Intractability: Checking Algorithms

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 11

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
  - All possible paths, all possible spanning trees, all possible subsets of edges, ...

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, . . . have polynomial time algorithms
- Search space for solutions is exponential
  - All possible paths, all possible spanning trees, all possible subsets of edges, . . .
- Brute force: scan exponential possibilities and choose the best

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, . . . have polynomial time algorithms
- Search space for solutions is exponential
  - All possible paths, all possible spanning trees, all possible subsets of edges, . . .
- Brute force: scan exponential possibilities and choose the best
- Do all problems admit such efficient solutions?

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, . . . have polynomial time algorithms
- Search space for solutions is exponential
  - All possible paths, all possible spanning trees, all possible subsets of edges, . . .
- Brute force: scan exponential possibilities and choose the best
- Do all problems admit such efficient solutions?
- Unfortunately not

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, . . . have polynomial time algorithms
- Search space for solutions is exponential
  - All possible paths, all possible spanning trees, all possible subsets of edges, . . .
- Brute force: scan exponential possibilities and choose the best
- Do all problems admit such efficient solutions?
- Unfortunately not
- For a large class of "natural" problems, no shortcut is known to exist

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes
- Student: Given N, find p,q such that $pq = N$
  - Generate a solution

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes
- Student: Given N, find p,q such that $pq = N$
  - Generate a solution
- Teacher: Given a student's solution p,q, verify that $pq = N$
  - Check a solution

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes
- Student: Given N, find p,q such that $pq = N$
  - Generate a solution
- Teacher: Given a student's solution p,q, verify that $pq = N$
  - Check a solution

Checking algorithms

- Checking algorithm C for problem P

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes
- Student: Given N, find p,q such that $pq = N$
  - Generate a solution
- Teacher: Given a student's solution p,q, verify that $pq = N$
  - Check a solution

Checking algorithms

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution "certificate" S for I

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes
- Student: Given N, find p,q such that $pq = N$
  - Generate a solution
- Teacher: Given a student's solution p,q, verify that $pq = N$
  - Check a solution

Checking algorithms

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution "certificate" S for I
- C outputs yes if S represents a valid solution for I, no otherwise

Generating vs checking

- A teacher assigns homework:
  - Factorize a large number that is the product of two primes
- Student: Given N, find p,q such that $pq = N$
  - Generate a solution
- Teacher: Given a student's solution p,q, verify that $pq = N$
  - Check a solution

Checking algorithms

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution "certificate" S for I
- C outputs yes if S represents a valid solution for I, no otherwise
- For factorization
  - I is N
  - S is $\{p, q\}$
  - C involves verifying that $pq = N$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n)$
  - Disjunction of literals (variables, negated variables)

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n)$
  - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n)$
  - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

- Assign suitable values {True,False} to $x_1, x_2, x_3, \ldots$ so that the formula evaluates to True

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n)$
  - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

- Assign suitable values {True,False} to $x_1, x_2, x_3, \ldots$ so that the formula evaluates to True

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

- $x_1 =$ True, $x_2 =$ True, $x_3 =$ False makes this formula evaluate to True

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n)$
  - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

- Assign suitable values {True,False} to $x_1, x_2, x_3, \ldots$ so that the formula evaluates to True

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

- $x_1 = \text{True}$, $x_2 = \text{True}$, $x_3 = \text{False}$ makes this formula evaluate to True

- Add a clause

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge \wedge (x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Boolean operators
  - $\neg x_j$ — negation of x_j
  - $x_i \vee x_j$ — x_i or x_j
  - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n)$
  - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

- Assign suitable values {True,False} to $x_1, x_2, x_3, \ldots$ so that the formula evaluates to True

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

- $x_1 =$ True, $x_2 =$ True, $x_3 =$ False makes this formula evaluate to True
- Add a clause

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge \wedge (x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

- Now there is no satisfying assignment

Boolean satisfiability

- Generating a solution

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known
- Checking a solution
  - Given formula F and valuation $V(x)$ for each x, substitute into formula and evaluate

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known
- Checking a solution
  - Given formula F and valuation $V(x)$ for each x, substitute into formula and evaluate

- Input format is important

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known
- Checking a solution
  - Given formula F and valuation $V(x)$ for each x, substitute into formula and evaluate

- Input format is important
- Suppose a clause is a conjunction of literals ...

$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \ldots \wedge x_k)$$

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known
- Checking a solution
  - Given formula F and valuation $V(x)$ for each x, substitute into formula and evaluate

- Input format is important
- Suppose a clause is a conjunction of literals ...

$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \ldots \wedge x_k)$$

- ...and a formula is a disjunction of clauses $C_1 \vee C_2 \vee \ldots \vee C_m$

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known
- Checking a solution
  - Given formula F and valuation $V(x)$ for each x, substitute into formula and evaluate

- Input format is important
- Suppose a clause is a conjunction of literals ...

$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \ldots \wedge x_k)$$

- ...and a formula is a disjunction of clauses $C_1 \vee C_2 \vee \ldots \vee C_m$
- Each clause forces a unique valuation

Boolean satisfiability

- Generating a solution
  - Try each possible assignment V to $x_1, x_2, \ldots, x_n$
  - n variables — 2^n possible assignments
  - Is there a better algorithm? Not known
- Checking a solution
  - Given formula F and valuation $V(x)$ for each x, substitute into formula and evaluate

- Input format is important
- Suppose a clause is a conjunction of literals ...

$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \ldots \wedge x_k)$$

- ... and a formula is a disjunction of clauses $C_1 \vee C_2 \vee \ldots \vee C_m$
- Each clause forces a unique valuation
- Try each clause in sequence

Travelling salesman

- A network of cities with distances between each pair

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \dots, x$ of minimum cost, visiting all vertices

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \ldots, x$ of minimum cost, visiting all vertices

- Designing a checking algorithm

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \ldots, x$ of minimum cost, visiting all vertices

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \ldots, x$ of minimum cost, visiting all vertices

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \ldots, x$ of minimum cost, visiting all vertices

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \ldots, x$ of minimum cost, visiting all vertices

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle $x, y, z, \ldots, x$ of minimum cost, visiting all vertices

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?
- Transform the problem

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?
- Transform the problem
- Is there a tour with cost at most K?

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?
- Transform the problem
- Is there a tour with cost at most K?
- Now, given a solution S, we can check it

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?

- Transform the problem
- Is there a tour with cost at most K?
- Now, given a solution S, we can check it
- For the original problem, cost is at most the sum of all the edge weights in the graph

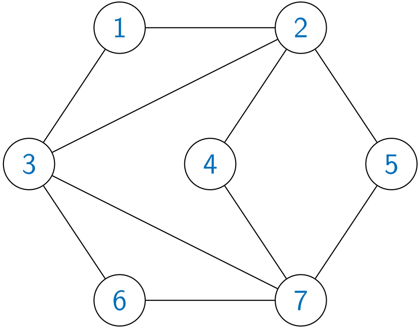
Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
  - Verify that S is a cycle
  - Compute its cost
  - How to check that S is the least cost cycle?

- Transform the problem
- Is there a tour with cost at most K?
- Now, given a solution S, we can check it
- For the original problem, cost is at most the sum of all the edge weights in the graph
- Find optimum K — test different values using binary search

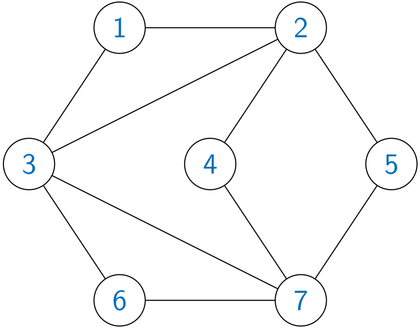
Independent set

- u, v are independent if there is no edge (u, v)



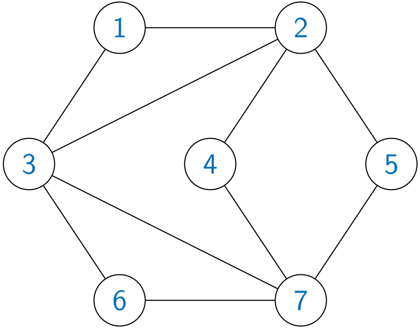
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent



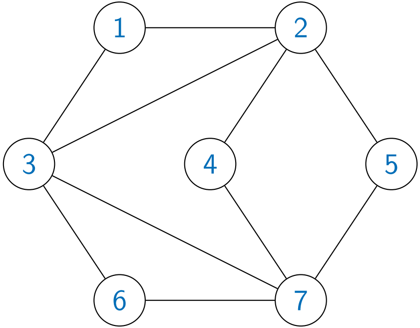
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other



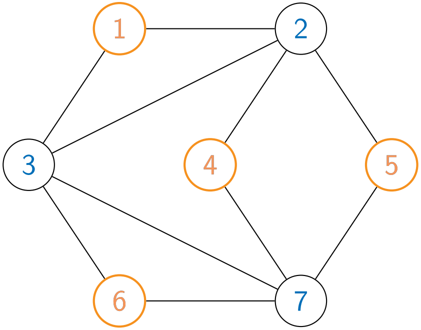
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph



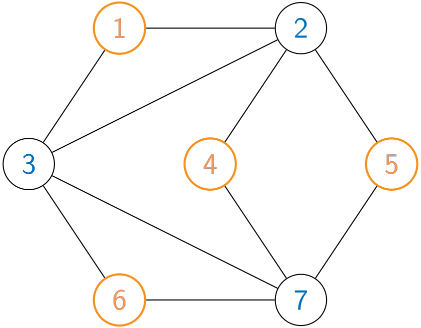
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph



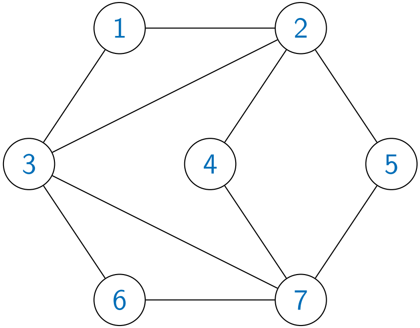
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph
- Checking version: Is there an independent set of size K?



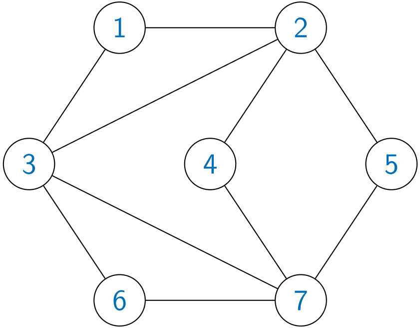
Vertex cover

- Node u covers every edge (u, v) incident on u



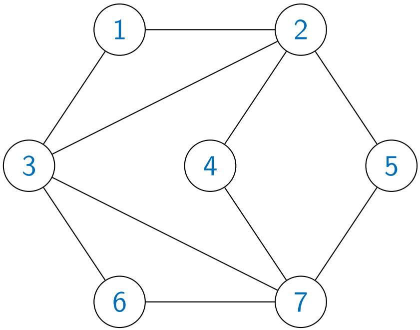
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U



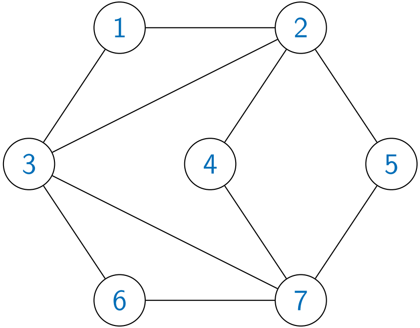
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads



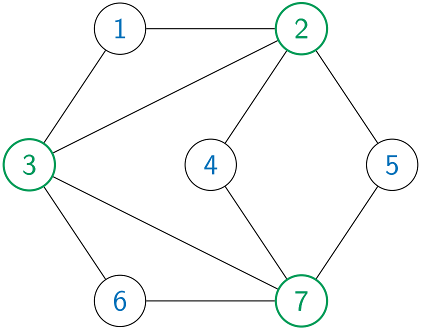
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph



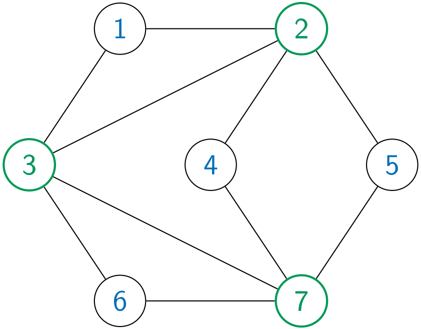
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph



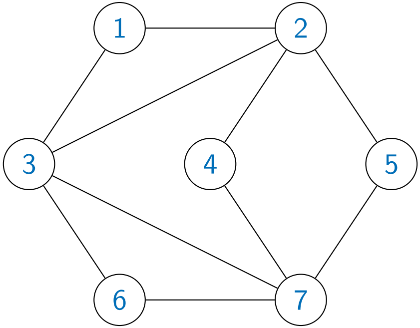
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph
- Checking version: Is there an vertex cover of size K?



Connecting independent set, vertex cover

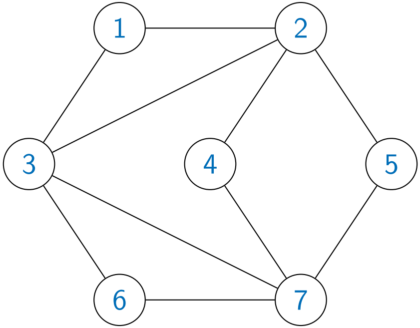
- U is an independent set of size K iff $V \setminus U$ is a vertex cover of size $N - K$



Connecting independent set, vertex cover

- U is an independent set of size K iff $V \setminus U$ is a vertex cover of size $N - K$

($\Rightarrow$) Every edge (u, v) has at most one end point in U, so at least one end point in $V \setminus U$

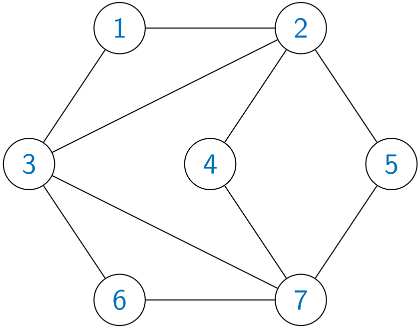


Connecting independent set, vertex cover

- U is an independent set of size K iff $V \setminus U$ is a vertex cover of size $N - K$

($\Rightarrow$) Every edge (u, v) has at most one end point in U, so at least one end point in $V \setminus U$

($\Leftarrow$) For any edge (u, v), at least one endpoint is in $V \setminus U$, so there are no edges (u, v) within U

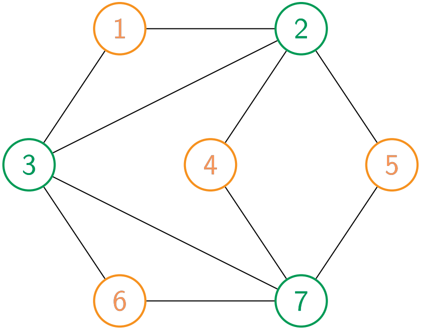


Connecting independent set, vertex cover

- U is an independent set of size K iff $V \setminus U$ is a vertex cover of size $N - K$

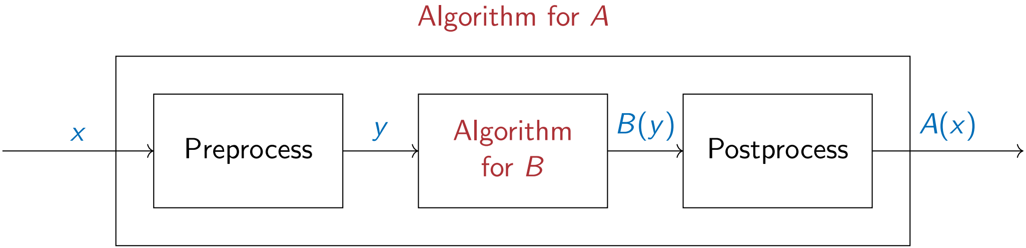
($\Rightarrow$) Every edge (u, v) has at most one end point in U, so at least one end point in $V \setminus U$

($\Leftarrow$) For any edge (u, v), at least one endpoint is in $V \setminus U$, so there are no edges (u, v) within U



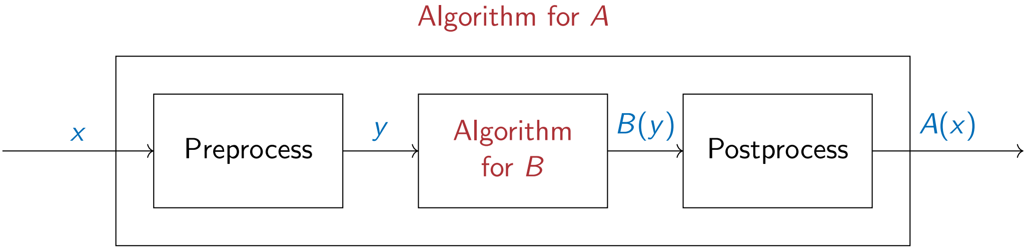
Reductions

- Independent set and vertex cover reduce to each other



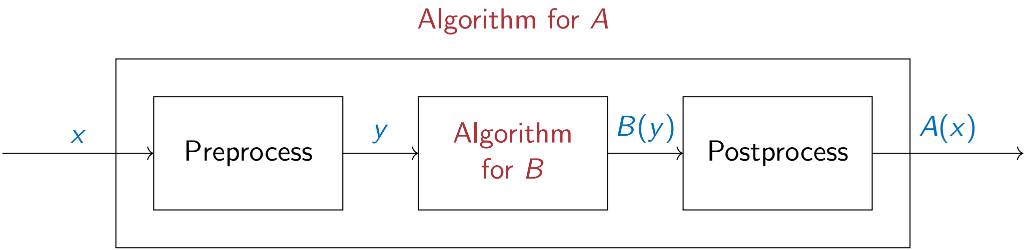
Reductions

- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B



Reductions

- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B
- Many pairs of checkable problems are inter-reducible



Reductions

- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B
- Many pairs of checkable problems are inter-reducible
- All "equally" hard