COMP 3011
DESIGN AND ANALYSIS OF ALGORITHMS
FALL 2024

Dynamic Programming& Polynomial-time Reductions

LI Bo
Department of Computing
The Hong Kong Polytechnic University

Longest Common Subsequence

2

Longest Common Subsequence (LCS)

- ***Subsequence***: A subsequence of a given sequence is just the given sequence with zero or more elements left out.
- Given a sequence X $=< x_1, \cdots, x_m >$, another sequence Z $=< z_1, \cdots, z_k >$ is a subsequence of X if there exists a strictly increasing $i_1, i_2, \cdots, i_k$ of indices of X such that for all $j = 1, \cdots, k$, we have $x_{i_j} = z_j$.
- E.g. X $=< A, B, C, B, D, A, B >$
  $Z = < B, C, D, B >$ with index sequence 2, 3, 5, 7.

- ***Common Subsequence***: Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.
- E.g. $X =< A, B, C, B, D, A, B >$
  $Y =< B, D, C, A, B, A >$
  The sequence $< B, C, A >$ is a common subsequence of both X and Y.
  A longer common subsequence $< B, C, B, A >$
- ***Longest-Common-Subsequence (LCS)***: Given two sequences X $=< x_1, \cdots, x_m >$, and Y $=< y_1, \cdots, y_n >$, and wish to find a maximum length common subsequence of X and Y.

Longest Common Subsequence LCS

Prefix: Given $X =< x_1, \cdots, x_m >$, $X_i =< x_1, \cdots, x_i >$ for $i = 0,1, \cdots, m$ is a ***prefix*** of X.

- E.g. X $=< A, B, C, B, D, A, B >$
  $X_4 = < A, B, C, B >$ is a prefix.
- X_0 is the empty sequence.

Step 1: Optimal substructure of an LCS

Given X $=< x_1, \cdots, x_m >$, Y $=< y_1, \cdots, y_n >$ and Z $=< z_1, \cdots, z_k >$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z_k is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z_k is an LCS of X and Y_{n-1}.

If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. ← Appending $x_m = y_n$ to this LCS yields an LCS of X and Y .

If $x_m \neq y_n$, we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. ← Whichever of these two LCSs is longer is an LCS of X and Y

Longest Common Subsequence LCS

If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. ← Appending $x_m = y_n$ to this LCS yields an LCS of X and Y .

If $x_m \neq y_n$, we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. ← Whichever of these two LCSs is longer is an LCS of X and Y

Step 2: A Recursive solution

- Define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j
- If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0.
- Else

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Longest Common Subsequence LCS

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Step 3: Computing the length of an LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

help us construct/visualize an optimal solution.

$c[i,j]$: length of an LCS of X_i and Y_j

Running time: $O(mn)$

i \ j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest Common Subsequence LCS

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Step 3: Computing the length of an LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i-1, j-1] + 1
            b[i, j] = "↖"
        elseif c[i-1, j] ≥ c[i, j-1]
            c[i, j] = c[i-1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j-1]
            b[i, j] = "←"
return c and b
```

help us construct/visualize an optimal solution.

$c[i,j]$: length of an LCS of X_i and Y_j

Running time: $O(mn)$

X=ABCBDAB
Y=BDCABA

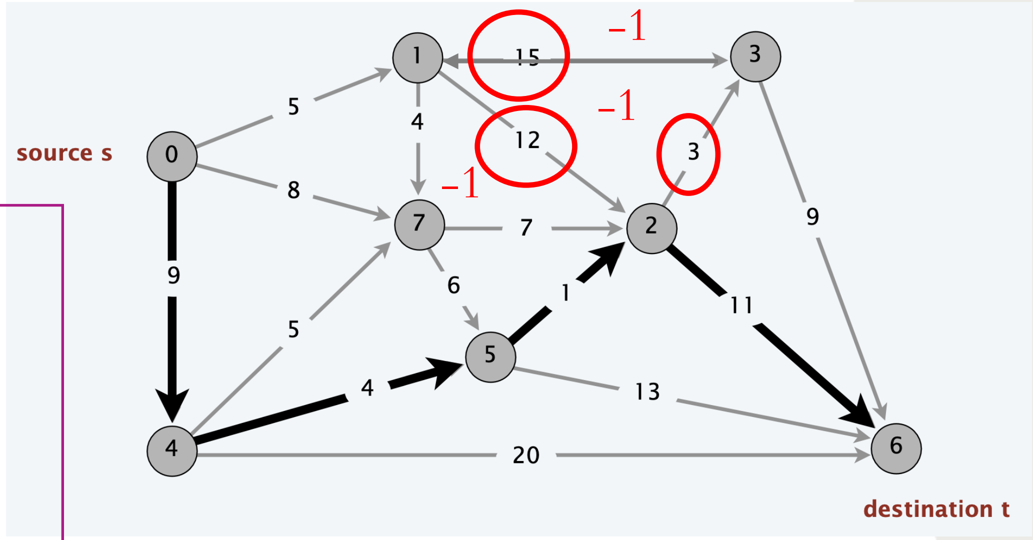
Shortest Path in Graphs

Shortest Path in Graphs

Let $G = (V, E)$ be a directed graph. Assume that each edge $(i, j) \in E$ has an associated weight c_{ij}, representing the cost for going directly from node i to node j in the graph.

Given the graph has no negative cycles, find a path P from an origin node s to a destination node t with minimum total cost:

$$\sum_{i \in P} c_{ij}$$



More complex setting:
the costs may be negative!
(with application in financial settings)

Assumption: there is no negative cycle -- that is, a directed cycle C such that

$$\sum_{i \in C} c_{ij} < 0 .$$

Remember ***Dijkstra's Algorithm***?

Bellman-Ford Algorithm

Lemma. If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most $n-1$ edges.

Proof:

- Since every cycle has nonnegative cost, the shortest path P from s to t with the fewest number of edges does not repeat any vertex v.
- For if P did repeat a vertex v, we could remove the portion of P between consecutive visits to v, resulting in a path of no greater cost and fewer edges.

- Let $OPT(i,v)$ denote the minimum cost of a $v-t$ path using at most i edges.
- Our problem is to compute $OPT(n-1,s)$.

Case 1.

If the path P uses at most $i-1$ edges, then

$$OPT(i,v) = OPT(i-1,v).$$

Case 2.

If the path P uses i edges, and the first edge is (v,w),

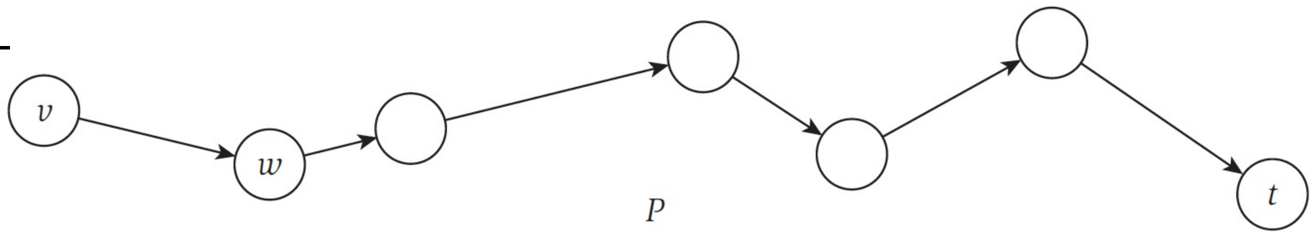
$$OPT(i,v) = c_{vw} + OPT(i-1,w).$$

$$OPT(0,v) = \infty$$

If $i > 0$ then $\quad OPT(0,t) = 0$

$$OPT(i,v) = \min\begin{cases} OPT(i-1,v) \\ \min\limits_{w\in V}(OPT(i-1,w) + c_{vw}) \end{cases}$$

$O(n)$ subproblems



Bellman-Ford Algorithm

Theorem. The Shortest-Path method correctly computes the minimum cost of an s-t path in any graph that has no negative cycles, and runs in $O(n^3)$ time

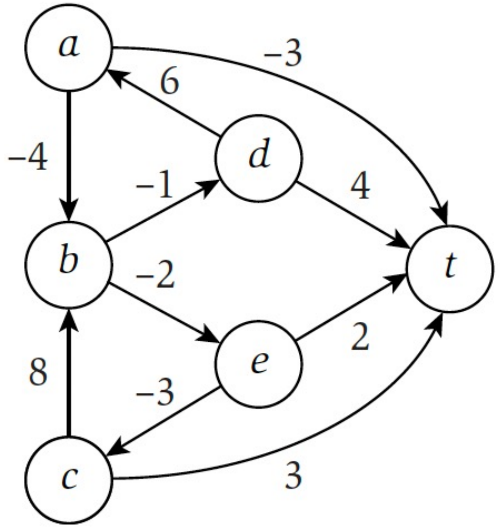
$$OPT(i,v) = \min \begin{cases} OPT(i-1,v) \\ \min\limits_{w\in V}(OPT(i-1,w) + c_{vw}) \end{cases}$$

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0 ... n − 1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For v ∈ V in any order
      Compute M[i, v] using the above recurrence
    Endfor
  Endfor
  Return M[n − 1, s]
```

containing the optimal values of the subproblems

table M has n^2 entries

each entry needs $O(n)$ time



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Elements of Dynamic Programming

Elements of Dynamic Programming

When should we look for a dynamic-programming solution to a problem?

Two key ingredients: optimal substructure and overlapping subproblems

Optimal Substructure

- Optimal substructure: An optimal solution to the problem contains within it optimal solutions to subproblems.
- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.

We must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

E.g. weighted interval scheduling

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{OPT(j-1),\ w_j + OPT(p(j))\} & \text{if } j > 0 \end{cases}$$

Overlapping Subproblems

- The space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.
- When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.

Divide-and-conquer approach is suitable usually generates brand-new problems

REDUCTIONS

REMEMBER FIBONACCI?

- ***Leonardo Fibonacci*** (Italian mathematician)
- But today Fibonacci is most widely known for his famous sequence of numbers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

- More formally, the Fibonacci numbers F_n are generated by the simple rule

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & n \geq 2 \\ 1, & n = 1 \\ 0, & n = 0 \end{cases}$$

- In fact, the Fibonacci numbers grow almost as fast as the powers of 2: for example, F_{30} is over a million, and F_{100} is already 21 digits long! In general, $F_n \approx 2^{0.694n}$.

A DIRECT ALGORITHM

$T(n) = T(n-1) + T(n-2) + 3$ for $n > 1$.

➢ By the recursive definition of F_n,

$T(n) \geq F_n \approx 2^{0.694n}$???

function $fib1(n)$

if $n = 0$: return 0   1 step

if $n = 1$: return 1   1 step

If $n \leq 1,\ T(n) \leq 2$

return $fib1(n-1) + fib1(n-2)$ $T(n-1) + T(n-2)$ +1 steps

➢ Whenever we have an algorithm, there are three questions we always ask:

- ➢ 1. Is it correct? ✓
- ➢ 2. How much time does it take, as a function of n?
- ➢ 3. And can we do better? ✓

A POLYNOMIAL ALGORITHM

- Why not save the known results?

  function $fib2(n)$

  if $n = 0$ return 0 1 step

  create an array $f[0,1,\cdots,n]$ 1 step

  $f[0] = 0, f[1] = 1$ 2 steps

  for $i = 2,\cdots,n$: $n-1$ rounds

  $f[i] = f[i-1] + f[i-2]$ 1 step

  $\}$ $n-1$ steps

  return $f[n]$ 1 step

- In total $\underline{T(n) = n + 4}$ steps.

  polynomial steps!!!

Conclusion

- Fibonacci numbers can be computed efficiently.
- But we need to be very careful and work hard.

REMEMBER SATISFIABILITY?

Can we do better?

- ***Literal***. A Boolean variable or its negation x_i or $\overline{x_i}$
- ***Clause***. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$
- ***Conjunctive normal form (CNF)***. A propositional formula Φ that is a conjunction of clauses.

- ***SAT***. Given a CNF formula Φ, does it have a satisfying truth assignment?
- ***3-SAT***. SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).

Exhaustive search: try all 2^n truth assignments.

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

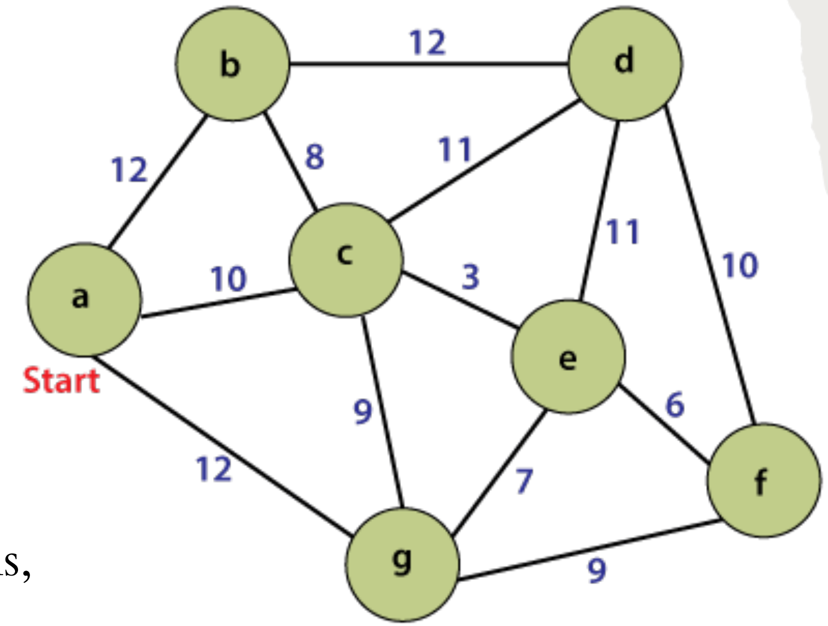
yes instance: x_1 = true, x_2 = true, x_3 = false, x_4 = false

Travelling Salesman Problem (TSP)

No known poly-time algorithm

We are given n cities $1, \cdots, n$, and a nonnegative integer distance d_{ij} between any two cities i and j (assume that the distances are symmetric, that is, $d_{ij} = d_{ji}$ for all i and j).

We are asked to find the shortest tour of the cities -- that is, the permutation π such that $\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}$ (where by $\pi(n+1)$ we mean $\pi(1)$) is as small as possible.



- We can solve this problem by enumerating all possible solutions, computing the cost of each, and picking the best.
- This would take time proportional to $n!$ (there are $\frac{1}{2}(n-1)!$ tours to be considered), which is not a polynomial bound.

Most outstanding and persistent failure.

ALGORITHM DESIGN PATTERNS

- Greedy.
- Divide and conquer.
- Dynamic programming.
- LP and Duality.
- Local search.
- Reductions.

REDUCTIONS

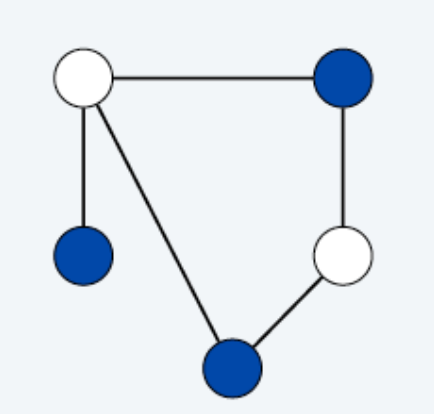
BIPARTITE GRAPHS

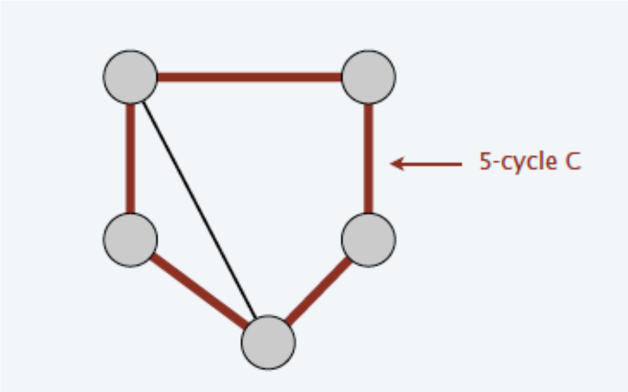
Lemma. Let G be a connected graph, and let $L_0, \dots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

i. No edge of G joins two nodes of the same layer, and G is bipartite.

ii. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Corollary. A graph G is bipartite if and only if it contains no odd-length cycle.





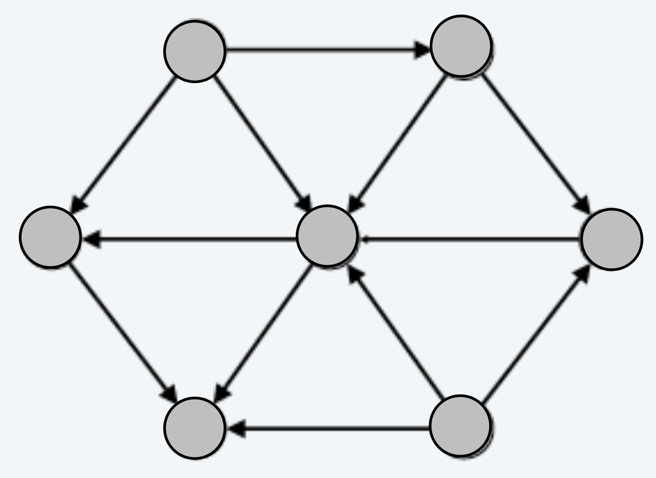
DIRECTED ACYCLIC GRAPHS

If the graph contains a cycle, then no linear ordering is possible.

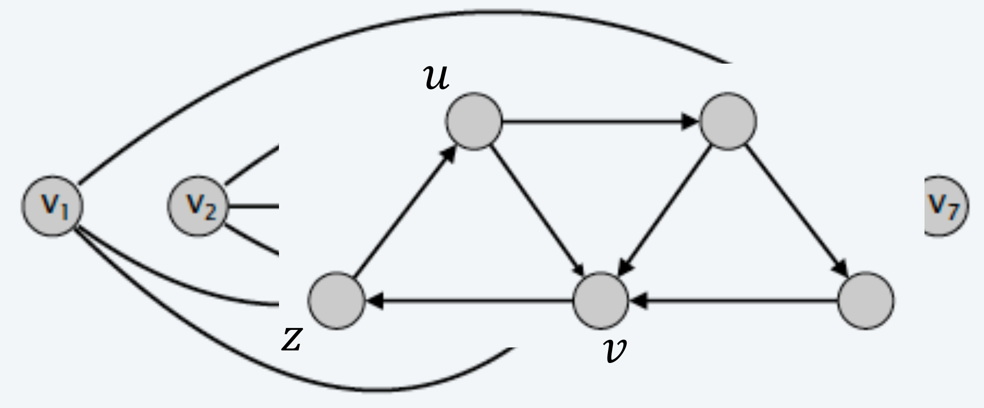
Definitions

An ordering of the nodes so that all edges point "forward".

- A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.
- A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \dots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



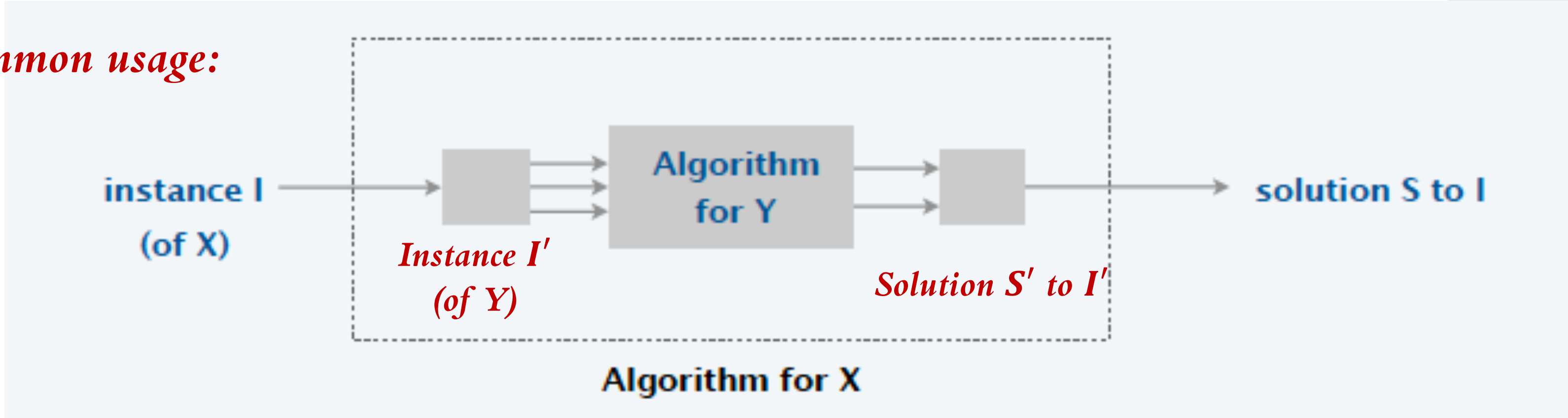
a DAG



a topological ordering

POLY-TIME REDUCTIONS

Common usage:



Desiderata. Suppose we could solve problem Y in polynomial time.

What else could we solve in polynomial time?

Reduction. Problem X polynomial-time reduces to problem Y ($X \leq_P Y$)
if ***arbitrary instances of problem X*** can be solved using:

- Polynomial number of standard computational steps, and
- Polynomial number of calls to ***oracle*** that solves problem Y.

Each primitive operation takes a constant amount of time.

- Primitive Operations:
  - Arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
  - Logic operations (and, or)
  - Read/write memory
  - Array indexing
  - Following a pointer
  - Data movement (load, store, copy)
  - Control (conditional and unconditional branch, subroutine call and return)

POLY-TIME REDUCTIONS

Design algorithms.

If $X \leq_P Y$ and Y can be solved in polynomial time, then X can be solved in polynomial time.

Establish intractability.

If $X \leq_P Y$ and X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.

Establish equivalence.

If both $X \leq_P Y$ and $Y \leq_P X$, we use notation $X \equiv_P Y$. In this case, X can be solved in polynomial time if and only if Y can be.

SET COVER AND VERTEX COVER

SET-COVER.

Given a set U of elements, a collection S of subsets of U, and an integer k, are there $\leq k$ of these subsets whose union is equal to U ?

VERTEX-COVER.

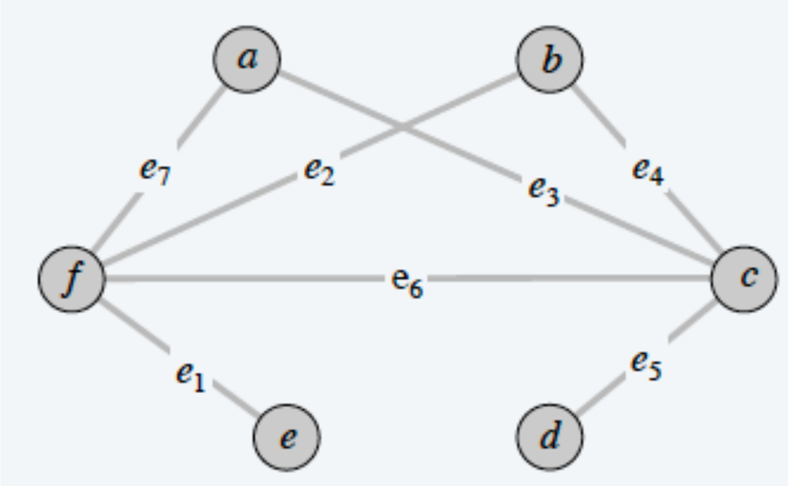
Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$S_a = \{3, 7\}$ $\quad$ $S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$ $\quad$ $S_d = \{5\}$

$S_e = \{1\}$ $\quad$ $S_f = \{1, 2, 6, 7\}$



SET COVER AND VERTEX COVER

Theorem. VERTEX-COVER $\leq_P$ SET-COVER.

Proof.

Given a VERTEX-COVER instance $G = (V, E)$ and k, we construct a SET-COVER instance (U, S, k) that has a set cover of size k iff G has a vertex cover of size k.

Construction

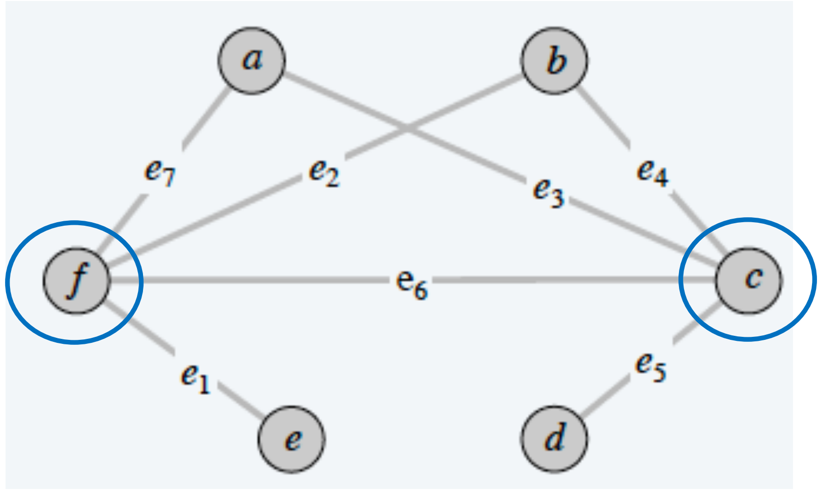
- $U = E$.
- $S_v = \{e \in E : e \text{ incident } to\ v\}$ for each $v \in V$.

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$S_a = \{3, 7\}$ $\quad S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$ $\quad S_d = \{5\}$

$S_e = \{1\}$ $\quad S_f = \{1, 2, 6, 7\}$



SET COVER AND VERTEX COVER

Lemma. (U, S, k) contains a set cover of size k iff $G = (V, E)$ contains a vertex cover of size k.

$\Longrightarrow$

Let $Y \subseteq S$ be a set cover of size k in (U, S, k).
Then $X = \{v : S_v \in Y\}$ is a vertex cover of size k in G.

$\Longleftarrow$

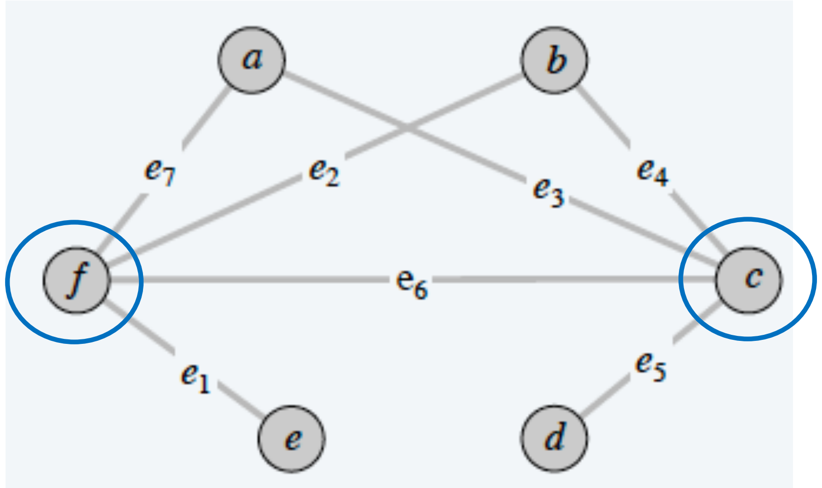
Let $X \subseteq V$ be a vertex cover of size k in G.
Then $Y = \{S_v : v \in X\}$ is a set cover of size k.

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$S_a = \{3, 7\}$ $\quad$ $S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$ $\quad$ $S_d = \{5\}$

$S_e = \{1\}$ $\quad$ $S_f = \{1, 2, 6, 7\}$



INDEPENDENT-SET & VERTEX-COVER

INDEPENDENT-SET.

Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or more) vertices such that no two are adjacent?

VERTEX-COVER.

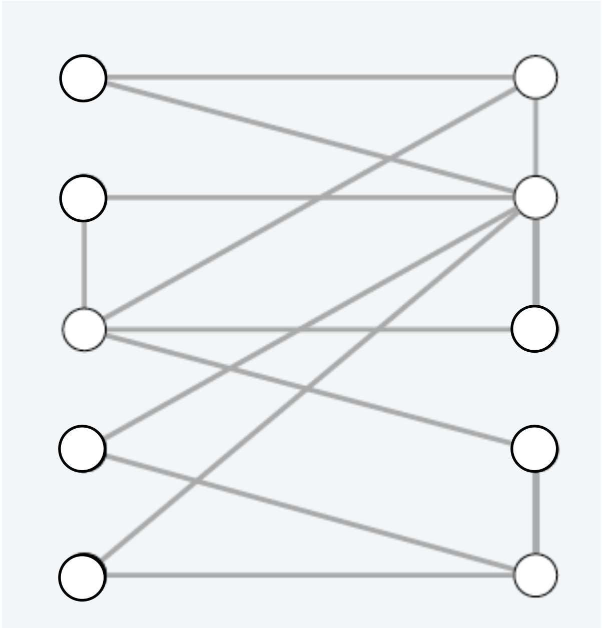
Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Q: Is there an independent set of size ≥ 6 ?

Q: Is there an independent set of size ≥ 7 ?

Q: Is there a vertex cover of size ≤ 4 ?

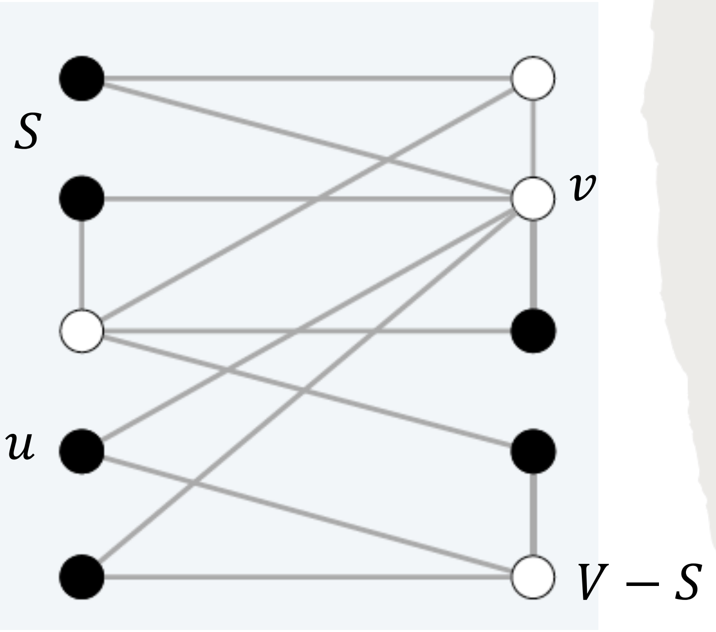
Q: Is there a vertex cover of size ≤ 3 ?



INDEPENDENT-SET & VERTEX-COVER

Theorem. INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.

Proof.



We show S is an independent set of size k if and only if $V - S$ is a vertex cover of size $n - k$.

$\Rightarrow$

- Let S be any independent set of size k.
- $V - S$ is of size $n - k$.
- Consider an arbitrary edge $(u, v) \in E$.
- S is independent
  $\Rightarrow$ either $u \notin S$, or $v \notin S$
  $\Rightarrow$ either $u \in V - S$, or v
- Thus, $V - S$ covers (u, v)

INDEPENDENT-SET $\leq_P$ VERTEX-COVER

INDEPENDENT-SET $\geq_P$ VERTEX-COVER

$\Leftarrow$

- Let $V - S$ be any vertex cover of size $n - k$.
- S is of size k.
- Consider an arbitrary edge $(u, v) \in E$.

or $v \in V - S$, or both.

$\notin S$, or both.

dent set.

Basic reduction strategies:

- Simple equivalence: INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.
- Special case to general case: VERTEX-COVER $\leq_P$ SET-COVER.
- Encoding with gadgets: 3-SAT $\leq_P$ INDEPENDENT-SET.

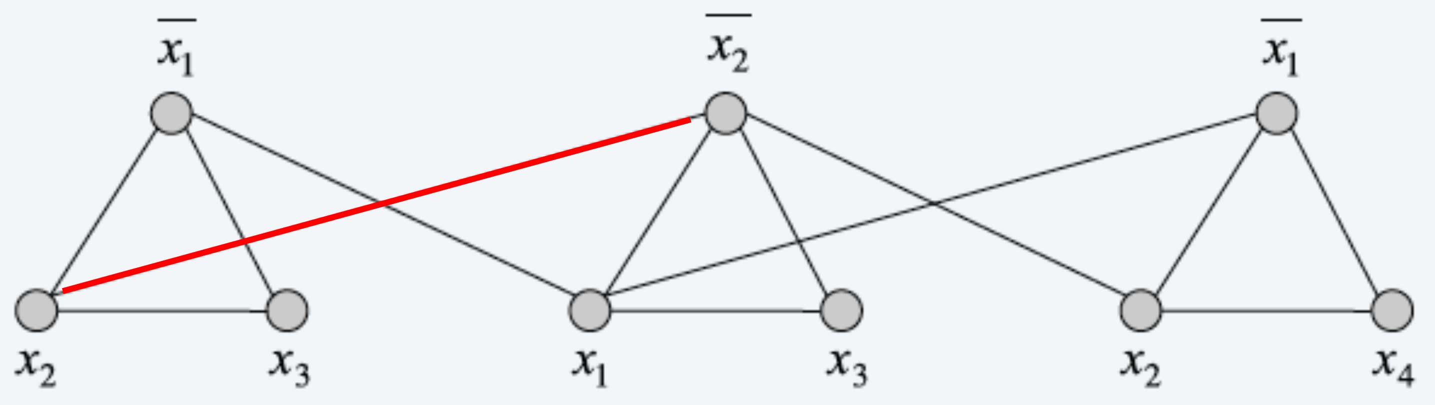
Theorem. 3-SAT $\leq_P$ INDEPENDENT-SET.

Proof.

Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size $k = |\Phi|$ if and only if Φ is satisfiable.

Φ: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

(G, k):



At most one is in IS

At most one of x_i and $\overline{x_i}$ is in IS

Construction

- G contains 3 nodes for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

SAT⟹IS

- Consider any satisfying assignment for Φ.
- Select one true literal from each clause/triangle.
- This is an independent set of size $k = |\Phi|$.

SAT⟸IS

- Let S be independent set of size k.
- S must contain exactly one node in each triangle.
- Set these literals to *true* (and remaining literals consistently).
- All clauses in Φ are satisfied.

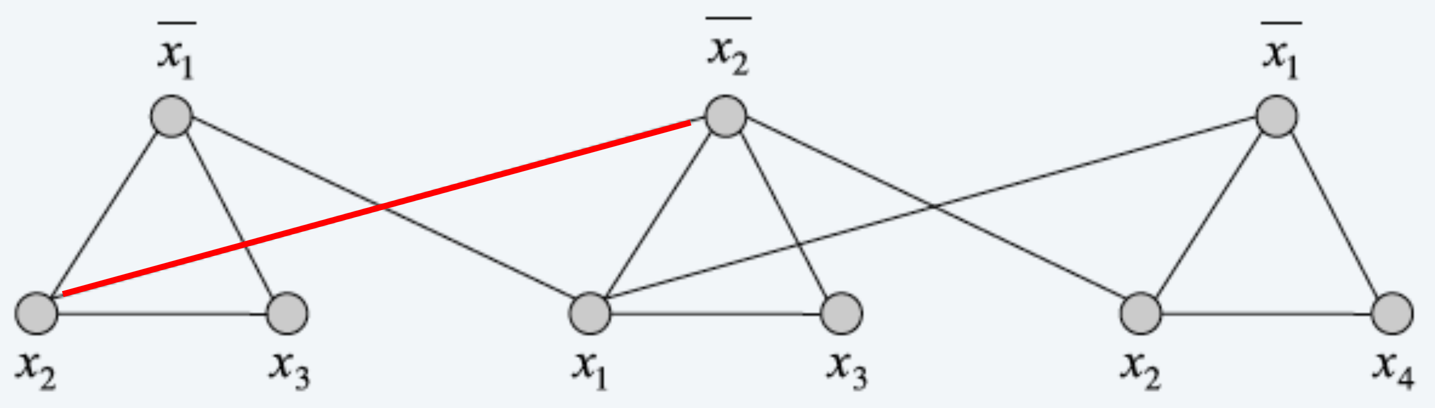
At most one is in IS

At most one of x_i and $\overline{x_i}$ is in IS

31

Φ: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

(G, k):



Construction

- G contains 3 nodes for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

REVIEW

Basic reduction strategies:

- Simple equivalence: INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.
- Special case to general case: VERTEX-COVER $\leq_P$ SET-COVER.
- Encoding with gadgets: 3-SAT $\leq_P$ INDEPENDENT-SET.

Properties (Transitivity):

If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

Proof idea: Compose the two algorithms.

Example: 3-SAT $\leq_P$ INDEPENDENT-SET $\leq_P$ VERTEX-COVER $\leq_P$ SET-COVER.

Decision problem. Does there exist a vertex cover of size $\leq k$?
Search problem. Find a vertex cover of size $\leq k$.
Optimization problem. Find a vertex cover of minimum size.

Three problems poly-time reduce to one another.

- VERTEX-COVER. Does there exist a vertex cover of size $\leq k$?
- FIND-VERTEX-COVER. Find a vertex cover of size $\leq k$.
- FIND-MIN-VERTEX-COVER. Find a vertex cover of minimum size.

Theorem.
VERTEX-COVER $\equiv_P$ FIND-VERTEX-COVER

$\leq_P$
Decision problem is a special case of search problem

$\geq_P$
To find a vertex cover of size $\leq k$:
- Determine if there exists a vertex cover of size $\leq k$.
- Find v s.t. $G - \{v\}$ has a cover of size $\leq k - 1$.
  (any vertex in a vertex cover of size $\leq k$ satisfies).
- Include v in the vertex cover.
- Find a vertex cover of size $\leq k - 1$ in $G - \{v\}$.

Theorem.
FIND-VERTEX-COVER $\equiv_P$ FIND-MIN-VERTEX-COVER

$\leq_P$
Search problem is a special case of optimization problem.

$\geq_P$
To find vertex cover of minimum size:
- Binary search (or linear search) for size k^* of min vertex cover.
- Solve search problem for given k^*.