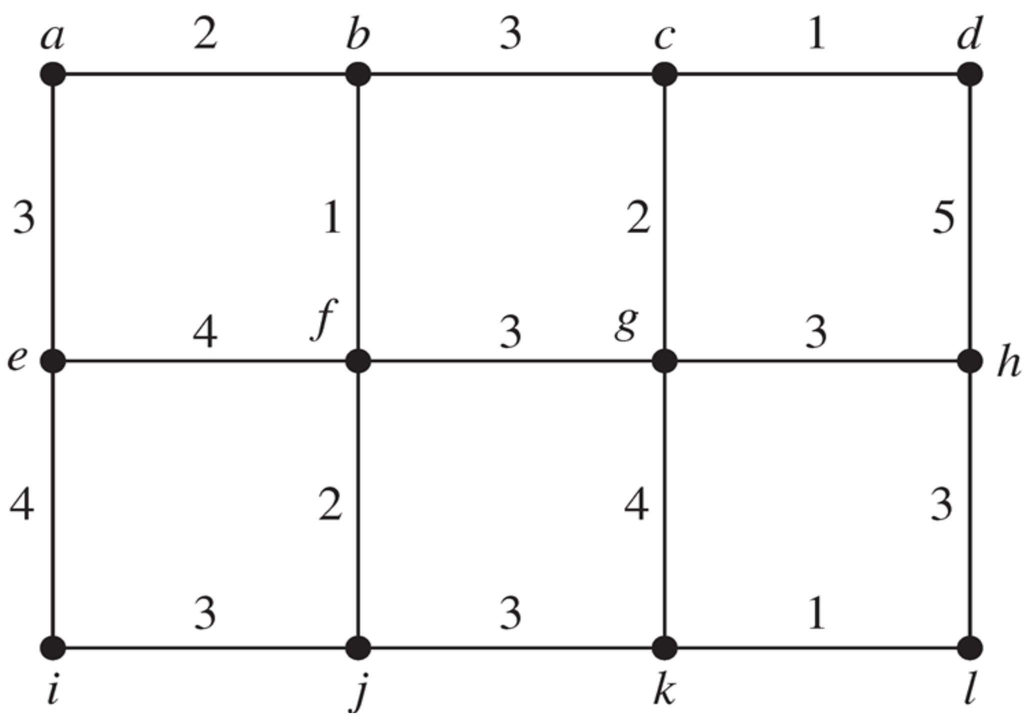
ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
    e := any edge in G with smallest weight that does not form a simple circuit when added to T
    $T := T$ with e added
return T {T is a minimum spanning tree of G}

time complexity: $e \log e$ *Union-Find*
see *CLRS* / *Algorithm Design*, J. Kleinberg, E. Tardos

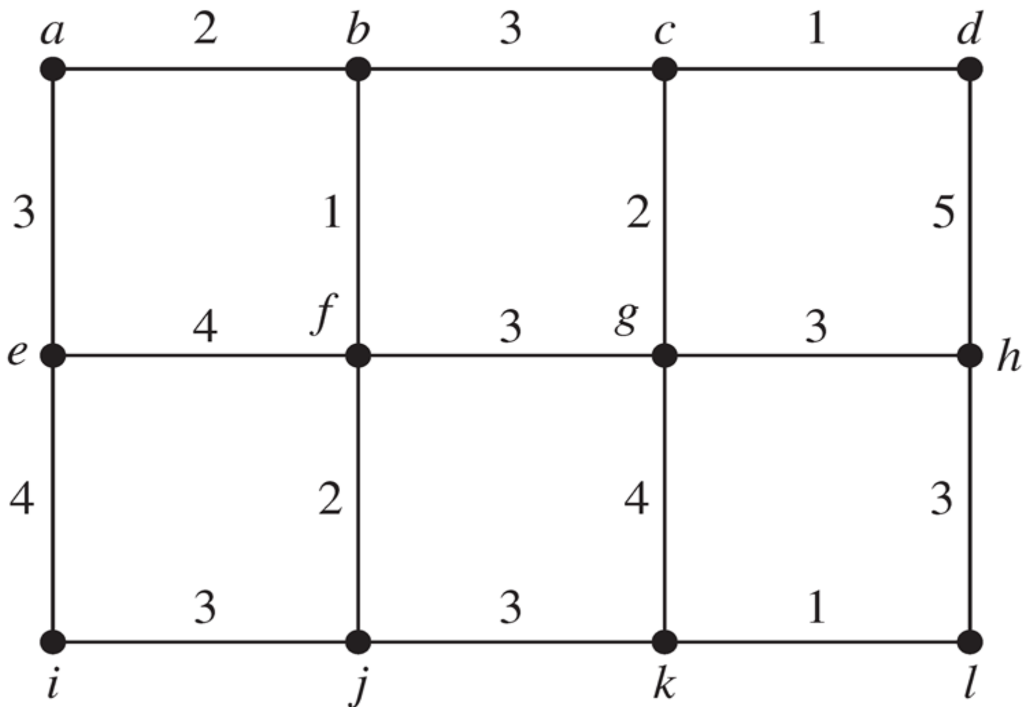
Kruscal's Algorithm

- **Example**



Kruscal's Algorithm

Example



Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

Kruscal's Algorithm: Correctness

- **Proof** by *induction*. Similar to Prim's algorithm.

Kruscal's Algorithm: Correctness

- **Proof** by *induction*. Similar to Prim's algorithm.
- A unifying structure: Given a graph $G = (V, E)$, every subset $S \subseteq V$ defines a *cut* $(S, \bar{S})$. We consider the edge set between S and $\bar{S}$.

Kruscal's Algorithm: Correctness

- **Proof** by *induction*. Similar to Prim's algorithm.
- A unifying structure: Given a graph $G = (V, E)$, every subset $S \subseteq V$ defines a *cut* $(S, \bar{S})$. We consider the edge set between S and $\bar{S}$.

  Theorem Let $(S, \bar{S})$ be an arbitrary cut, and let e be an edge across the cut (one endpoint in S, the other in $\bar{S}$) that has the smallest weight of all edges cross the cut. Then there must be an MST T containing e.

  Theorem Let $(S, \bar{S})$ be an arbitrary cut, and let E' be the set of edges across the cut of minimum weight ($w(e) = w(e')$ for any two edges $e, e' \in E'$ and $w(e) < w(e')$ for any $e \in E'$ and $e' \notin E'$). Let T be an arbitrary MST. Then T must contain some edge in E'.

NP-complete Problems

- Class **NP** vs Class **P**
  - **P**: decision problems solvable in polynomial time
  - **NP**: decision problems with certificates verifiable in polynomial time (polynomial time verification)

NP-complete Problems

- Class **NP** vs Class **P**
  - **P**: decision problems solvable in polynomial time
  - **NP**: decision problems with certificates verifiable in polynomial time (polynomial time verification)

- Some examples in Class NP, but will focus on intuition
  More reading:
  CLRS / M. Sipser: Introduction to Theory of Computation

NP-complete Problems

- Class **NP** vs Class **P**
  – **P**: decision problems solvable in polynomial time
  – **NP**: decision problems with certificates verifiable in polynomial time (polynomial time verification)

- Some examples in Class NP, but will focus on intuition
  More reading:
  CLRS / M. Sipser: Introduction to Theory of Computation

- Approximation Algorithm
  Natural idea: settle for *non-optimal* solutions for these “hard” problems, if we can find such close-to-the-optimal solutions reasonably fast.

Satisfiability Problem

- Satisfiability (*SAT*) – one of the most important **NP** problems

Satisfiability Problem

- Satisfiability (*SAT*) – one of the most important **NP** problems
- **Definition** A *Boolean formula* is a logical formula consisting of
  - Boolean variables (0 = false, 1 = true),
  - logical operations
    - ◇ $\neg x$: Negation
    - ◇ $x \vee y$: Disjunction
    - ◇ $x \wedge y$: Conjunction

  With the truth table defined by:

x	y	$\neg x$	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

Satisfiable

- A given Boolean formula is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables s.t. the final result is 1.

Satisfiable

- A given Boolean formula is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables s.t. the final result is 1.

Example. $f(x, y, z) = (x \wedge (y \vee \neg z)) \vee (\neg y \wedge z \wedge \neg x)$

x	y	z	$(x \wedge (y \vee \neg z))$	$(\neg y \wedge z \wedge \neg x)$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

Satisfiable

- A given Boolean formula is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables s.t. the final result is 1.

Example. $f(x, y, z) = (x \wedge (y \vee \neg z)) \vee (\neg y \wedge z \wedge \neg x)$

x	y	z	$(x \wedge (y \vee \neg z))$	$(\neg y \wedge z \wedge \neg x)$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

The assignment, $x = 1, y = 1, z = 0$ makes $f(x, y, z)$ true, and hence it is satisfiable.

Satisfiable

- **Example.** $f(x,y) = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$

x	y	$x \vee y$	$\neg x \vee y$	$x \vee \neg y$	$\neg x \vee \neg y$	$f(x,y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

Satisfiable

- **Example**. $f(x,y) = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$

x	y	$x \vee y$	$\neg x \vee y$	$x \vee \neg y$	$\neg x \vee \neg y$	$f(x,y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x,y)$ true, and hence it is NOT satisfiable.

Satisfiable

- **Example**. $f(x,y) = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$

x	y	$x \vee y$	$\neg x \vee y$	$x \vee \neg y$	$\neg x \vee \neg y$	$f(x,y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x,y)$ true, and hence it is NOT satisfiable.

Boolean formulas in the form of $f(x,y)$ are called 2-conjunctive normal form (2-CNF).

Satisfiable

- **Example**. $f(x,y) = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$

x	y	$x \vee y$	$\neg x \vee y$	$x \vee \neg y$	$\neg x \vee \neg y$	$f(x,y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x,y)$ true, and hence it is NOT satisfiable.

Boolean formulas in the form of $f(x,y)$ are called 2-conjunctive normal form (2-CNF).

Definition For a fixed k, Boolean formulas in the following form are called *k*-conjunctive normal form (k-CNF):

$$f_1 \wedge f_2 \wedge \cdots \wedge f_n$$

where each f_i is of the form $f_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,k}$, and each $y_{i,j}$ is a variable or the negation of a variable.

2SAT Problem

- **2SAT**
  Instance: A 2-CNF formula f
  Problem: To decide whether f is *satisfiable*

2SAT Problem

- **2SAT**
  Instance: A 2-CNF formula f
  Problem: To decide whether f is *satisfiable*

  Example a 2-CNF formula

$$(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$

2SAT Problem

- **2SAT**
  Instance: A 2-CNF formula f
  Problem: To decide whether f is *satisfiable*

  Example a 2-CNF formula

$$(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$

  Theorem 2SAT $\in$ Class **P**

2SAT Problem

- **2SAT**
  Instance: A 2-CNF formula f
  Problem: To decide whether f is *satisfiable*

  Example a 2-CNF formula
  $$(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$

  Theorem **2SAT** $\in$ Class **P**
  Proof. We will show how to solve 2SAT efficiently using path searches in graphs.

Path Searching in Graphs

- **Theorem** Given a graph $G = (V, E)$ and two vertices $u, v \in V$, finding if there is a path from u to v in the graph G is polynomial-time decidable.

Path Searching in Graphs

- **Theorem** Given a graph $G = (V, E)$ and two vertices $u, v \in V$, finding if there is a path from u to v in the graph G is polynomial-time decidable.

  Proof.
  Use some basic search algorithms in graph theory (DFS/BFS).

Graph Construction from Boolean Formula

- For a Boolean formula, use vertex to represent each variable and a negation of a variable
- There is an edge $(x, y) \in E$ if and only if there exists a clause equivalent to $(\neg x \vee y)$

Graph Construction from Boolean Formula

- For a Boolean formula, use vertex to represent each variable and a negation of a variable
- There is an edge $(x, y) \in E$ if and only if there exists a clause equivalent to $(\neg x \vee y)$

Example

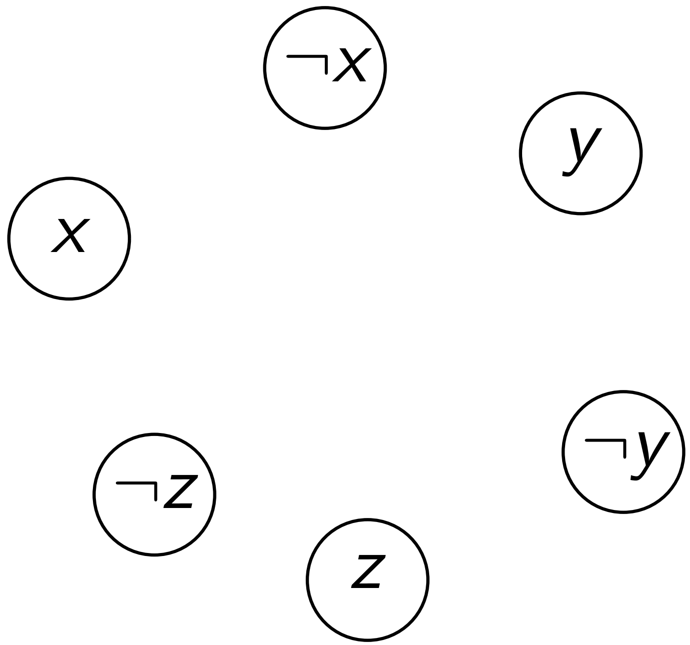
$$f = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$

Graph Construction from Boolean Formula

- For a Boolean formula, use vertex to represent each variable and a negation of a variable
- There is an edge $(x, y) \in E$ if and only if there exists a clause equivalent to $(\neg x \vee y)$

Example

$$f = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$

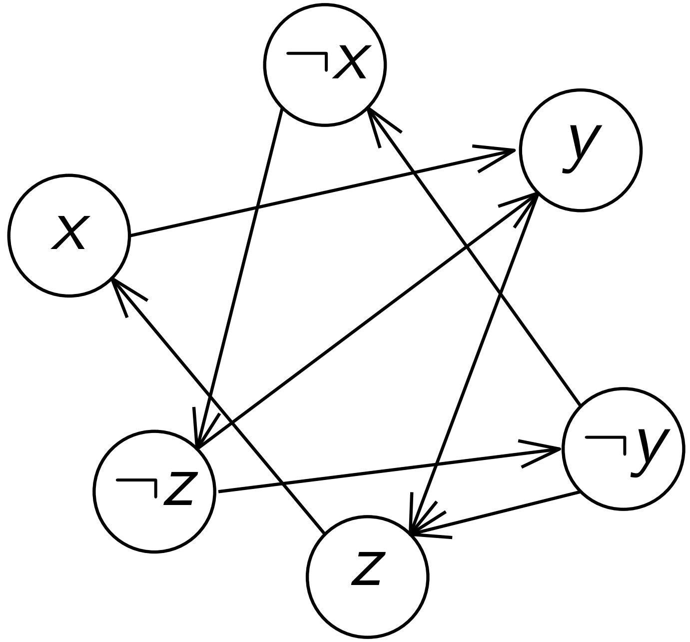


Graph Construction from Boolean Formula

- For a Boolean formula, use vertex to represent each variable and a negation of a variable
- There is an edge $(x, y) \in E$ if and only if there exists a clause equivalent to $(\neg x \vee y)$

Example

$$f = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$



Observation

- **Claim** If the graph G contains a path from x to y, then it also contains a path from $\neg y$ to $\neg x$.

Observation

- **Claim** If the graph G contains a path from x to y, then it also contains a path from $\neg y$ to $\neg x$.

  Proof. If there is an edge $(x, y) \in E$, this means the clause $\neg x \vee y$ is included in the 2-CNF. The equivalent clause $y \vee \neg x$ implies that $(\neg y, \neg x) \in E$.

Observation

- **Claim** If the graph G contains a path from x to y, then it also contains a path from $\neg y$ to $\neg x$.

  Proof. If there is an edge $(x, y) \in E$, this means the clause $\neg x \vee y$ is included in the 2-CNF. The equivalent clause $y \vee \neg x$ implies that $(\neg y, \neg x) \in E$.

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

Observation

- **Claim** If the graph G contains a path from x to y, then it also contains a path from $\neg y$ to $\neg x$.

  Proof. If there is an edge $(x, y) \in E$, this means the clause $\neg x \vee y$ is included in the 2-CNF. The equivalent clause $y \vee \neg x$ implies that $(\neg y, \neg x) \in E$.

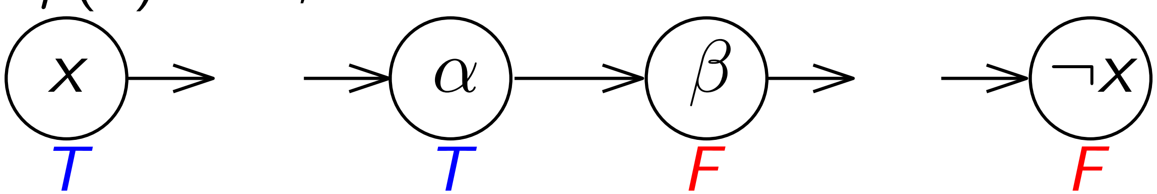
- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G
  Proof. Suppose that there are paths $x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$ for some variable x. For any possible assignment ρ:
  If $\rho(x) = T$,

Observation

- **Claim** If the graph G contains a path from x to y, then it also contains a path from $\neg y$ to $\neg x$.

  Proof. If there is an edge $(x, y) \in E$, this means the clause $\neg x \vee y$ is included in the 2-CNF. The equivalent clause $y \vee \neg x$ implies that $(\neg y, \neg x) \in E$.

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G
  Proof. Suppose that there are paths $x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$ for some variable x. For any possible assignment ρ:
  If $\rho(x) = T$,

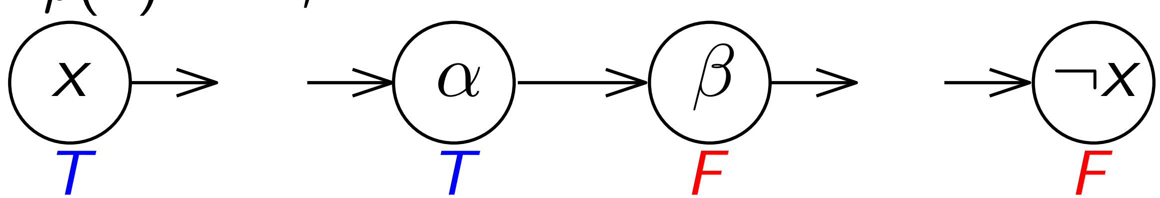


Observation

- **Claim** If the graph G contains a path from x to y, then it also contains a path from $\neg y$ to $\neg x$.

  Proof. If there is an edge $(x, y) \in E$, this means the clause $\neg x \vee y$ is included in the 2-CNF. The equivalent clause $y \vee \neg x$ implies that $(\neg y, \neg x) \in E$.

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G
  Proof. Suppose that there are paths $x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$ for some variable x. For any possible assignment ρ:
  If $\rho(x) = T$,



$(\neg\alpha \vee \beta)$ is false!

Correctness Proof II

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

  Proof. (cont')
  Suppose that there are no such paths ($x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$). We will find a satisfiable assignment as follows.

Correctness Proof II

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

  Proof. (cont')
  Suppose that there are no such paths ($x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$). We will find a satisfiable assignment as follows.
  – Pick an unassigned literal α, with no path from α to $\neg\alpha$, and assign it T
  – Assign T to all reachable vertices
  – Assign F to all their negations
  – Repeat until all vertices are assigned

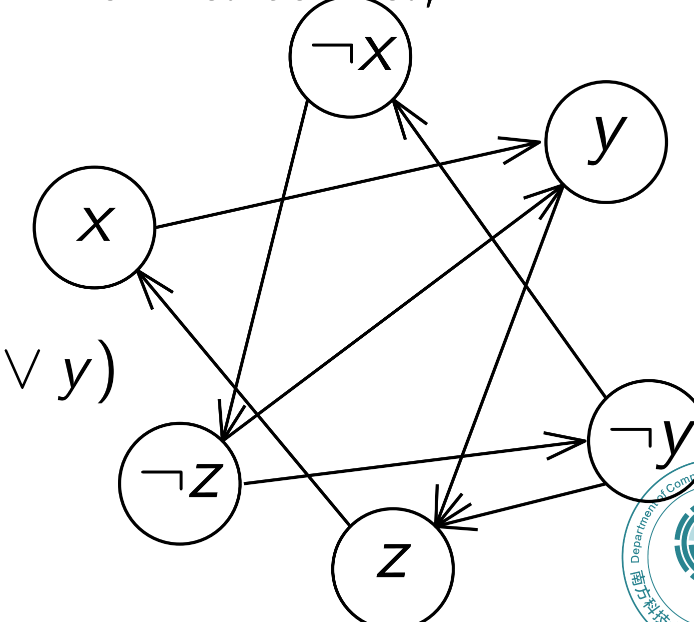
Correctness Proof II

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

Proof. (cont')
Suppose that there are no such paths ($x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$). We will find a satisfiable assignment as follows.

– Pick an unassigned literal α, with no path from α to $\neg\alpha$, and assign it T
– Assign T to all reachable vertices
– Assign F to all their negations
– Repeat until all vertices are assigned

$f = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$



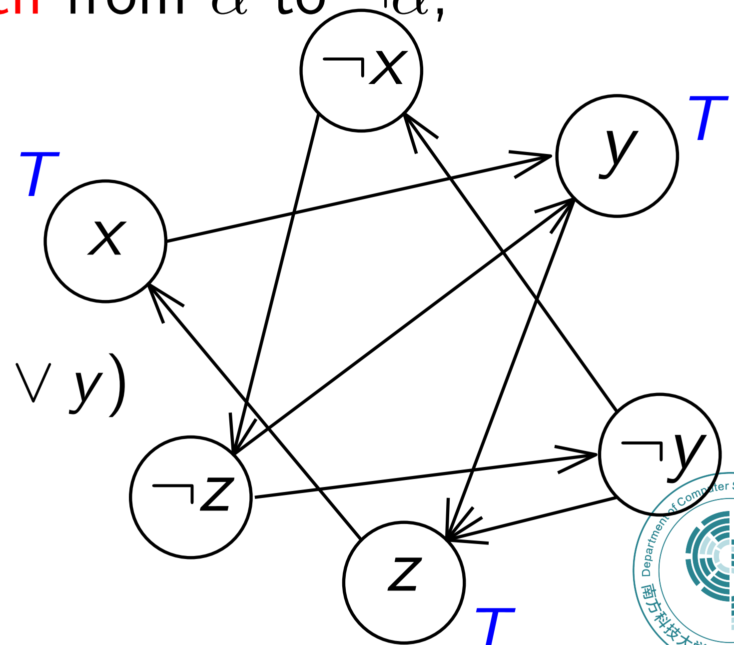
Correctness Proof II

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

Proof. (cont')
Suppose that there are no such paths ($x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$). We will find a satisfiable assignment as follows.

– Pick an unassigned literal α, with no path from α to $\neg\alpha$, and assign it T
– Assign T to all reachable vertices
– Assign F to all their negations
– Repeat until all vertices are assigned

$f = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$



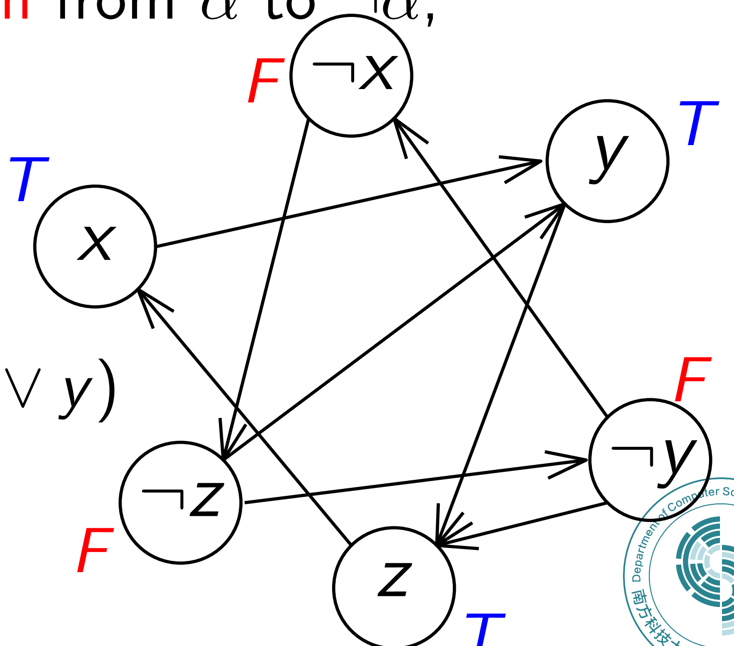
Correctness Proof II

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

Proof. (cont')
Suppose that there are no such paths ($x \to \cdots \to \neg x$ and $\neg x \to \cdots \to x$). We will find a satisfiable assignment as follows.

– Pick an unassigned literal α, with no path from α to $\neg\alpha$, and assign it T
– Assign T to all reachable vertices
– Assign F to all their negations
– Repeat until all vertices are assigned

$f = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$



Correctness Proof II

- **Claim** The assignment algorithm is well defined.

Correctness Proof II

- **Claim** The assignment algorithm is well defined.

  Proof.
  There cannot be two paths from x to both y and $\neg y$ (this will lead to the same assignment for both y and $\neg y$, contradiction!).

Correctness Proof II

- **Claim** The assignment algorithm is well defined.

  Proof.
  There cannot be two paths from x to both y and $\neg y$ (this will lead to the same assignment for both y and $\neg y$, contradiction!).

  If so, there will be a path from x to $\neg y$, and also a path from $\neg y$ to $\neg x$ (why?). Put the two paths together, we will have a path from x to $\neg x$, contradiction!

Correctness Proof II

- **Claim** The assignment algorithm is well defined.

  Proof.
  There cannot be two paths from x to both y and $\neg y$ (this will lead to the same assignment for both y and $\neg y$, contradiction!).

  If so, there will be a path from x to $\neg y$, and also a path from $\neg y$ to $\neg x$ (why?). Put the two paths together, we will have a path from x to $\neg x$, contradiction!

- **Claim** A 2-CNF formula f is unsatisfiable if and only if there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

Correctness Proof II

- **Claim** The assignment algorithm is well defined.

  Proof.
  There cannot be two paths from x to both y and $\neg y$ (this will lead to the same assignment for both y and $\neg y$, contradiction!).

  If so, there will be a path from x to $\neg y$, and also a path from $\neg y$ to $\neg x$ (why?). Put the two paths together, we will have a path from x to $\neg x$, contradiction!

- **Claim** A 2-CNF formula f is unsatisfiable if and only if
  there exists a variable x such that:
  – there is a path from x to $\neg x$ in the graph G
  – there is a path from $\neg x$ to x in the graph G

  Theorem A 2-CNF formula f is satisfiable if and only if
  there are no paths from x to $\neg x$ or from $\neg x$ to x for any literal x.

2SAT $\in$ P

- An efficient algorithm for **2SAT** is the following.
  - In the constructed graph G, for each variable x, check whether there is a path from x to $\neg x$ and vice versa.
  - Output NO if any of these tests succeeds.
  - Output YES otherwise.

2SAT $\in$ P

- An efficient algorithm for **2SAT** is the following.
  - In the constructed graph G, for each variable x, check whether there is a path from x to $\neg x$ and vice versa.
  - Output NO if any of these tests succeeds.
  - Output YES otherwise.

- **Theorem 2SAT** $\in$ Class **P**

Polynomial-Time Reduction

- Let L_1 and L_2 be two decision problems

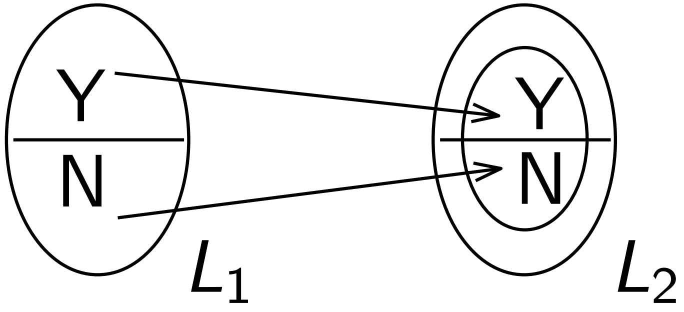
Polynomial-Time Reduction

- Let L_1 and L_2 be two decision problems
- A *polynomial-time reduction* from L_1 to L_2 is a transformation f with the following two properties:

  (1) f transforms an input x for L_1 into an input $f(x)$ for L_2 s.t.

  a yes-input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2

  (2) f is computable in *polynomial time* in size(x)



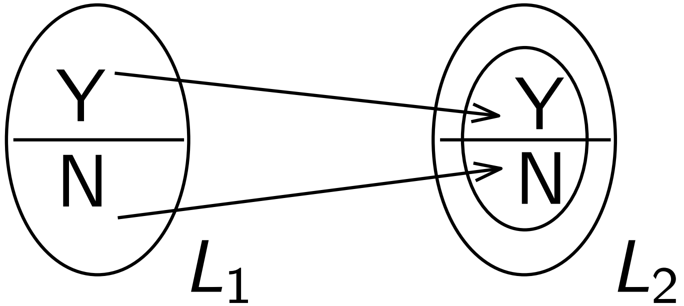
Polynomial-Time Reduction

- Let L_1 and L_2 be two decision problems
- A *polynomial-time reduction* from L_1 to L_2 is a transformation f with the following two properties:

  (1) f transforms an input x for L_1 into an input $f(x)$ for L_2 s.t.

  a yes-input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2

  (2) f is computable in *polynomial time* in size(x)



If such an f exists, we say that L_1 is *polynomial-time reducible* to L_2, and write $L_1 \leq_P L_2$.

Polynomial-Time Reduction

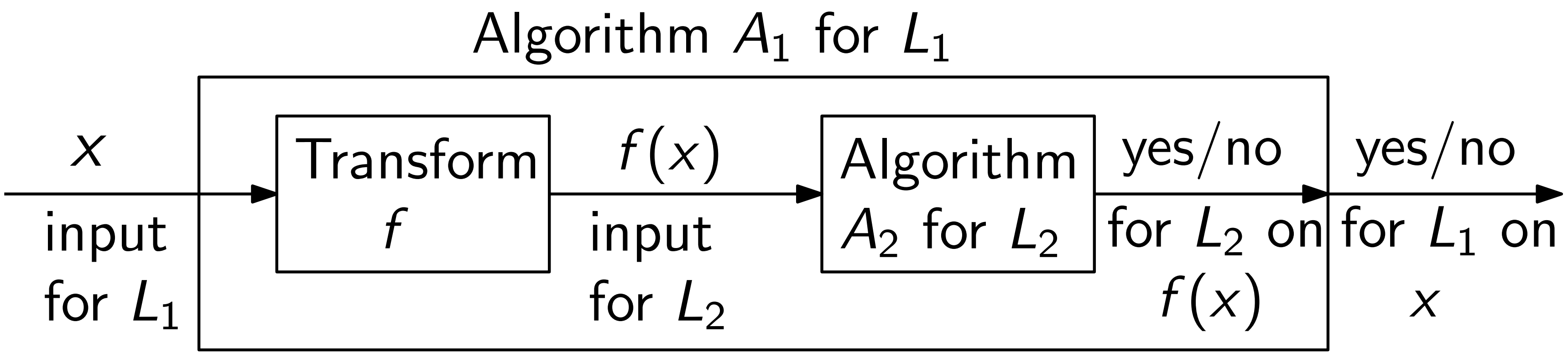
- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2

Polynomial-Time Reduction

- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2
- Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1:

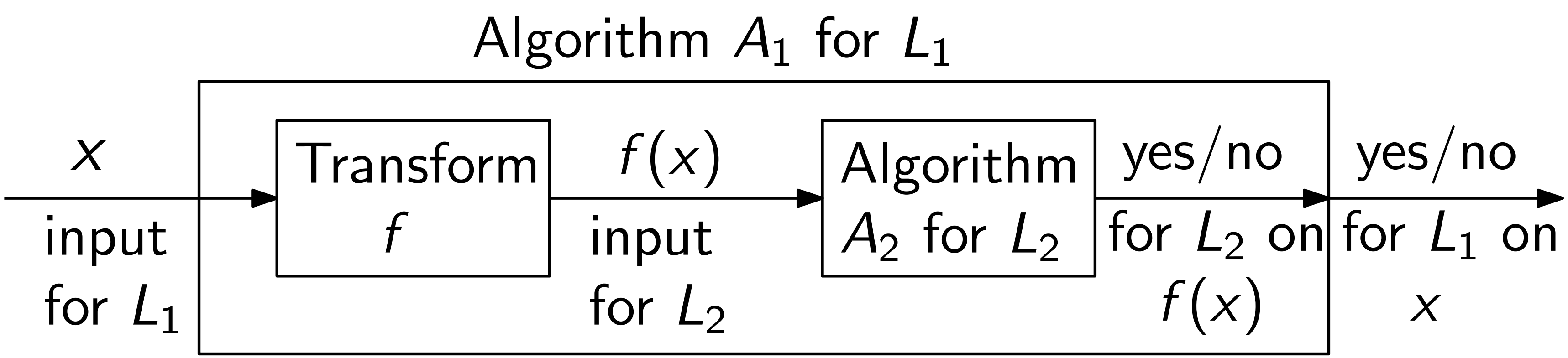
Polynomial-Time Reduction

- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2
- Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1:



Polynomial-Time Reduction

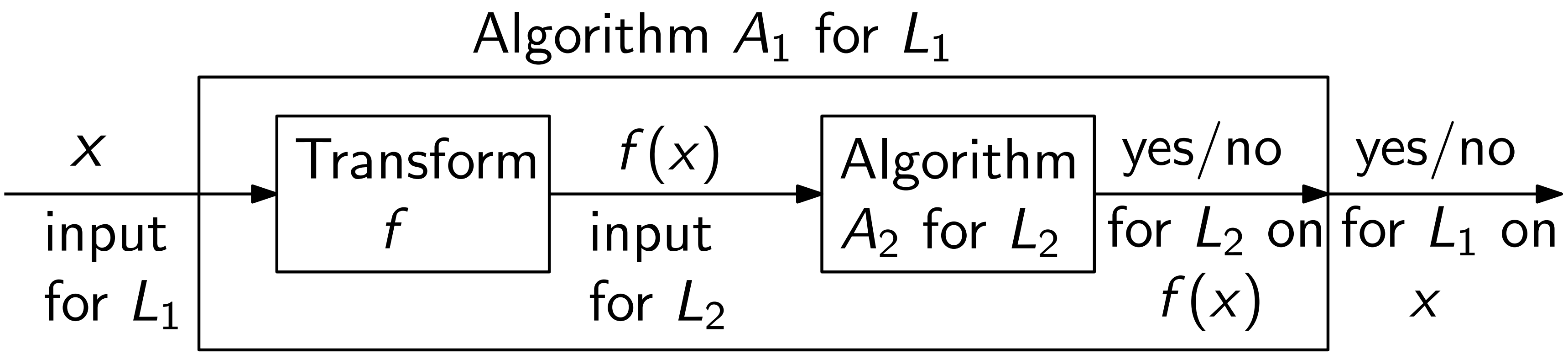
- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2
- Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1:



- If A_2 is polynomial-time algorithm, so is A_1

Polynomial-Time Reduction

- Intuitively, $L_1 \leq_P L_2$ means that L_1 is no harder than L_2
- Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1:



- If A_2 is polynomial-time algorithm, so is A_1

Theorem If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$

Lemma If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

The Class NP-Complete (NPC)

- The Class *NPC* consists of all decision problems L s.t.
  (1) $L \in NP$
  (2) for every $L' \in NP$, $L' \leq_P L$

The Class NP-Complete (NPC)

- The Class *NPC* consists of all decision problems L s.t.
  (1) $L \in NP$
  (2) for every $L' \in NP$, $L' \leq_P L$

  From the definition of *NPC*, it seems impossible to prove that one decision problem $L \in NPC$.
  – By definition, it requires to show every $L' \in NP$, $L' \leq_P L$.
  – But there are infinitely many problems in NP, so how can we argue there exists a reduction from every L' to L?

The Class NP-Complete (NPC)

- The Class *NPC* consists of all decision problems L s.t.
  (1) $L \in NP$
  (2) for every $L' \in NP$, $L' \leq_P L$

  From the definition of *NPC*, it seems impossible to prove that one decision problem $L \in NPC$.
  – By definition, it requires to show every $L' \in NP$, $L' \leq_P L$.
  – But there are infinitely many problems in NP, so how can we argue there exists a reduction from every L' to L?

  However, due to the transitivity property of $\leq_P$, we can do the following to prove a decision problem $L \in NPC$:
  – prove $L \in NP$ (usually easy)
  – for some $L' \in NPC$, prove $L' \leq_P L$

The Class NP-Complete (NPC)

- The Class *NPC* consists of all decision problems L s.t.
  (1) $L \in NP$
  (2) for every $L' \in NP$, $L' \leq_P L$

From the definition of *NPC*, it seems impossible to prove that one decision problem $L \in NPC$.
– By definition, it requires to show every $L' \in NP$, $L' \leq_P L$.
– But there are infinitely many problems in NP, so how can we argue there exists a reduction from every L' to L?

However, due to the transitivity property of $\leq_P$, we can do the following to prove a decision problem $L \in NPC$:
– prove $L \in NP$ (usually easy)
– for some $L' \in NPC$, prove $L' \leq_P L$

Proof. Let L'' be any problem in NP. Since $L' \in NPC$, by definition we have $L'' \leq_P L'$. Since $L' \leq_P L$, then by transitivity, we have $L'' \leq_P L$.

SAT $\in NPC$ (Cook's Theorem)

- **Theorem** (Cook's Theorem) **SAT** $\in NPC$.

SAT $\in$ *NPC* (Cook's Theorem)

- **Theorem** (Cook's Theorem) **SAT** $\in$ *NPC*.

  We will not prove this theorem, but will assume that **3SAT** $\in$ *NPC* as well. With this we will start to prove problems in Class **NPC**.

SAT $\in NPC$ (Cook's Theorem)

- **Theorem** (Cook's Theorem) **SAT** $\in NPC$.

  We will not prove this theorem, but will assume that **3SAT** $\in NPC$ as well. With this we will start to prove problems in Class **NPC**.

  We will prove:
  3SAT $\leq_P$ DCLIQUE
  DCLIQUE $\leq_P$ DVC

Next Lecture

- Graph NPC problems ...