Design and Analysis of Algorithms
NP-Completeness

Some Slides from: Haidong Xue
Provided by: Dr. Atif Tahir, Waheed Ahmed
Presented by: Farrukh Salim Shaikh

What is polynomial-time?

- Polynomial-time: running time is $O(n^k)$, where k is a constant.
- Are they polynomial-time running time?
  - $T(n) = 3$
    - yes
  - $T(n) = n$
    - yes
  - $T(n) = nlg(n)$
    - yes
  - $T(n) = n^3$
    - yes

What is polynomial-time?

- Are the polynomial-time?
  - $T(n) = 5^n$
    - No
  - $T(n) = n!$
    - No
- Problems with polynomial-time algorithms are considered as tractable
- With polynomial-time, we can define **P** problems, and **NP** problems

What are P and NP?

- P problems
  - (The original definition) Problems that can be solved by **deterministic Turing machine** in polynomial-time.
  - (A equivalent definition) Problems that are solvable in polynomial time.
- NP problems
  - (The original definition) Problems that can be solved by **non-deterministic Turing machine** in polynomial-time.
  - (A equivalent definition) Problems that are **verifiable** in polynomial time.
    - Given a solution, there is a polynomial-time algorithm to tell if this solution is correct.

Why we study NPC?

- One of the most important reasons is:
  - If you see a problem is NPC, you can stop from spending time and energy to develop a fast polynomial-time algorithm to solve it.
- Just tell your boss it is a NPC problem
- How to prove a problem is a NPC problem?
  - Will discuss later ahead.

What if a NPC problem needs to be solved?

- Buy a more expensive machine and wait
  - (could be 1000 years)
- Turn to approximation algorithms
  - Algorithms that produce near optimal solutions

Approximation algorithms for NP-complete problems

Approximation algorithms for NPC problems

- If a problem is NP-complete, there is very likely no polynomial-time algorithm to find an optimal solution
- The idea of approximation algorithms is to develop polynomial-time algorithms to find a near optimal solution

Approximation algorithms for NPC problems

- E.g.: develop a greedy algorithm without proving the greedy choice property and optimal substructure.
- Are those solution found near-optimal?
- How near are they?

Approximation algorithms for NPC problems

- **Approximation ratio** $\boldsymbol{\rho(n)}$
  - Define the cost of the optimal solution as C*
  - The cost of the solution produced by a approximation algorithm is C
  - $\boldsymbol{\rho(n)} \geq max(\frac{C}{C^*}, \frac{C^*}{C})$
- The approximation algorithm is then called a $\boldsymbol{\rho(n)}$-approximation algorithm.

Approximation algorithms for NPC problems

- E.g.:
  - If the total weigh of a MST of graph G is 20
  - A algorithm can produce some spanning trees, and they are not MSTs, but their total weights are always smaller than 25
  - What is the approximation ratio?
    - 25/20 = 1.25
  - This algorithm is called?
    - A 1.25-approximation algorithm

Approximation algorithms for NPC problems

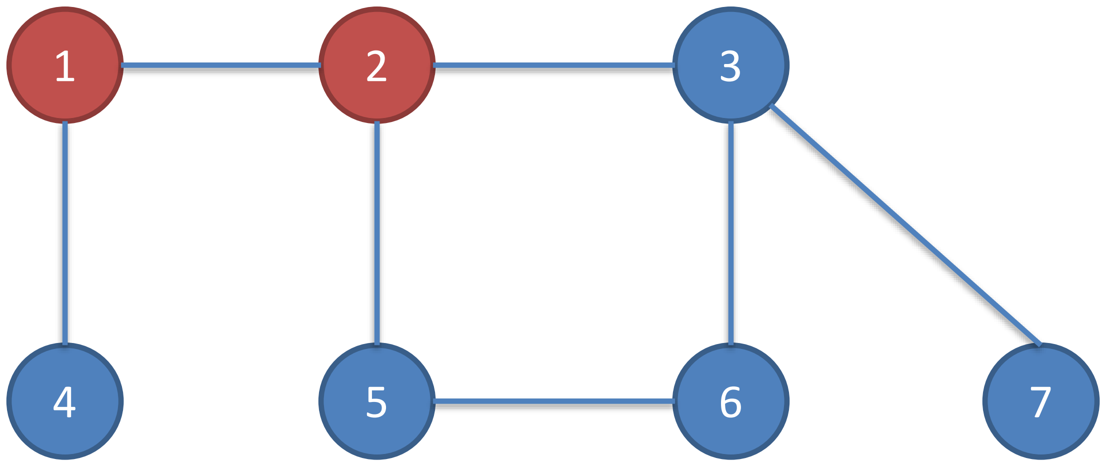
- What if $\boldsymbol{\rho(n)}$=1?
- It is an algorithm that can always find a optimal solution

Vertex-cover problem

Vertex-cover problem and a 2-approximation algorithm

- What is a vertex-cover?
- Given a undirected graph G=(V, E), **vertex-cover** V':
  - $V' \subseteq V$
  - for each edge (u, v) in E, either $u \in V'$ or $v \in V'$ (or both)
- The size of a vertex-cover is |V'|

Vertex-cover problem and a 2-approximation algorithm

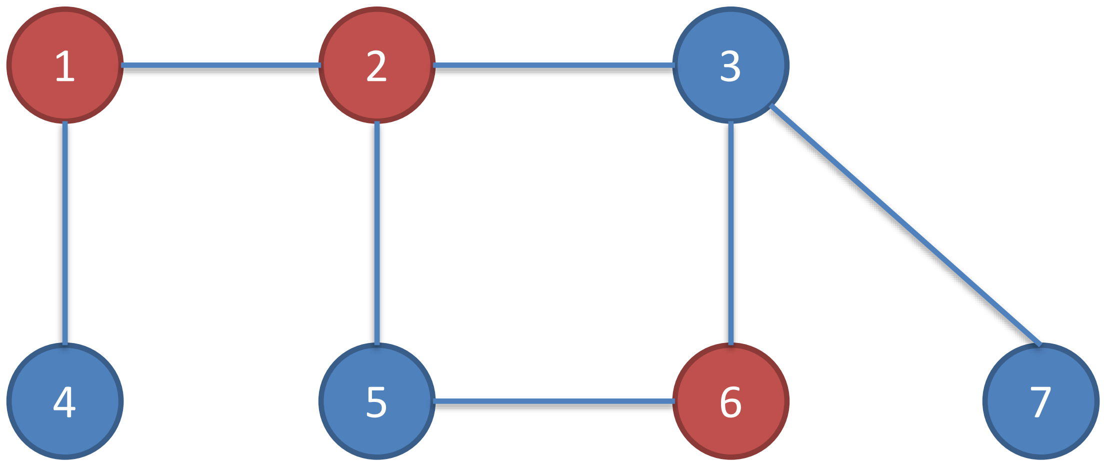


Are the red vertices a vertex-cover?

No. why?

Edges (5, 6), (3, 6) and (3, 7) are not covered by it

Vertex-cover problem and a 2-approximation algorithm

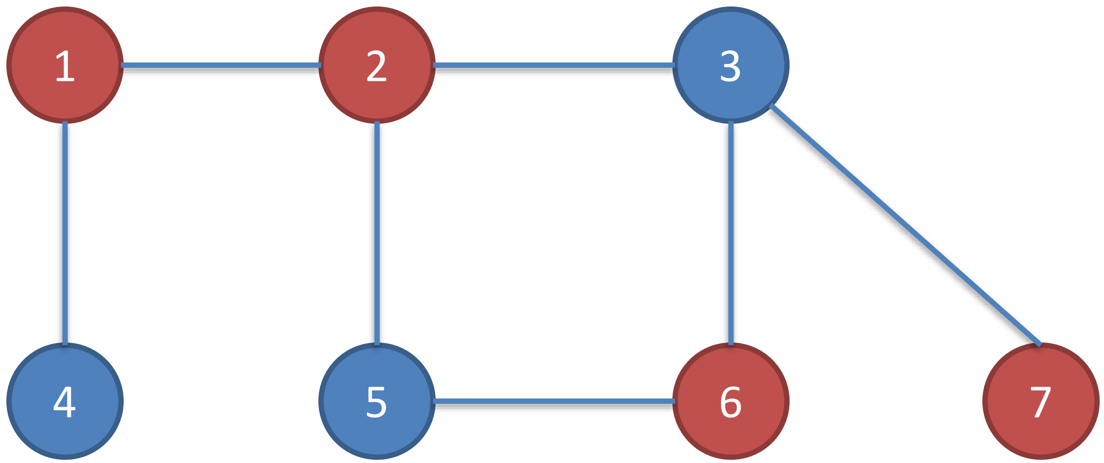


Are the red vertices a vertex-cover?

No. why?

Edge (3, 7) is not covered by it

Vertex-cover problem and a 2-approximation algorithm



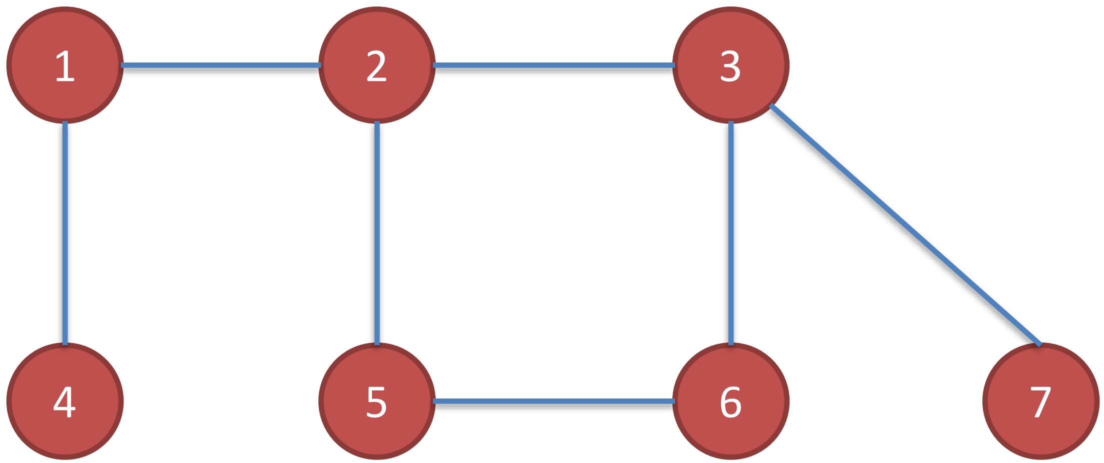
Are the red vertices a vertex-cover?

Yes

What is the size?

4

Vertex-cover problem and a 2-approximation algorithm



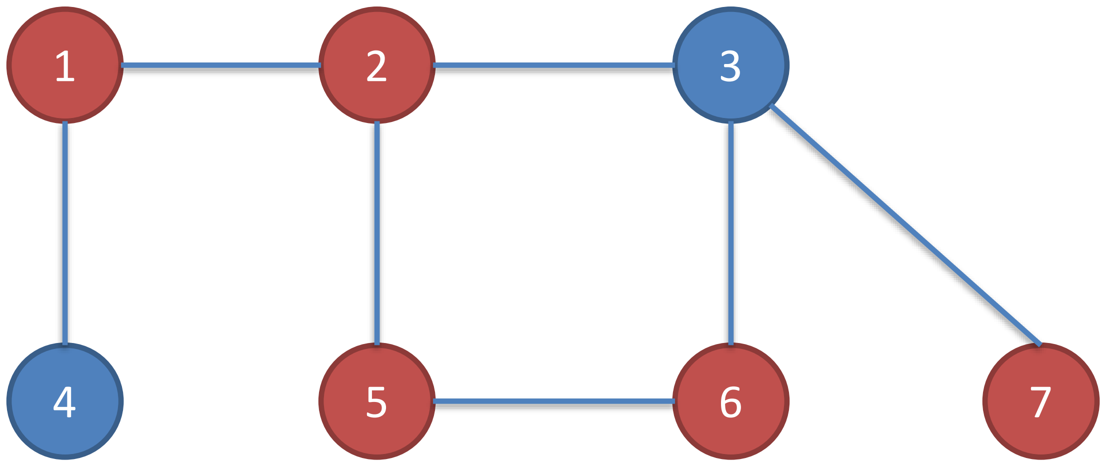
Are the red vertices a vertex-cover?

Yes

What is the size?

7

Vertex-cover problem and a 2-approximation algorithm



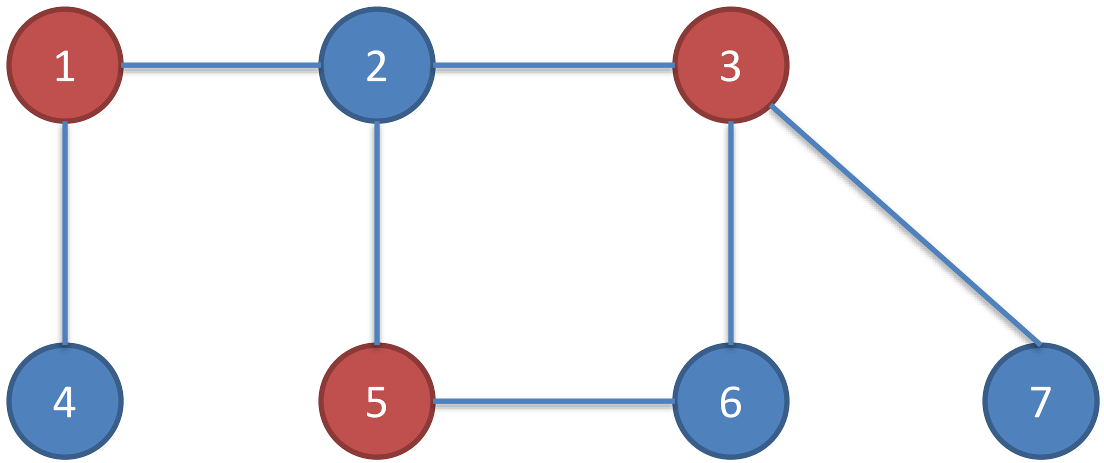
Are the red vertices a vertex-cover?

Yes

What is the size?

5

Vertex-cover problem and a 2-approximation algorithm



Are the red vertices a vertex-cover?

Yes

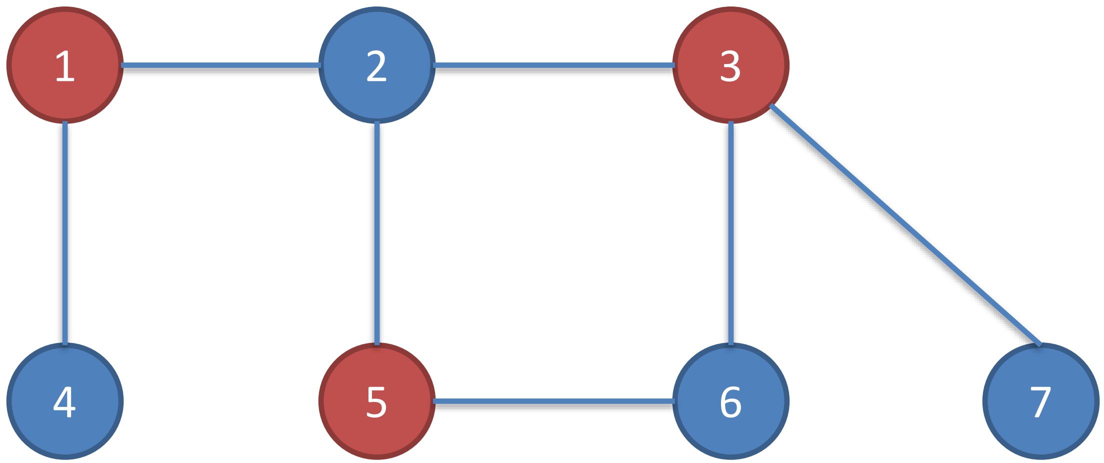
What is the size?

3

Vertex-cover problem and a 2-approximation algorithm

- **Vertex-cover problem**
  - Given a undirected graph, find a vertex cover with minimum size.

Vertex-cover problem and a 2-approximation algorithm



A minimum vertex-cover

Vertex-cover problem and a 2-approximation algorithm

- Vertex-cover problem is **NP-complete**
- A 2-approximation polynomial time algorithm is as the following:
- **APPROX-VERTEX-COVER**(G)

```
C = Ø;
E'=G.E;
while(E' ≠ Ø ){
    Randomly choose a edge (u,v) in E', put u and v into C;
    Remove all the edges that covered by u or v from E'
}
Return C;
```

Vertex-cover problem and a 2-approximation algorithm

APPROX-VERTEX-COVER(G)

$C = \emptyset$;

E'=G.E;

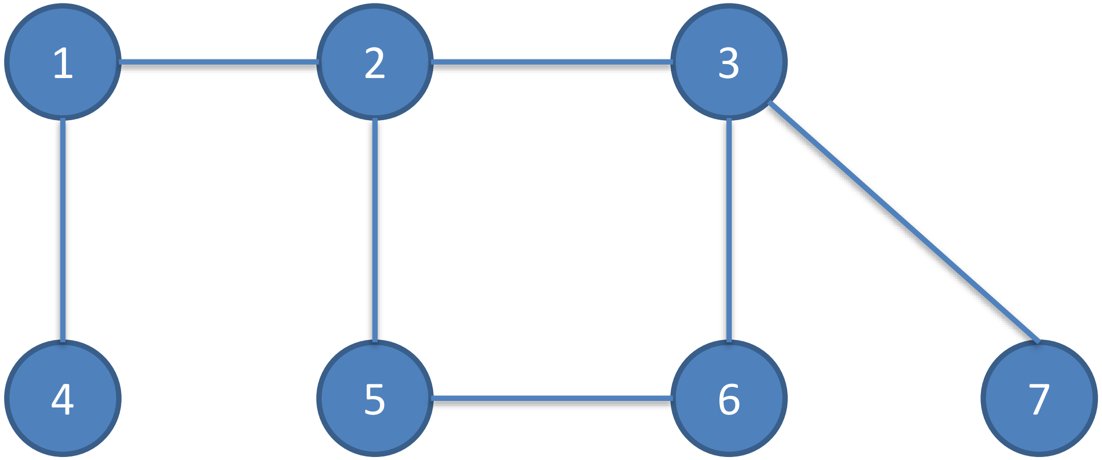
while(E' $\neq \emptyset$){

Randomly choose a edge (u,v) in E', put u and v into C;

Remove all the edges that covered by u or v from E'

}

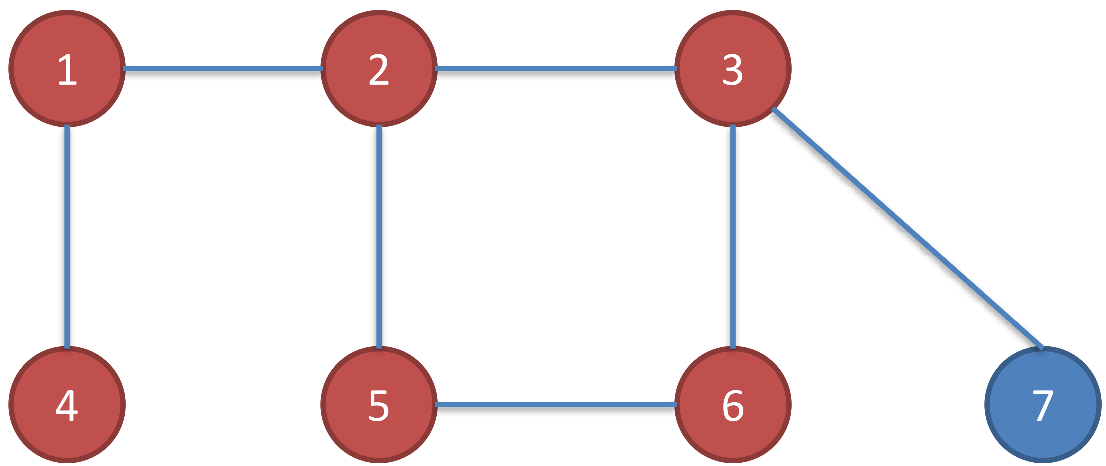
Return C;



Vertex-cover problem and a 2-approximation algorithm

APPROX-VERTEX-COVER(G)

C = ∅;
E'=G.E;
while(E' ≠ ∅){
Randomly choose a edge (u,v) in E', put u and v into C;
Remove all the edges that covered by u or v from E'
}
Return C;



It is then a vertex cover

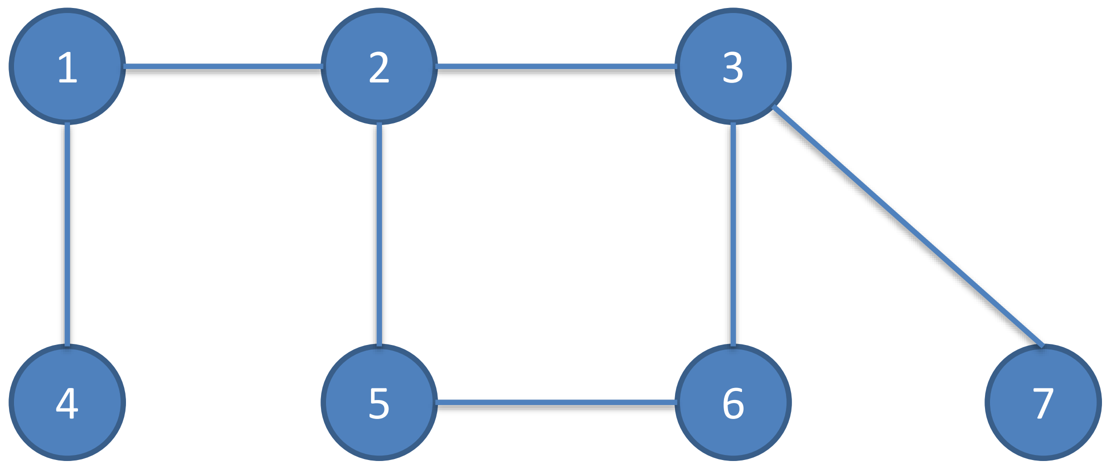
Size? 6

How far from optimal one? Max(6/3, 3/6) = 2

Vertex-cover problem and a 2-approximation algorithm

APPROX-VERTEX-COVER(G)

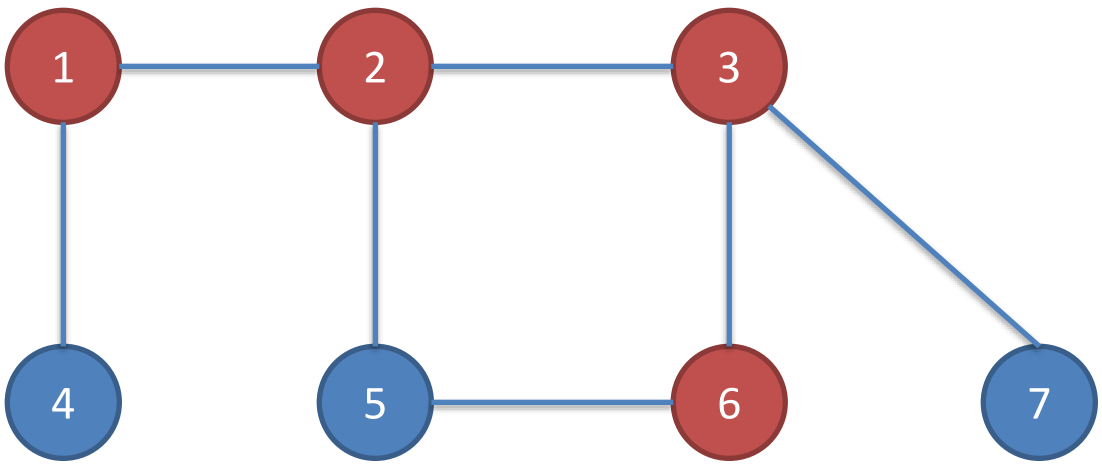
C = ∅;
E'=G.E;
while(E' ≠ ∅){
Randomly choose a edge (u,v) in E', put u and v into C;
Remove all the edges that covered by u or v from E'
}
Return C;



Vertex-cover problem and a 2-approximation algorithm

APPROX-VERTEX-COVER(G)

C = ∅;
E'=G.E;
while(E' ≠ ∅){
  Randomly choose a edge (u,v) in E', put u and v into C;
  Remove all the edges that covered by u or v from E'
}
Return C;

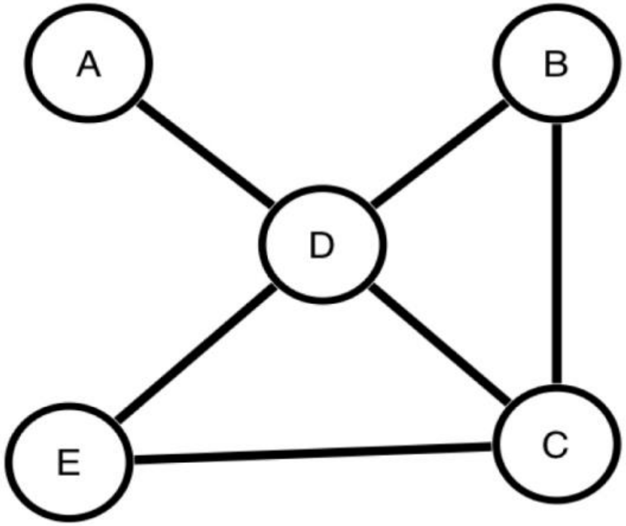


It is then a vertex cover

Size? 4

How far from optimal one? Max(4/3, 3/4) = 1.33

Applications of Vertex Cover Problem (VC)



- Minimum Vertex Cover (MVC) problem comes into play in scheduling problems.
- A scheduling problem can be modeled as a graph, where the vertices represent tasks or times, and an edge between vertices means that a conflict exists between those times or tasks.
- Finding the minimum number of tasks that needs to be removed in order to resolve all conflicts is equivalent to finding a minimum vertex cover.
- [Carruthers, Sarah, Ulrike Stege, and Michael Masson. "Human performance on hard non-Euclidean graph problems: Vertex cover." (2012).]

Vertex-cover problem and a 2-approximation algorithm

- **APPROX-VERTEX-COVER**(G) is a 2-approximation algorithm
- When the size of minimum vertex-cover is *s*
- The vertex-cover produced by **APPROX-VERTEX-COVER** is at most 2*s*

Vertex-cover problem and a 2-approximation algorithm

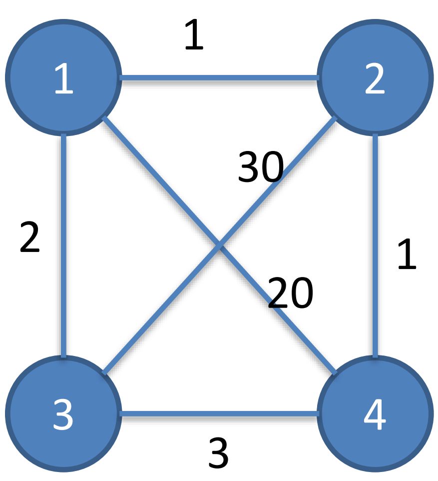
Proof:

- Assume a minimum vertex-cover is U*
- A vertex-cover produced by **APPROX-VERTEX-COVER**(G) is U
- The edges chosen in **APPROX-VERTEX-COVER**(G) is A
- A vertex in U* can only cover 1 edge in A
  - So |U*|>= |A|
- For each edge in A, there are 2 vertices in U
  - So |U|=2|A|
- So |U*|>= |U|/2
- So $\frac{|\mathrm{U}|}{|\mathrm{U}^*|} \leq 2$

Traveling-salesman problem

Traveling-salesman problem

- **Traveling-salesman problem (TSP):**
  - Given a weighted, undirected graph with V >= 3, start from certain vertex, find a **minimum** route visit each vertices once, and return to the original vertex.



Traveling-salesman problem

- TSP is a NP-complete problem
- There is **no polynomial-time approximation** algorithm with a **constant approximation ratio**
- Another strategy to solve NPC problem:
  - **Solve a special case**

Traveling-salesman problem

- **Triangle inequality**:
  - Weight(u, v) <= Weight(u, w) + Weight(w, v)
- E.g.:
  - If all the edges are defined as the distance on a 2D map, the triangle inequality is true
- For the TSPs where the triangle inequality is true:
  - There is a 2-approximation polynomial time algorithm

Metric TSP

- Metric TSP is an special case of the TSP that satisfies the triangle inequality.
- Each vertex should be connected with every other vertex.

Traveling-salesman problem

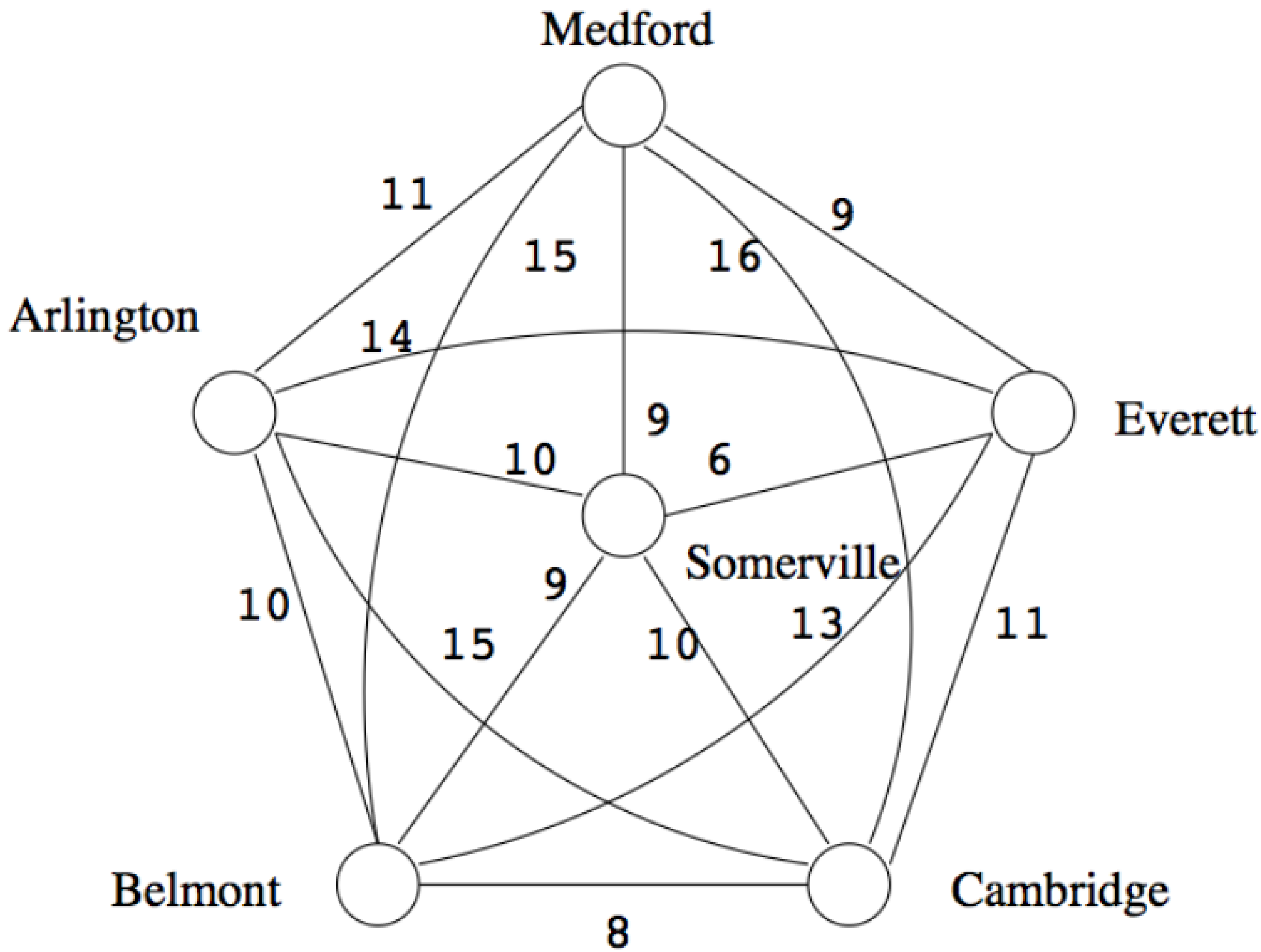
APPROX-TSP-TOUR(G)

Minimum Spanning Tree;

Creating a Cycle;

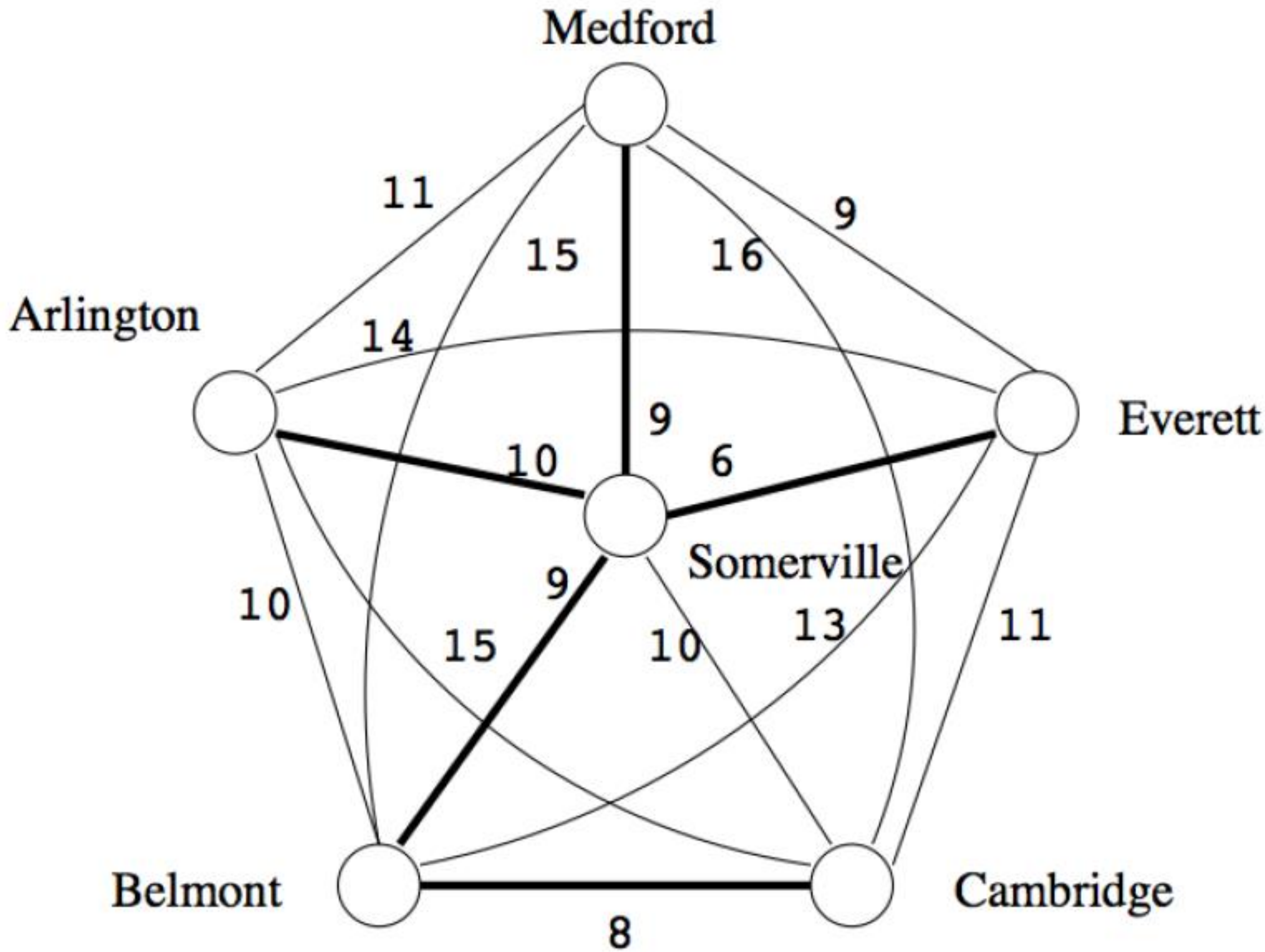
Removing Redundant Visits;

Metric TSP

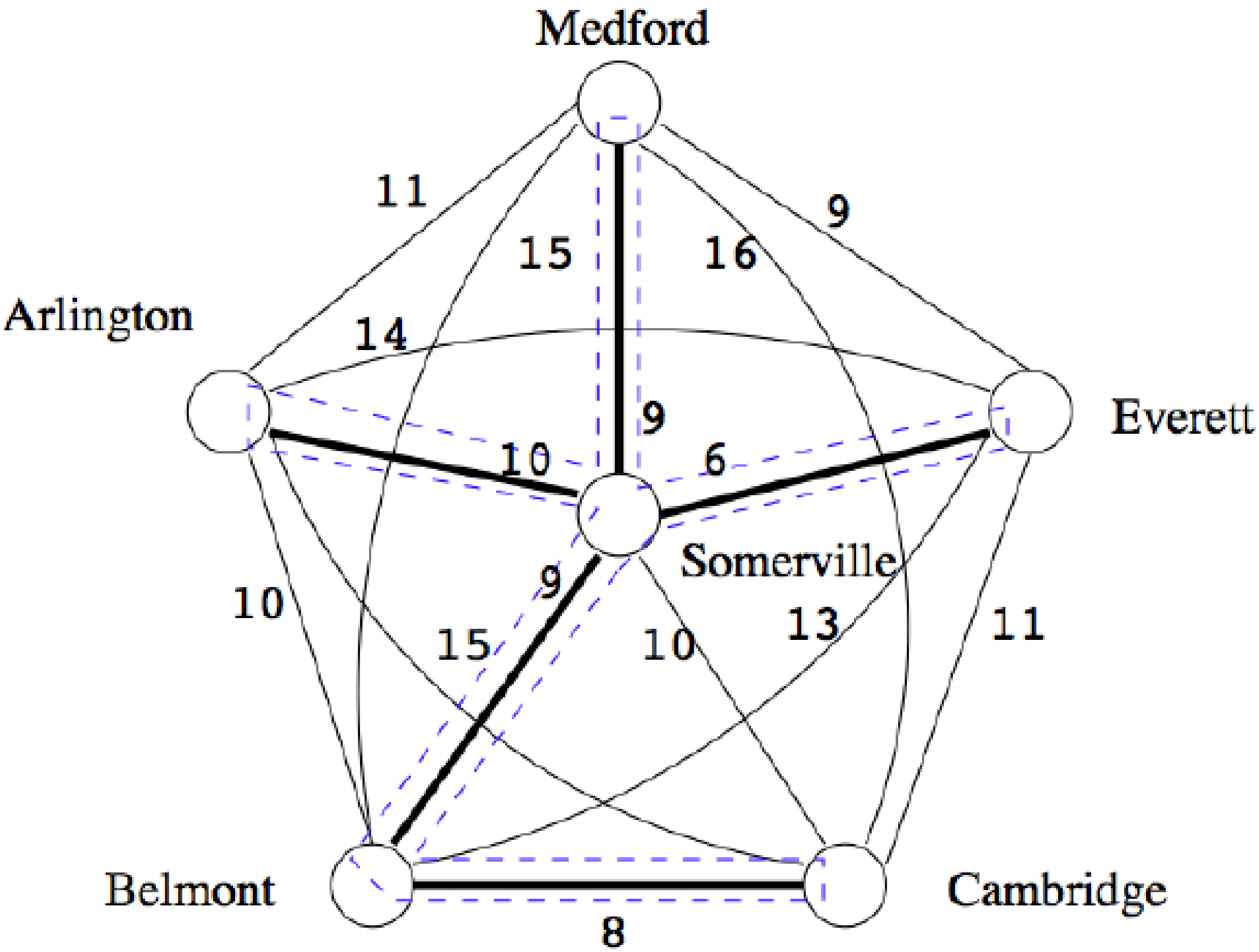


Medford
Arlington
Everett
Somerville
Belmont
Cambridge
11
9
15
16
14
9
10
6
10
9
15
10
13
11
8

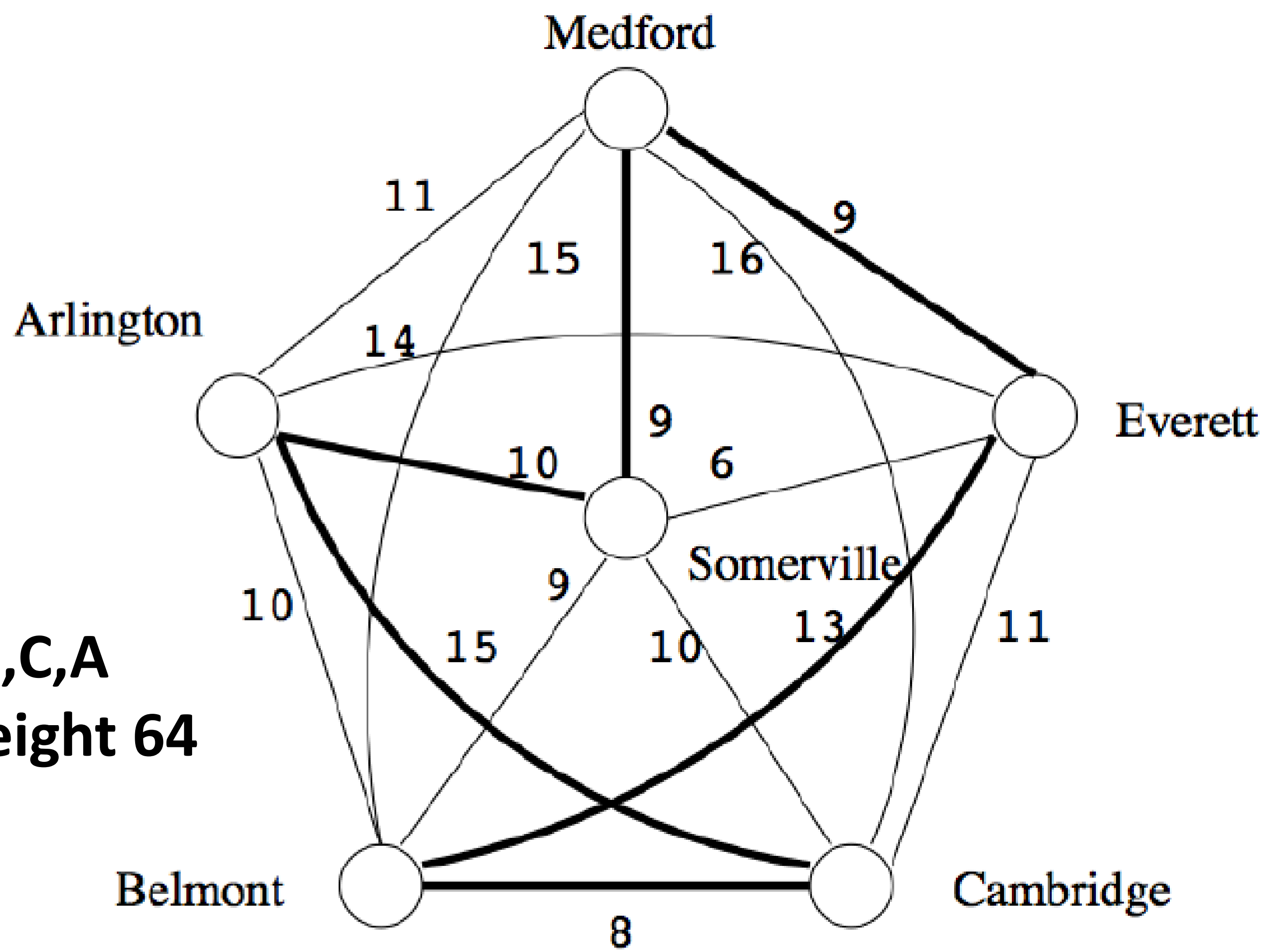
Step 1: MST


Medford
Arlington
Everett
Somerville
Belmont
Cambridge
11
9
15
16
14
9
10
6
10
9
15
10
13
11
8

Step 2: Creating a cycle - using DFS



Step 3: Removing Redundant Visits of DFS



S,M,E,B,C,A with weight 64

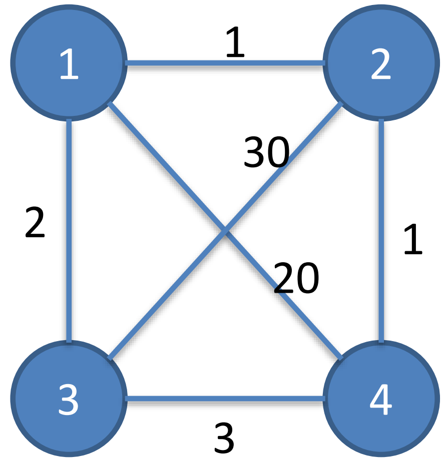
2 Approximation Algorithm

- **Claim:** The weight of the MST M is less than OPT, the weight of the TSP solution T.
  - i.e. OPT = w(T)
- Take T and remove an edge e. T is now a spanning tree.
- Because M is the MST,
  - $w(M) \leq w(T - e)$
  - $w(M) \leq w(T) - w(e)$
  - $w(M) \leq OPT - w(e)$
- Therefore, $w(M) < OPT$.
- Because each edge is used exactly twice during a depth first search on a tree (once descending, once ascending)
- $w(W) = 2 * w(M) < 2 * OPT$.

≤

Traveling-salesman problem

Can we apply the approximation algorithm on this one?



No. The triangle inequality is violated.

Traveling-salesman problem (Method 2)

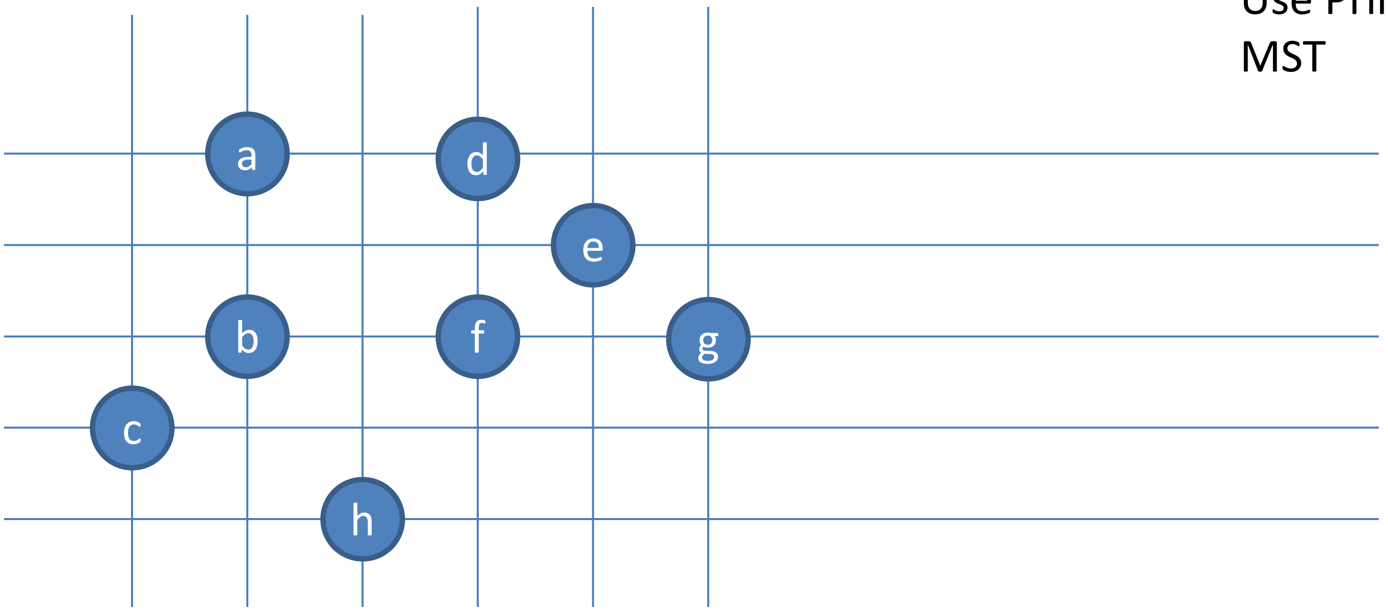
APPROX-TSP-TOUR(G)

Find a MST m;

Choose a vertex as root r;

return preorderTreeWalk(m, r);

Traveling-salesman problem (Method 2)

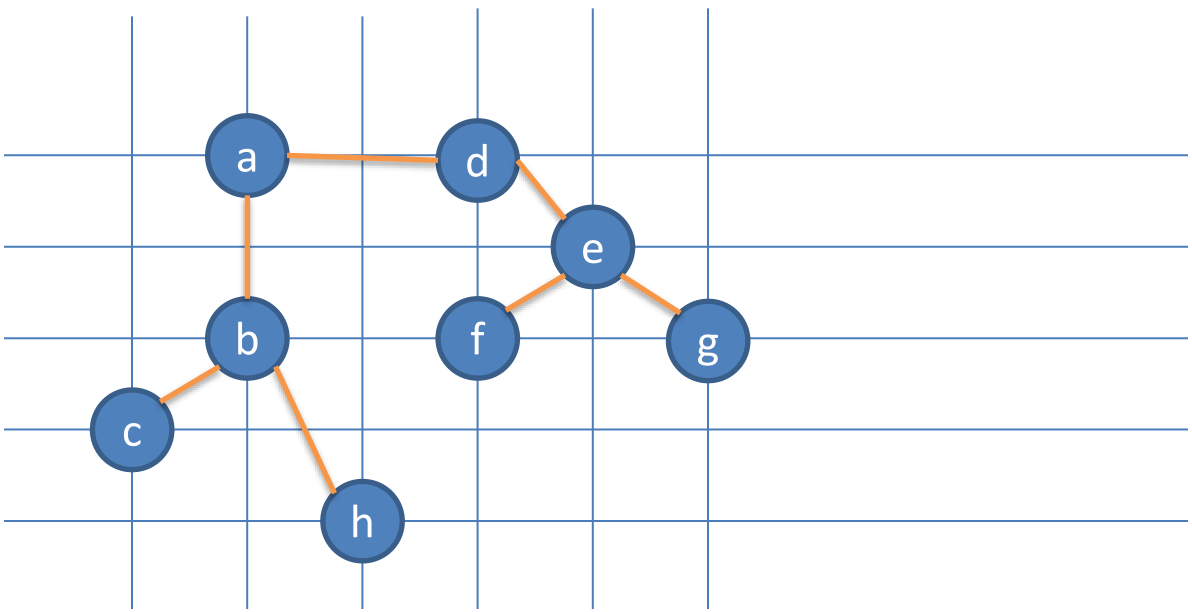


Use Prim's algorithm to get a MST

For any pair of vertices, there is a edge and the weight is the Euclidean distance

Triangle inequality is true, we can apply the approximation algorithm

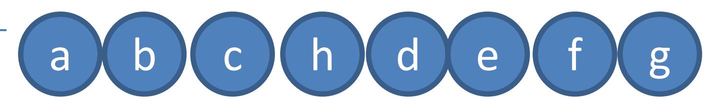
Traveling-salesman problem (Method 2)



Use Prim's algorithm to get a MST

Choose "a" as the root

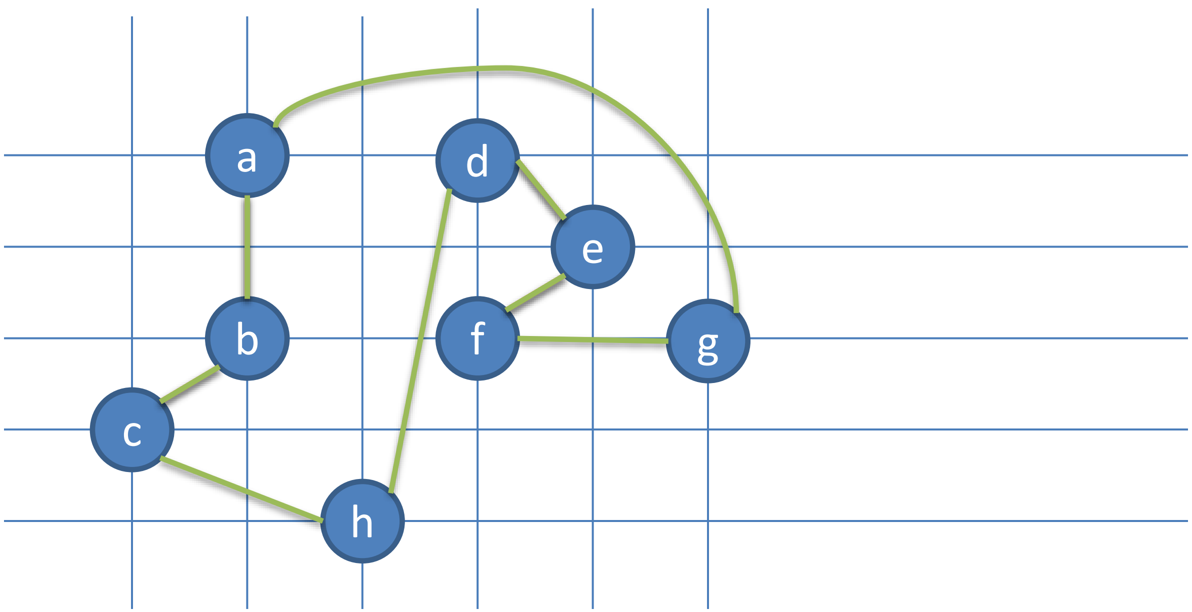
Preorder tree walk



For any pair of vertices, there is a edge and the weight is the Euclidean distance

Triangle inequality is true, we can apply the approximation algorithm

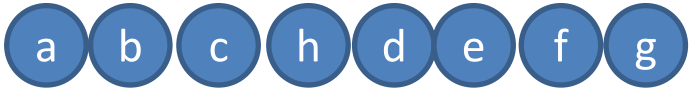
Traveling-salesman problem (Method 2)



Use Prim's algorithm to get a MST

Choose "a" as the root

Preorder tree walk



The route is then…

Because it is a 2-approximation algorithm

A TSP solution is found, and the total weight is at most twice as much as the optimal one

The set covering problem

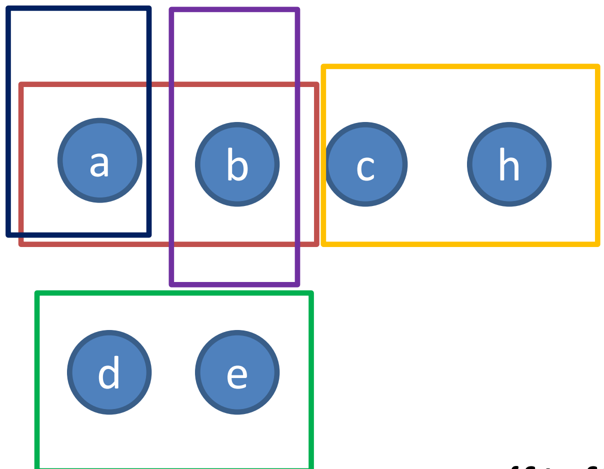
The set-covering problem

Set-covering problem

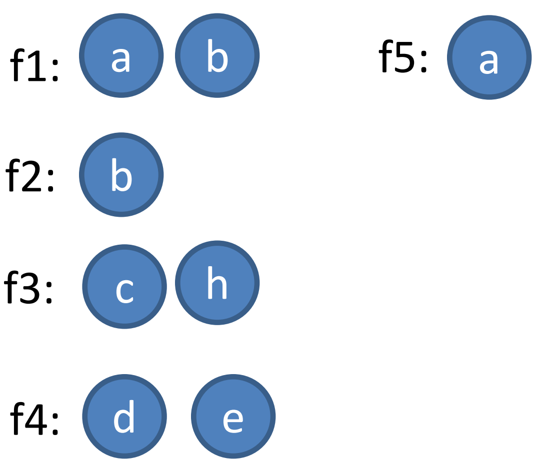
- Given a set X, and a family F of subsets of X, where F covers X, i.e. $X = \bigcup_{S \in F} S$.
- Find a subset of F that covers X and with minimum size

The set-covering problem

X:



F:



{f1, f3, f4} is a subset of F covering X

{f1, f2, f3, f4} is a subset of F covering X

{f2, f3, f4, f5} is a subset of F covering X

Here, {f1, f3, f4} is a minimum cover set

The set-covering problem

- **Set-covering problem** is NP-complete.
- If the size of the largest set in F is m, there is a $\sum_{i=1}^{m} 1/i$ - approximation polynomial time algorithm to solve it.

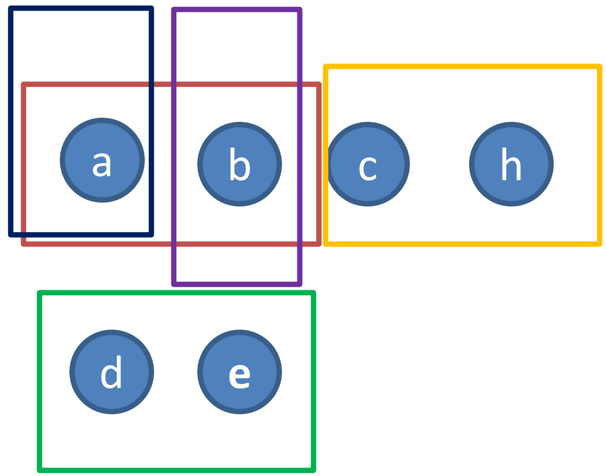
The set-covering problem

GREEDY-SET-COVER(X, F)

```
U=X;
C=∅;
While(U≠ ∅){
    Select S∈F that maximizes |S∩U|;
    U=U-S;
    C=C∪{S};
}
return C;
```

The set-covering problem

X:



We can choose from f1, f3 and f4

Choose f1

We can choose from f3 and f4

Choose f3

We can choose from f4

Choose f4

F:

f1: a b

f2: b

f3: c h

f4: d e

f5: a

U: a b c h d e

C:

f1: a b

f3: c h

f4: d e

Set Cover and its generalizations and variants are fundamental problems with numerous applications. Examples include:

- selecting a small number of nodes in a network to store a file so that all nodes have a nearby copy,
- selecting a small number of sentences to be uttered to tune all features in a speech-recognition model [11],
- selecting a small number of telescope snapshots to be taken to capture light from all galaxies in the night sky,
- finding a short string having each string in a given set as a contiguous sub-string.

What exactly NP is?

Revision

What is polynomial-time?

- Polynomial-time: running time is $O(n^{k})$, where k is a constant. $(Also\ written\ as\text{:}\ n^{O(1)})$
- Are they polynomial-time running time?
  - $T(n) = 3$
    - yes
  - $T(n) = n$
    - yes
  - $T(n) = nlg(n)$
    - yes
  - $T(n) = n^3$
    - yes

Revision

What is polynomial-time?

- Are the polynomial-time?
  - $T(n) = 5^n$
    - No
  - $T(n) = n!$
    - No
- Problems with polynomial-time algorithms are considered as tractable
- With polynomial-time, we can define **P** problems, and **NP** problems

Revision

What are P and NP?

- P problems
  - (The original definition) Problems that can be solved by **deterministic Turing machine** in polynomial-time.
  - (A equivalent definition) Problems that are solvable in polynomial time
- Examples
  - A set of decision problem with yes/no answer
  - Calculating the greatest common divisor
  - sorting n numbers in ascending or descending order
  - search the element in the list

What are P and NP?

Revision

- NP problems
  - (The original definition) Problems that can be solved by **non-deterministic Turing machine** in polynomial-time.
  - (A equivalent definition) Problems that are **verifiable** in polynomial time.
    - Given a solution, there is a polynomial-time algorithm to tell if this solution is correct
- Examples

(i) Integer Factorization

(ii) Graph Isomorphism

What are P and NP?

- Polynomial-time verification can be used to easily tell if a problem is a NP problem
- E.g.:
  - Sorting, n-integers
    - A candidate: an array
    - Verification: scan it once
  - Max-heapify, n-nodes:
    - A candidate: a complete binary search tree
    - Verification: scan all the nodes once
  - Find all the sub sets of a given set A, |A|=n
    - A candidate: a set of set
    - Verification: check each set

NP Problems

- NP = {Decision problems solvable in polynomial time via a “lucky” algorithm}. **The “lucky” algorithm can make lucky guesses always “right” without trying all options.**
  - Stage 1 Non-deterministic model (Guessing): algorithm makes guesses.
  - Stage 2: Deterministic (“verification”) says YES or NO () guesses guaranteed to lead to YES outcome if possible (no otherwise).

NP Problems: verification stage is polynomial

What are P and NP?

- Based on the definition of P and NP, which statements are correct?
  - "NP means non-polynomial"
    - No!
  - $\mathrm{P} \cap \mathrm{NP} = \emptyset$
    - No. $\mathrm{P} \subseteq \mathrm{NP}$
  - A P problem is also a NP problem
    - Yes. $\mathrm{P} \subseteq \mathrm{NP}$

What are P and NP?

- any problem solvable by a deterministic Turing machine in polynomial time is also solvable by a nondeterministic Turing machine in polynomial time. Thus, **P** $\subseteq$ **NP**

P = NP means whether an NP problem can belong to class P problem. In other words whether every problem whose solution can be verified by a computer in polynomial time can also be solved by a computer in polynomial time

What are P and NP?

In order to prove that **P ≠ NP**, we would need to prove that there exists a set of problems *X* such that:

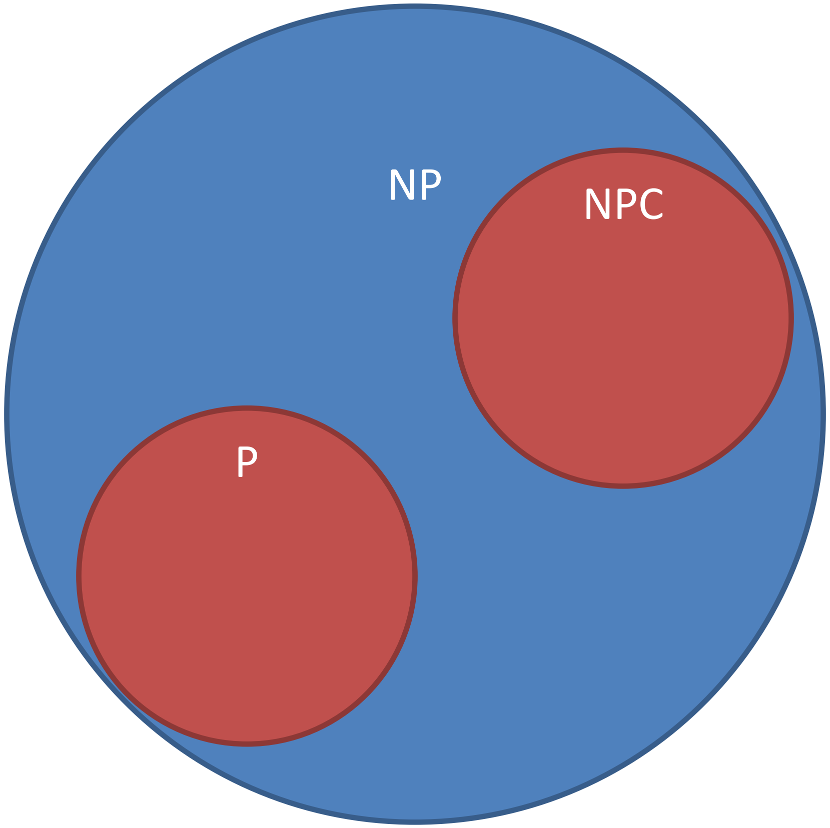
- *X* falls in **NP**. There exists an algorithm with which a nondeterministic Turing machine could solve problems in *X* in polynomial time
- *X* does *not* fall in **P**. There exists *no algorithm whatsoever* with which a deterministic Turing machine could solve problems in *X* in polynomial time

What are NP-complete problems?

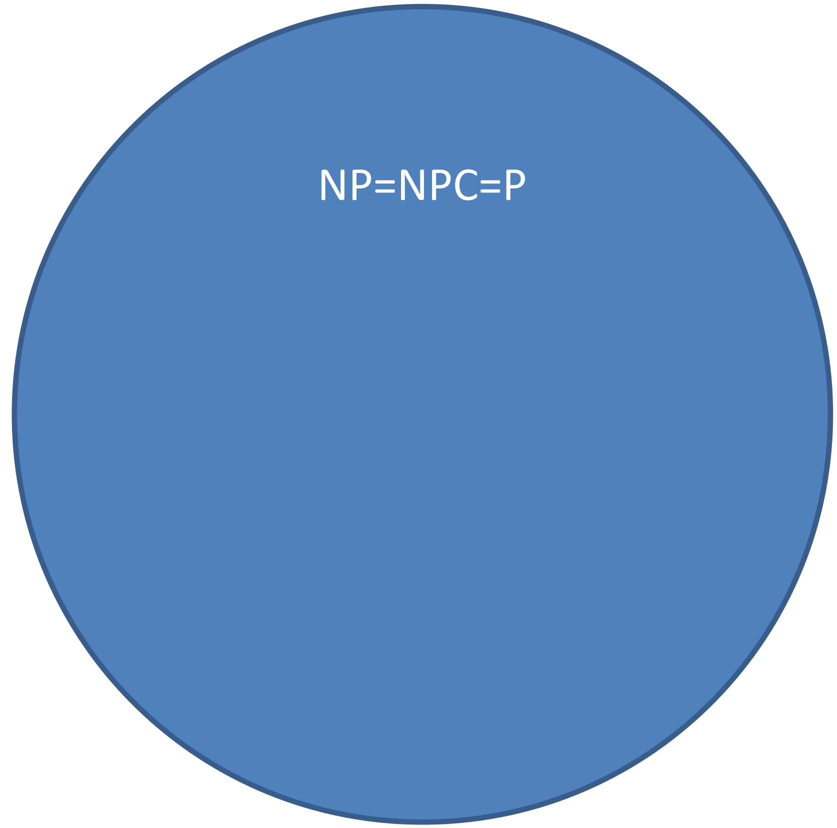
- A NP-complete problem(NPC) is
  - a NP problem
  - harder than all equal to all NP problems
- In other words, NPC problems are the hardest NP problems
- **So far**, no polynomial time algorithms are found for any of NPC problems

What are NP-complete problems?

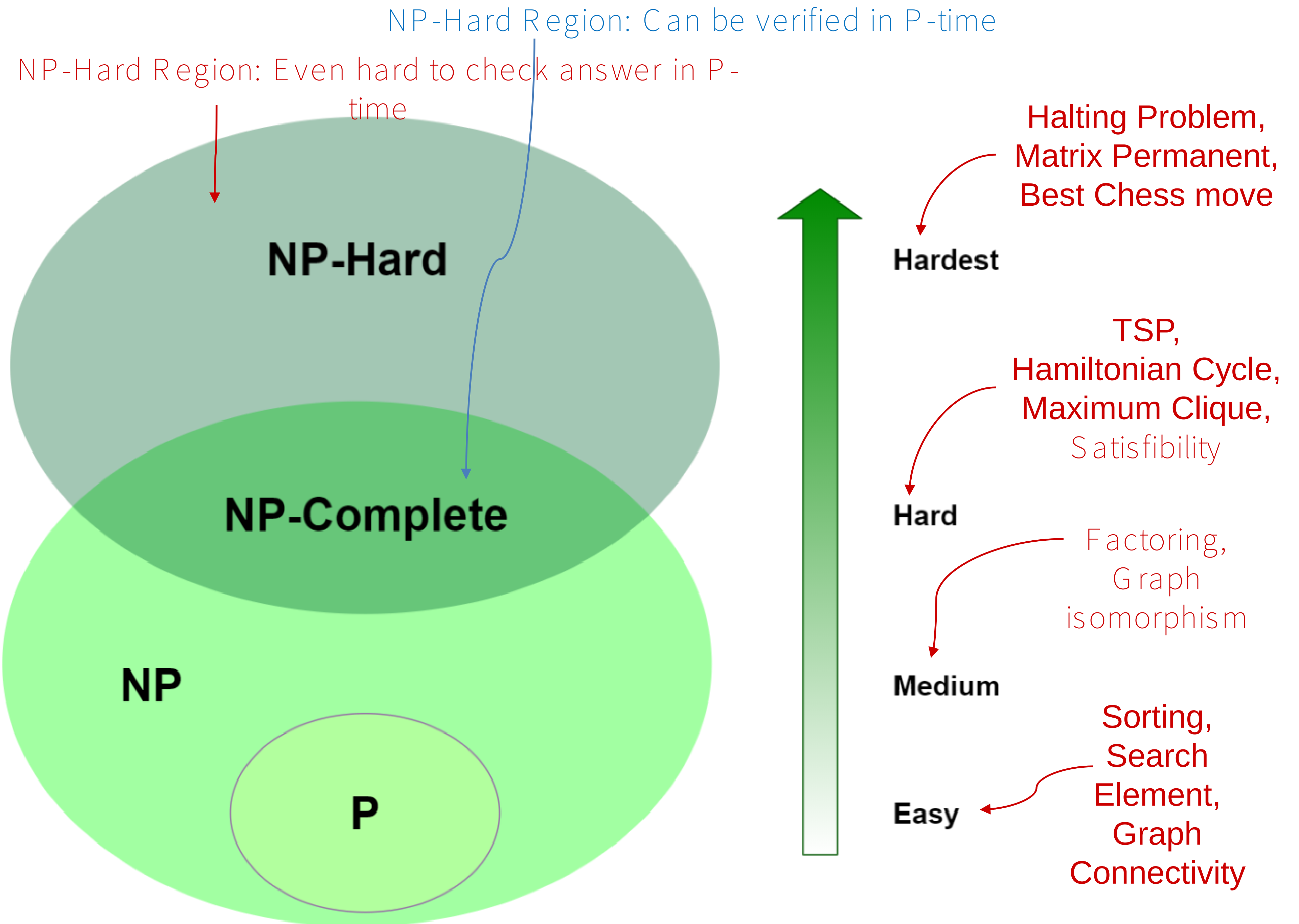
Most scientists believe:



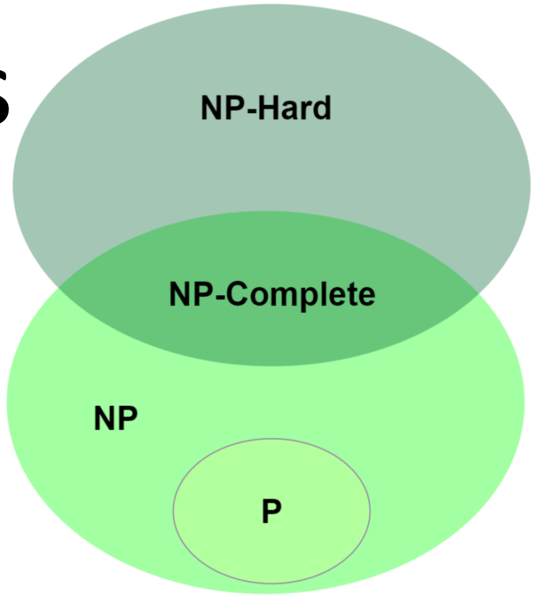
However, it is not ruled out that:



NP-Hard problems: problems harder than or equal to NPC problems


NP-Hard Region: Can be verified in P-time
NP-Hard Region: Even hard to check answer in P-time
NP-Hard
NP-Complete
NP
P
Hardest
Hard
Medium
Easy
Halting Problem, Matrix Permanent, Best Chess move
TSP, Hamiltonian Cycle, Maximum Clique, Satisfibility
Factoring, Graph isomorphism
Sorting, Search Element, Graph Connectivity

NP-Completeness



- A problem B is **NP-complete** if:

  (1) B $\in$ **NP**

  (2) A $\leq_p$ B for all A $\in$ **NP**
- If B satisfies only property (2) we say that B is **NP-hard**
- No polynomial time algorithm has been discovered for an **NP-Complete** problem
- No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem
- if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution.

Examples

Hamiltonian Paths

Optimization Problem: Given a graph, find a path that passes through every vertex exactly once

Decision Problem: Does a given graph have a Hamiltonian Path ?

Traveling Salesman

Optimization Problem: Find a minimum weight Hamiltonian Path

Decision Problem: Given a graph and an integer k, is there a Hamiltonian Path with a total weight at most k ?

Factoring and Graph isomorphism: are NP Problems which means we can verify solution in P time. But these problems still can not reduce to any existing NP-complete problem (so these problems are not hard as NP-complete problems) . Therefore may be in the computational complexity class NP-intermediate.

NP-Hard

- An example of an NP-hard problem is Travelling Salesman Problem (TSP)
  - A special case of TSP, i.e. Metric TSP happens to be NP-complete.

- There are decision problems that are NP-hard but not NP-complete such as the halting problem. That is the problem which asks "given a program and its input, will it run forever?“
  - The halting problem is not in NP since all problems in NP are decidable in a finite number of operations, but the halting problem, in general, is undecidable.

[https://en.wikipedia.org/wiki/NP-hardness]

Why we study NPC?

- For new computational problems we encounter:

1. We try for a while to develop an efficient algorithm; and if this fails,
2. We then try to prove it NP-complete.
3. Find Approximate Algorithms (sub-optimal solution) if required.

- How to prove a problem is a NPC problem?

How to prove a problem is a NPC problem?

- A common method is to prove that it is not easier than a known NPC problem.
- To prove problem A is a NPC problem
  - Choose a NPC problem B
  - Develop a **polynomial-time algorithm translate A to B**
- A **reduction algorithm**
- If A can be solved in polynomial time, then B can be solved in polynomial time. It is contradicted with B is a NPC problem.
- So, A cannot be solved in polynomial time, it is also a NPC problem.

Revision

What if a NPC problem needs to be solved?

- Buy a more expensive machine and wait
  - (could be 1000 years)
- Turn to approximation algorithms
  - Algorithms that produce near optimal solutions

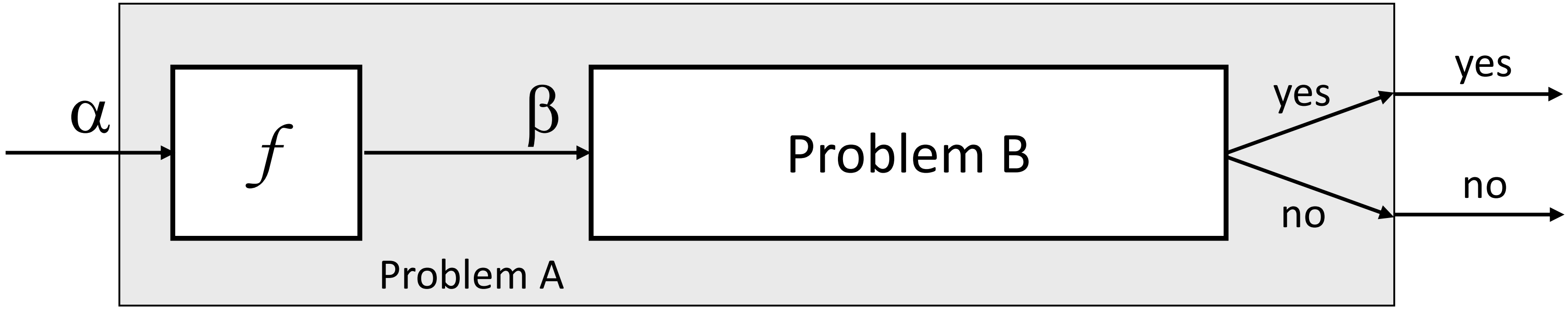
NP-Completeness Reduction

Reduction

- Reduction is a way of saying that one problem is **"easier"** than another.
- We say that problem A is easier than problem B,

  (i.e., we write **"A ≤ B"**)

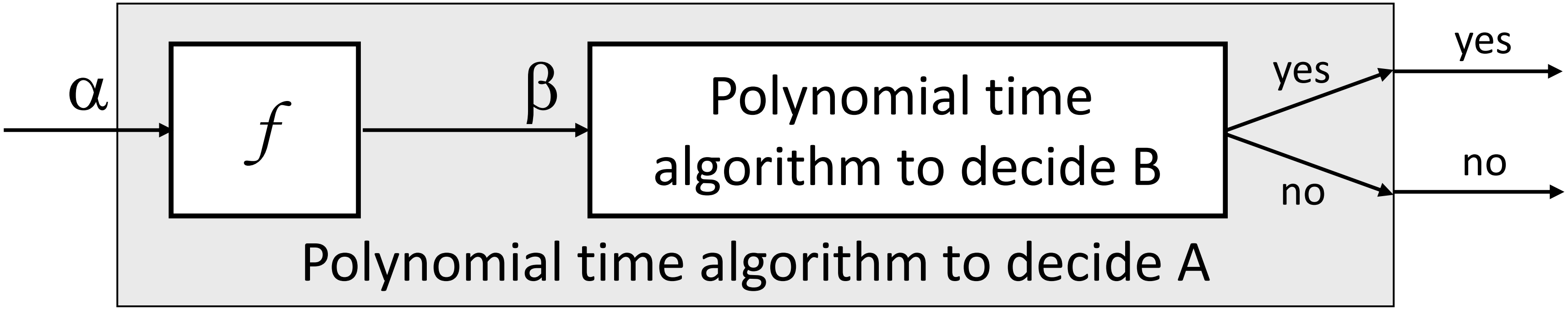
  if we can solve A using the algorithm that solves B.
- **Idea:** transform the inputs of A to inputs of B



Polynomial Reduction

Given two problems A, B, we say that A is polynomially **reducible** to B ($A \leq_p B$) if:

There exists a function f that converts the input of A to inputs of B in polynomial time



1. Use a **polynomial time** reduction algorithm to transform A into B
2. Run a known **polynomial time** algorithm for B
3. Use the answer for B as the answer for A

Not part of course

Reduction

Reduction can be used to show:

1. Problem is tractable (solvable)

2. Problem is intractable (unsolvable)
   – To spread hardness.

Not part of course

Reductions Spread Tractability

If problem A reduces to problem B and B can be solved by a polynomial-time algorithm, then A can also be solved by a polynomial-time algorithm (Figure 19.2).

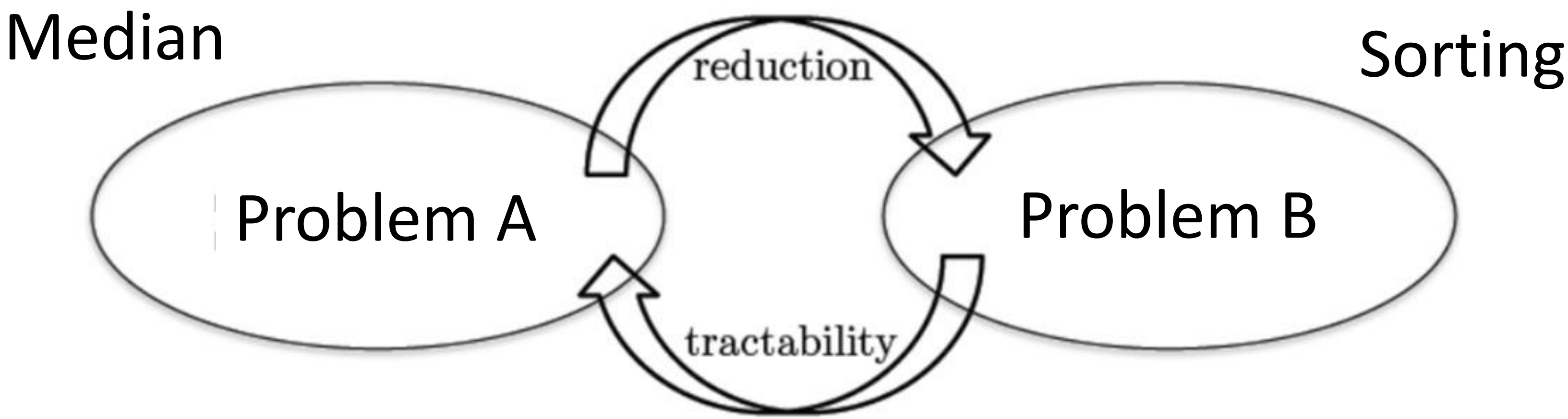


Figure 19.2: Spreading tractability from B to A: If problem A reduces to problem B and B is computationally tractable, then A is also computationally tractable.

If $A \leq_p B$ and $B \in P$, then $A \in P$

Not part of course

Reductions Spread Intractability

If problem A reduces to problem B and A is NP-hard, then B is also NP-hard (Figure 19.3).

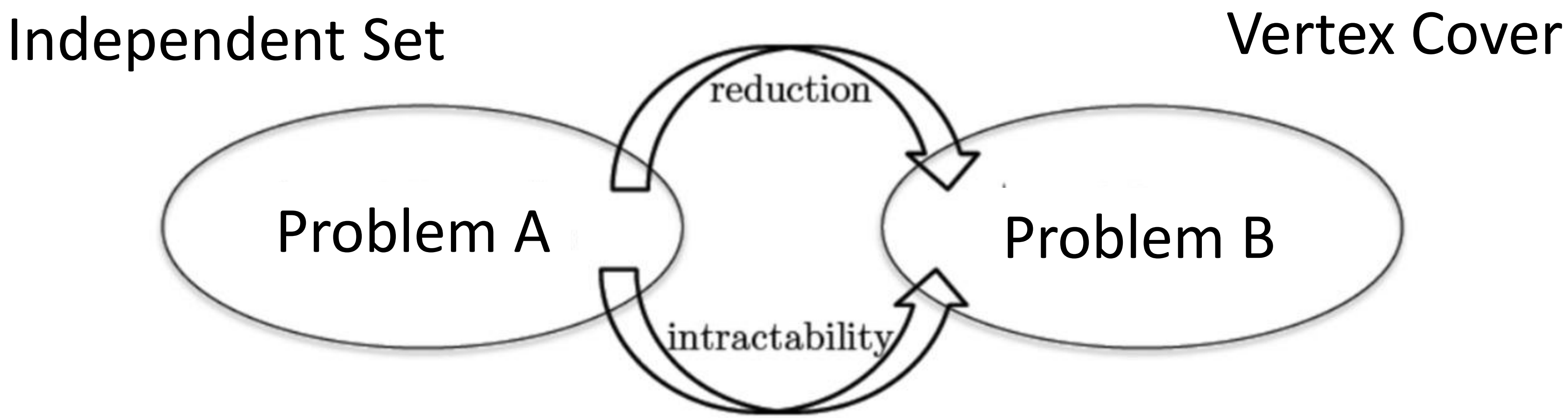
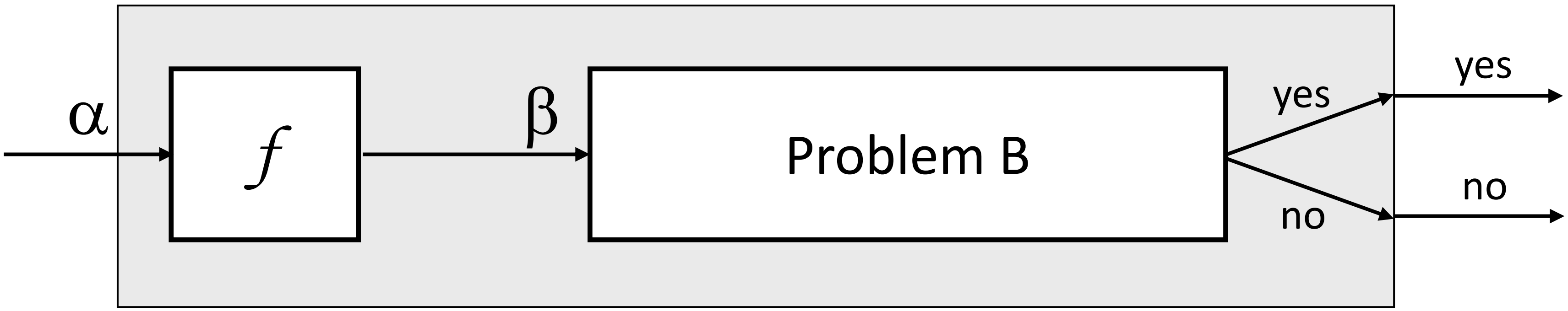


Figure 19.3: Spreading intractability in the opposite direction, from A to B: If problem A reduces to problem B and A is computationally intractable, then B is also computationally intractable.

if $A \leq_p B$ and $A \notin P$, then $B \notin P$

Implications of Reduction

- Problem A reduces to problem B if you can use an algorithm that solves B to help solve A.



Algorithm for Problem A

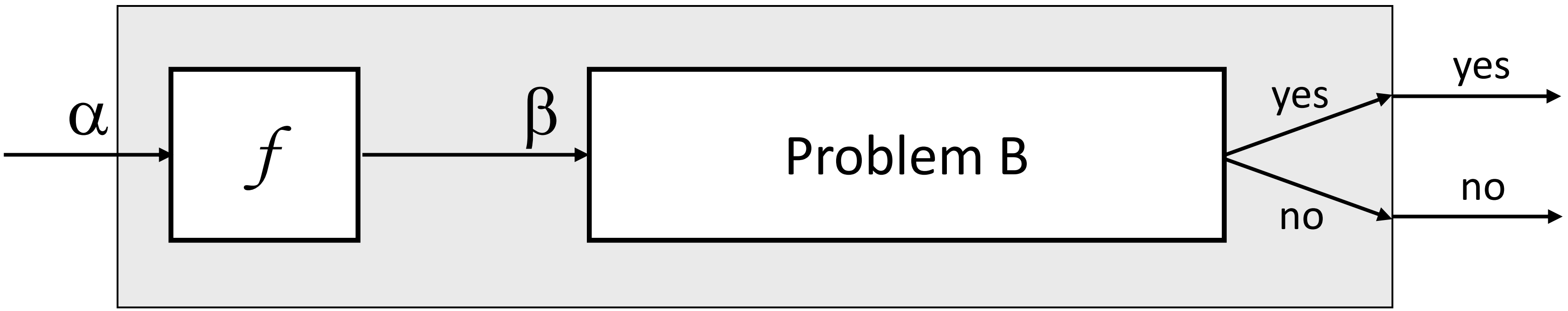
- If $A \leq_p B$ and $B \in P$, then $A \in P$

- if $A \leq_p B$ and $A \notin P$, then $B \notin P$

Cost of solving *A* = total cost of solving *B* + cost of reduction.

Reductions (Tractable Examples)

- Problem A reduces to problem B if you can use an algorithm that solves B to help solve A.



Algorithm for Problem A

- Example 1. [finding the median reduces to sorting]
- To find the median of *N* items:
  - Sort N items
  - Return item in the middle:

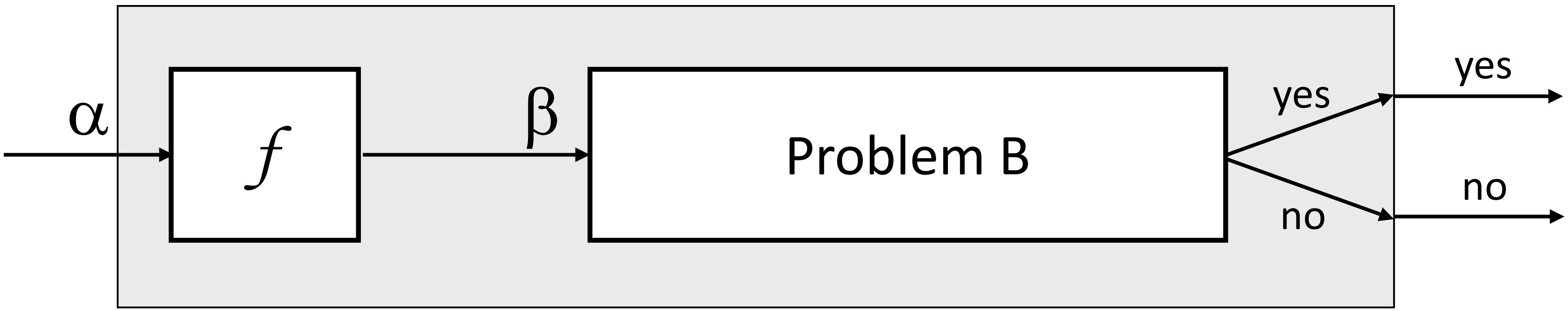
Cost of Sorting

Cost of Reduction

Cost of solving *finding median* = O(n log n) + 1

Reductions (Tractable Examples)

- Problem A reduces to problem B if you can use an algorithm that solves B to help solve A.



Algorithm for Problem A

- Example 2. [element distinctness reduces to sorting]
- To solve element distinctness on N items:
  - Sort N items
  - Check adjacent pairs for equality.

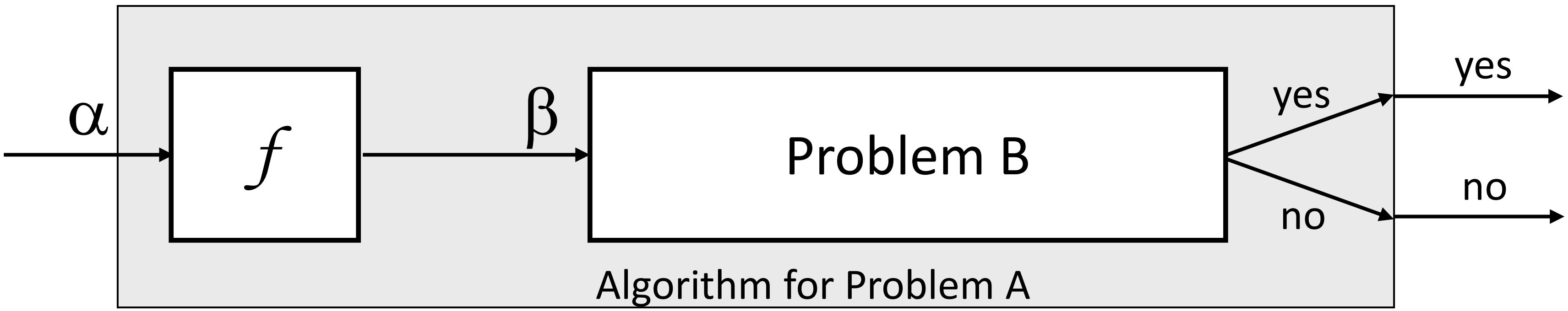
Cost of Sorting

Cost of Reduction

Cost of solving *element distinctness* = $\underbrace{O(n \log n)}_{\text{Cost of Sorting}} + \underbrace{O(n)}_{\text{Cost of Reduction}}$

Reductions (Tractable Examples)

- Convex hull. Given N points in the plane, identify the extreme points of the convex hull (in counterclockwise order).



- Example 3. [Convex hull reduces to sorting.]
- To solve convex hull:
  - Choose point p with smallest (or largest) y-coordinate.
  - Sort points by polar angle with p.
  - Consider points in order, and discard those that would create a clockwise turn.

Cost of Sorting   Cost of Reduction

Cost of solving *element distinctness* = O(n log n) + O(n)

RECIPE FOR PROVING PROBLEM Is NP-Complete

To Prove B Is NP-Complete

1. **Prove that B is a member of the class NP.** $B \in NP$

2. **Choose an NP-complete problem A.**

3. **Prove that there is a Levin reduction from A to B.**

Independent Set (Example 1)

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?

Ex. Is there an independent set of size ≥ 6 ?

Ex. Is there an independent set of size ≥ 7 ?

{1} is independent Set

{2, 3} is independent Set

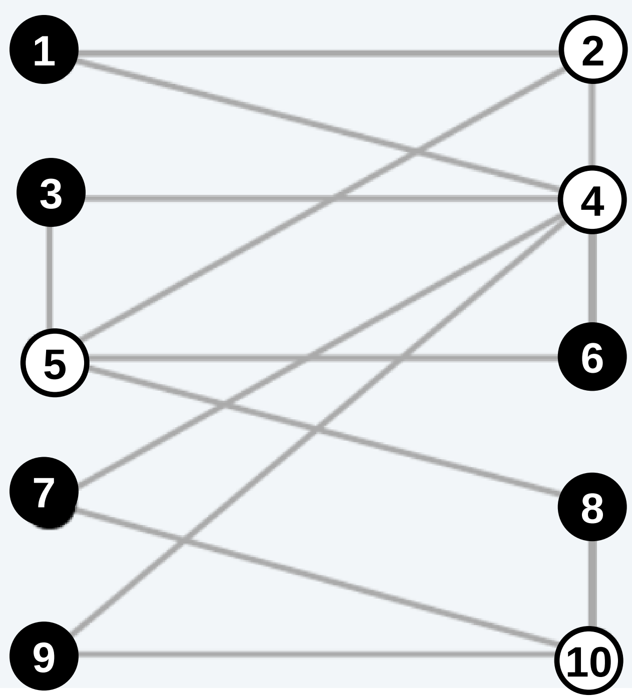
{1, 3} is independent Set

{1, 3, 6} is independent Set

~~{2, 5, 6, 7} is independent Set~~

{1, 3, 6, 7} is independent Set

{1, 3, 6, 7, 8 , 9} is independent Set



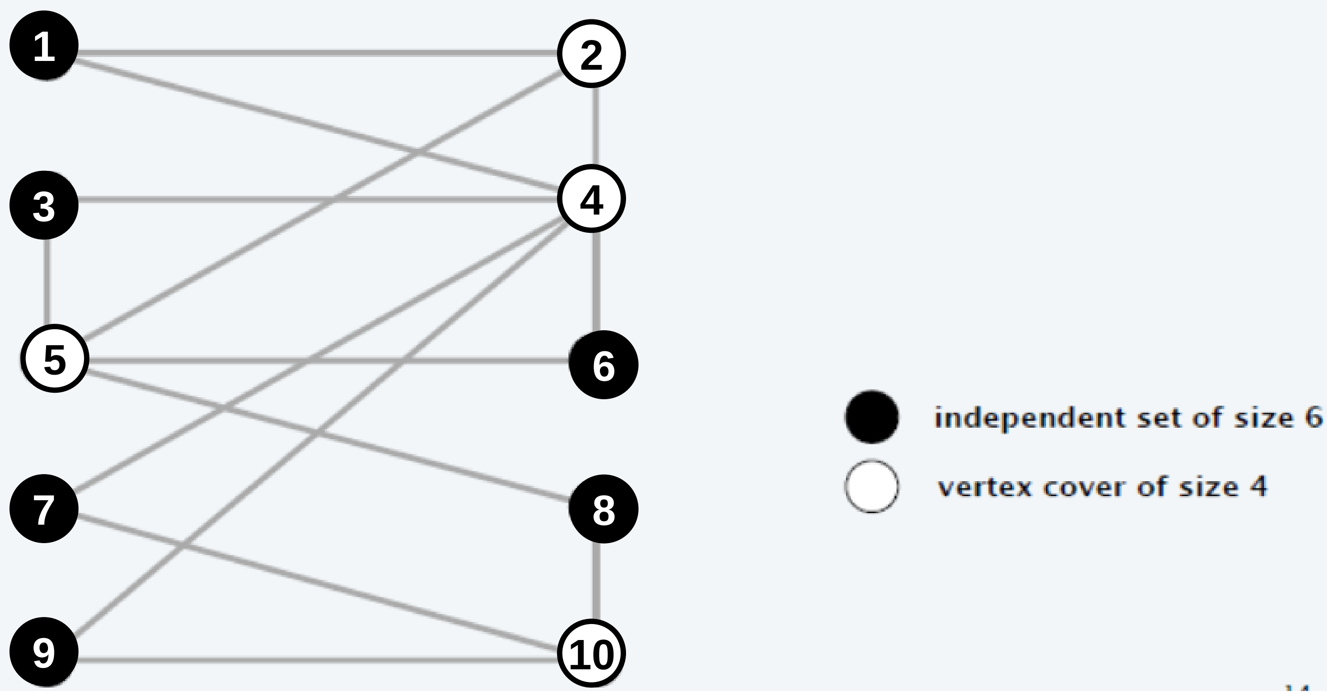
independent set of size 6

Vertex Cover (Example 1)

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?

Ex. Is there a vertex cover of size ≤ 4 ?

Ex. Is there a vertex cover of size ≤ 3 ?



Vertex Cover is NP-Complete

Recipe to Prove a Problem Is NP-Complete

- Prove that a Vertex Cover is NP-Complete
- Step 1. Vertex Cover ∈ NP
- Step 2. Choose an NP-Complete problem A (Independent Set).

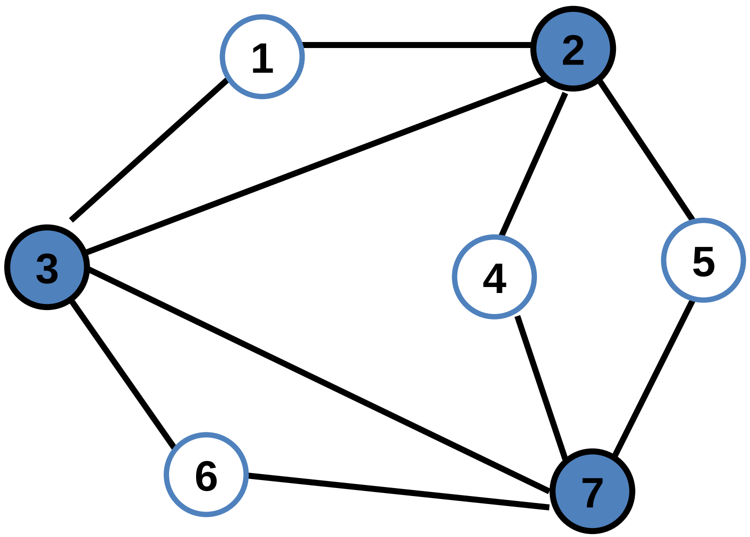
  Prove that A (Independent set) reduces to B (Vertex Cover)

  INDEPENDENT-SET $\leq_p$ VERTEX-COVER.

Step 1: Vertex Cover ∈ NP

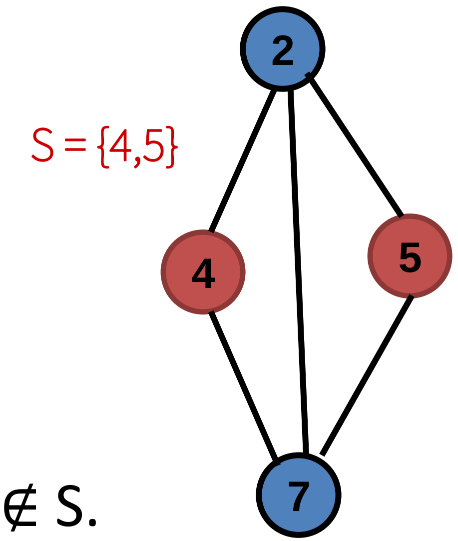
- Given Graph G = (V, E) contains Vertex cover of Size 3?

S is vertex cover if every edge in E has at least one endpoint in S. (Example V.C = {2,3,7})



Vertex cover and independent set reduce to one another

- **Lemma:** INDEPENDENT-SET $\leq_p$ VERTEX-COVER.
- **Proof:** We show S is an independent set of size k, iff $V - S$ is a vertex cover of size $n - k$.



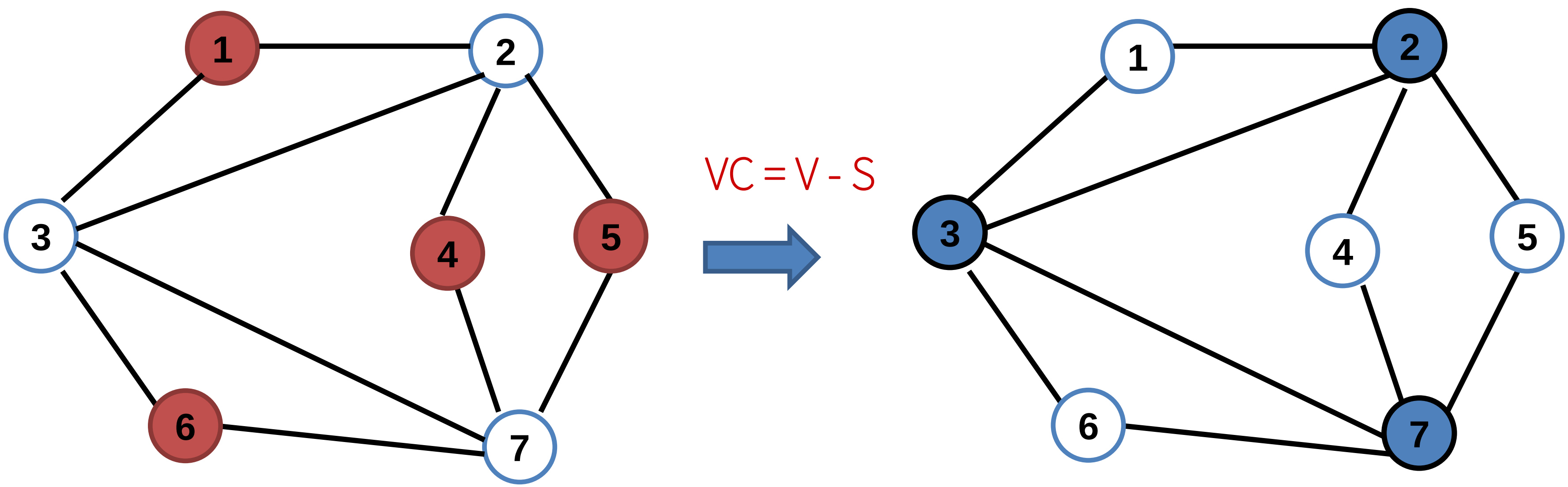
  - Let S be any independent set of size k.
  - Consider an arbitrary edge (u, v) ∈ E.
  - S independent ⇒ either u ∉ S, or v ∉ S, or both ∉ S.
  - Vertex Cover (V-S)⇒ either u ∈ V – S, or v ∈ V – S, or both ∈ V – S.
  - Thus, V – S covers (u, v). ▪

Independent Set (Example 2)

- S is independent if there are no edges between vertices in S. (Example I.S = {1,4,5,6})

V.C = V – S

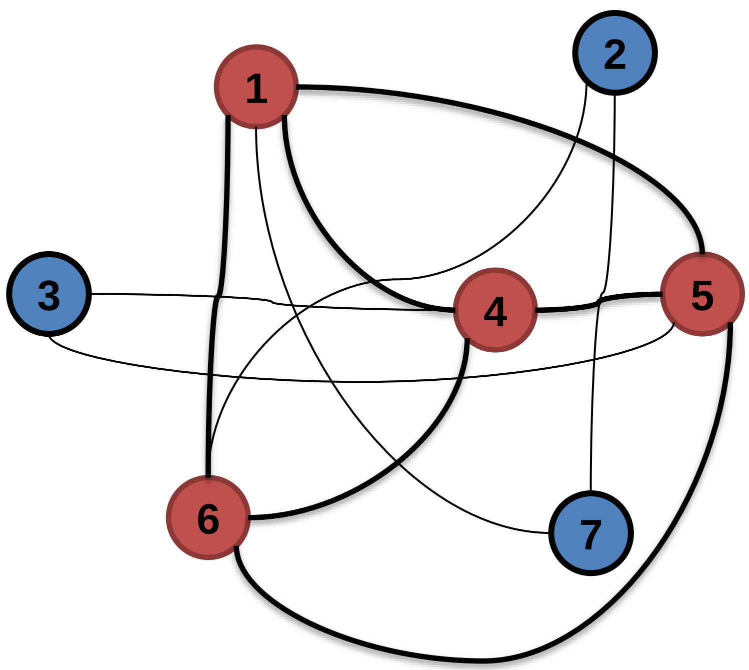
(V.C ={2,3,7})



Clique (Example 2)

- Clique
  - Graph G = (V, E), a subset S of the vertices is a clique if there is an edge between every pair of vertices in S. (Example Clique = {1,4,5,6})

Clique of size 4



Clique is NP-Complete

Recipe to Prove a Problem Is NP-Complete

- Prove that a Clique is NP-Complete

- Step 1. Clique $\in$ NP.

- Step 2. Choose known NP-Complete problem A (IS).

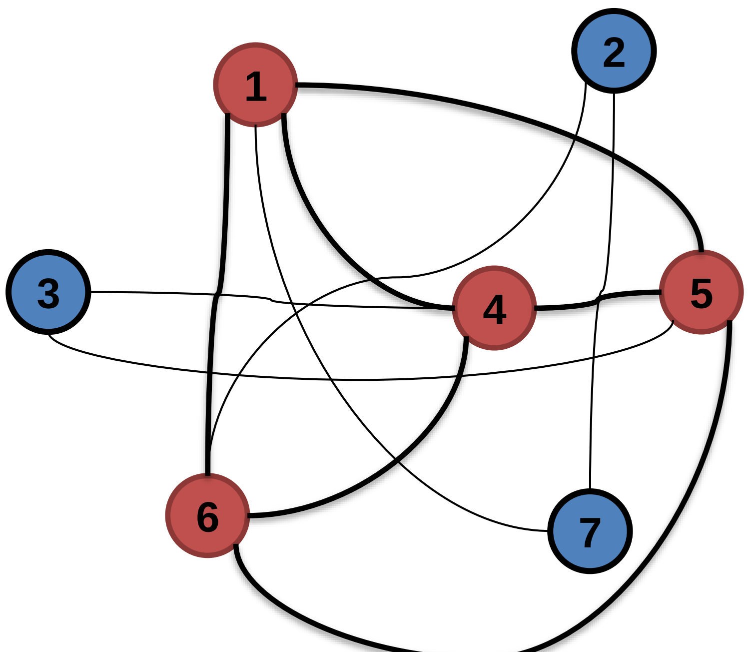
  Prove that A (IS) reduces to B (Clique)

  IS $\leq_p$ Clique.

Step 1: Clique ∈ NP.

- Graph G = (V, E), a subset S of the vertices is a clique if there is an edge between every pair of vertices in S. (Example Clique = {1,4,5,6})

- Given graph contains Clique of size = 4 ?



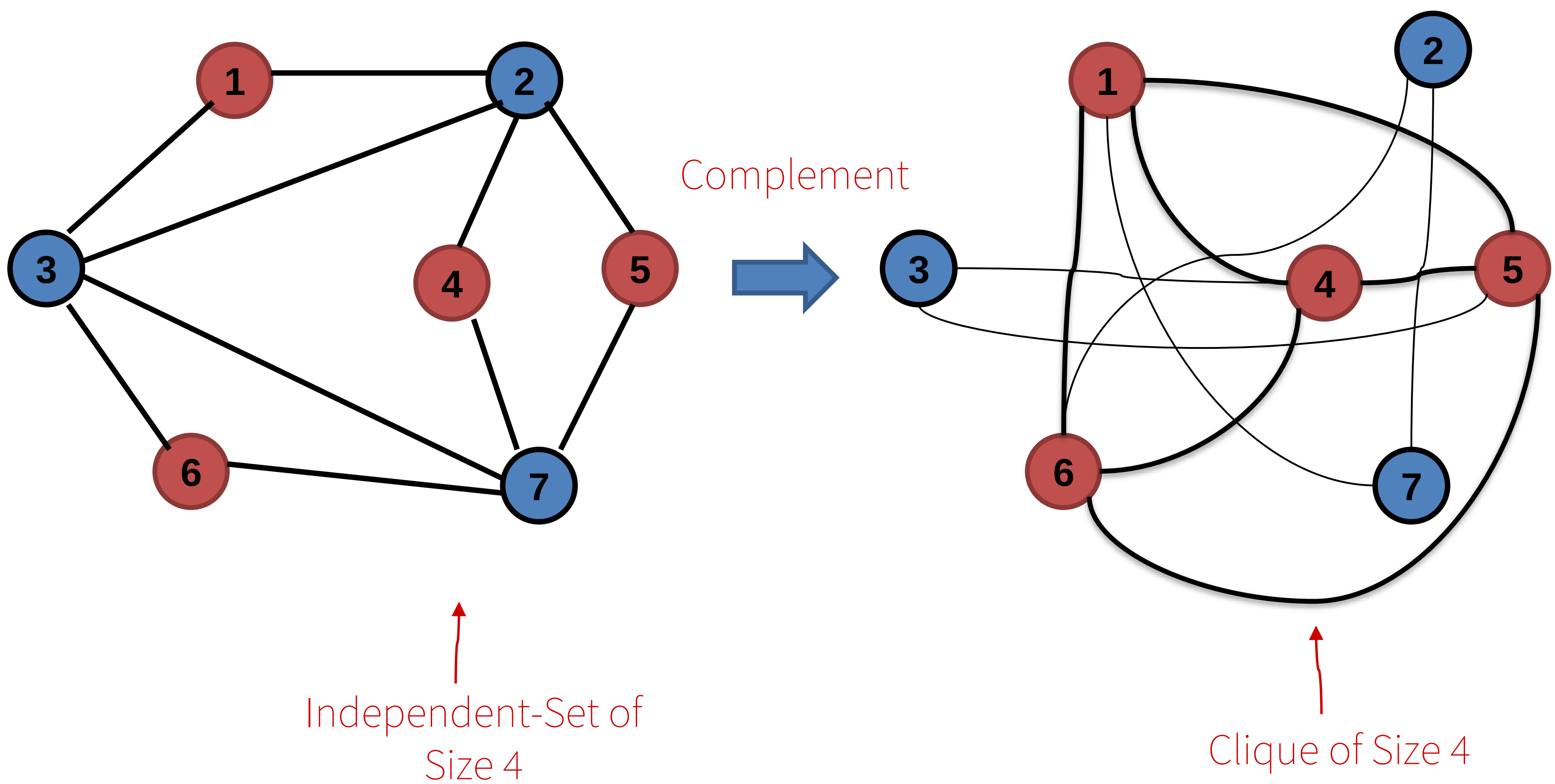
Step 2: IS $\leq_p$ Clique

- Lemma: S is Independent in G iff S is a Clique in the complement of G

- To reduce IS to Clique, we compute the complement of the graph. The complement has a clique of size K iff the original graph has an independent set of size K.

- Construction of Complement of the graph can easily be done in polynomial time.

IS $\leq_p$ Clique (Example 2)



Reduction: Independent Set, Vertex Cover, and Clique (Example 3)

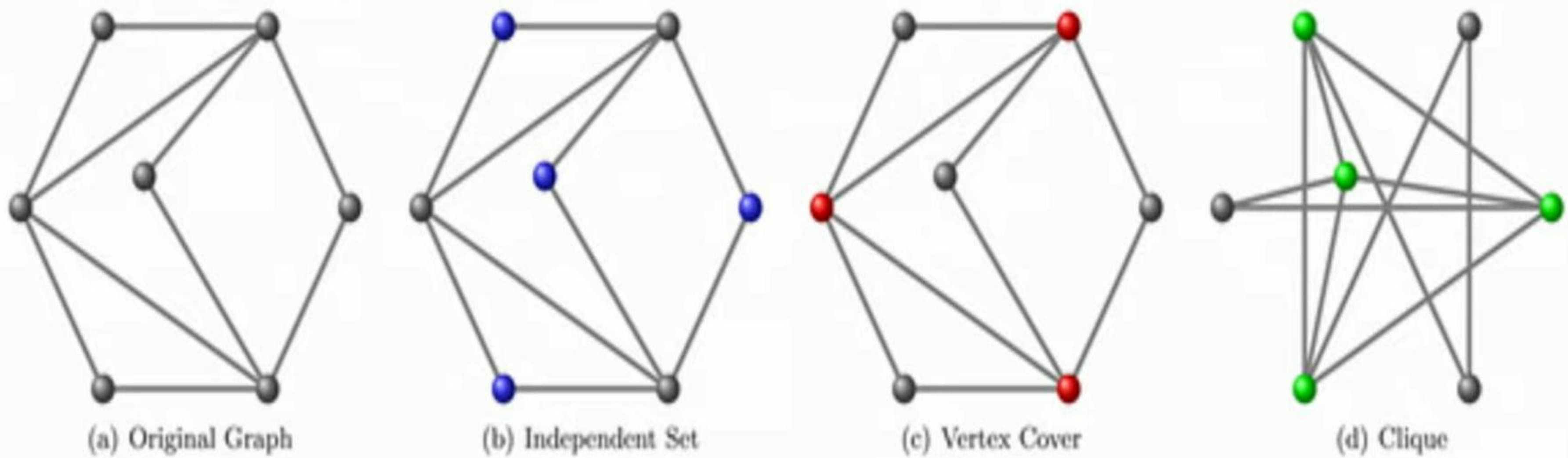


Figure 1: Relations among Independent Set, Vertex Cover, and Clique

INDEPENDENT-SET is NP-Complete

Recipe to Prove a Problem Is NP-Complete

- To prove that a problem B (INDEPENDENT-SET) is NP-Complete:

- Step 1. INDEPENDENT-SET $\in$ NP

- Step 2. Choose an NP-Complete problem A (3-SAT).

  Prove that A (3-SAT) reduces to B (INDEPENDENT-SET)

  3-SAT $\leq_p$ INDEPENDENT-SET

Satisfiability

Literal. A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

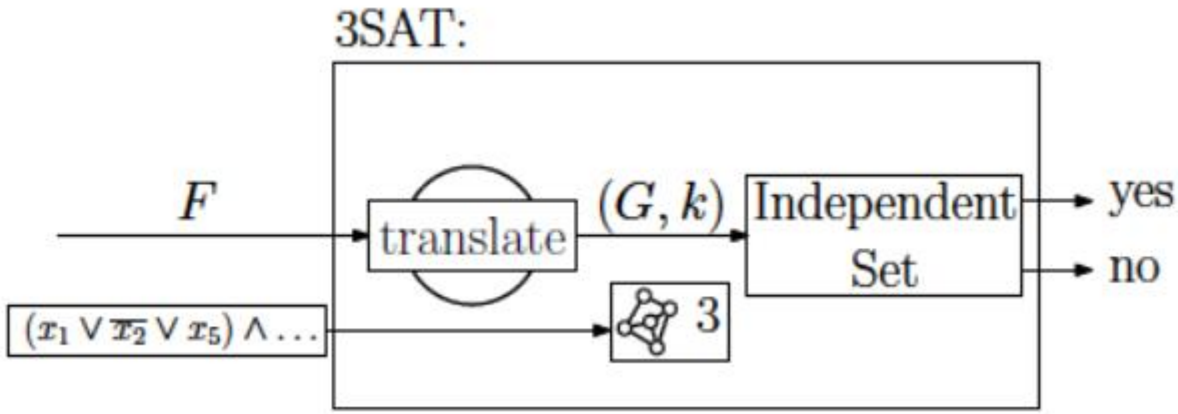
Conjunctive normal form (CNF). A propositional formula Φ that is a conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

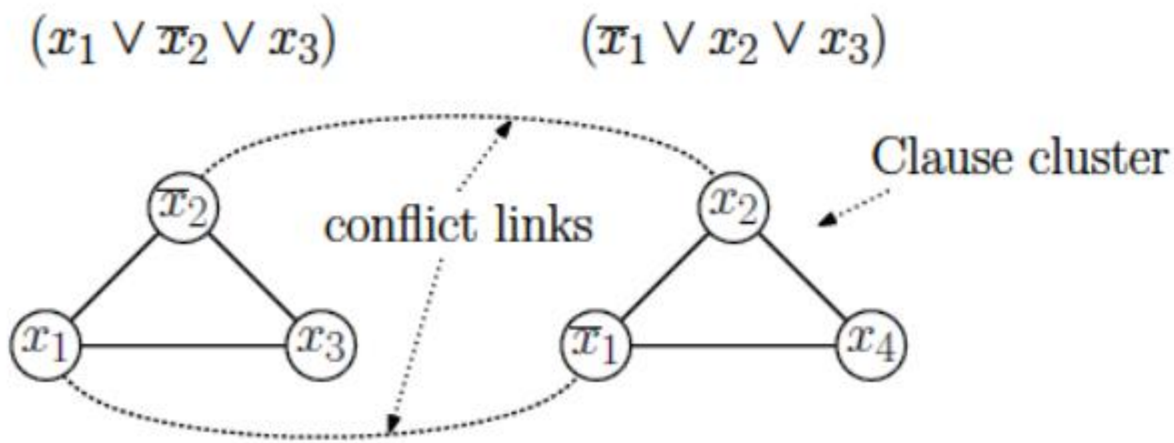
SAT. Given a CNF formula Φ, does it have a satisfying truth assignment?
3-SAT. SAT where each clause contains exactly 3 literals
(and each literal corresponds to a different variable).

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

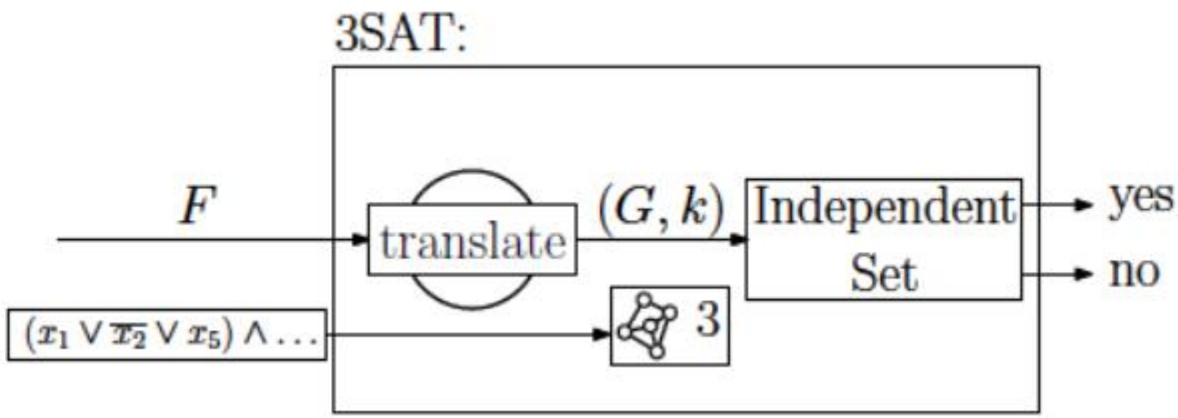
yes instance: x_1 = true, x_2 = true, x_3 = false, x_4 = false

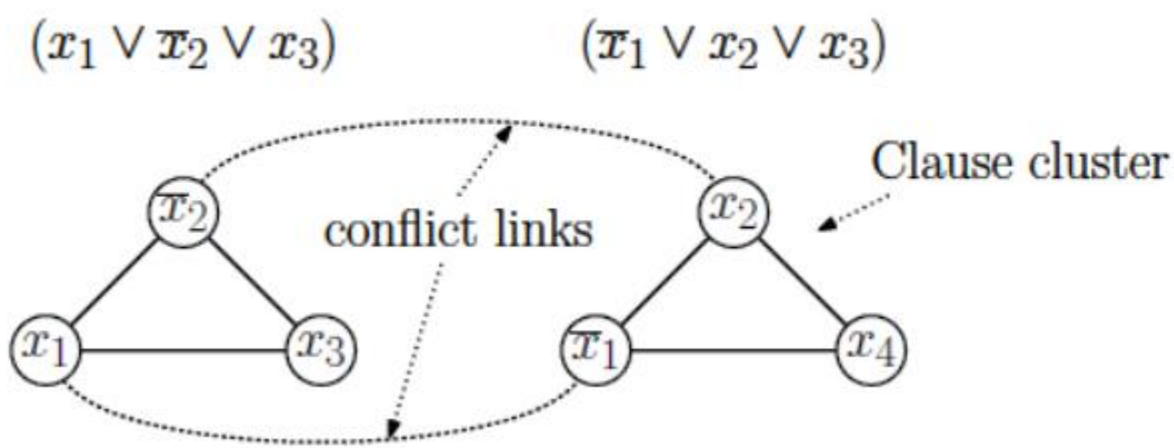
3-satisfiability reduces to independent set





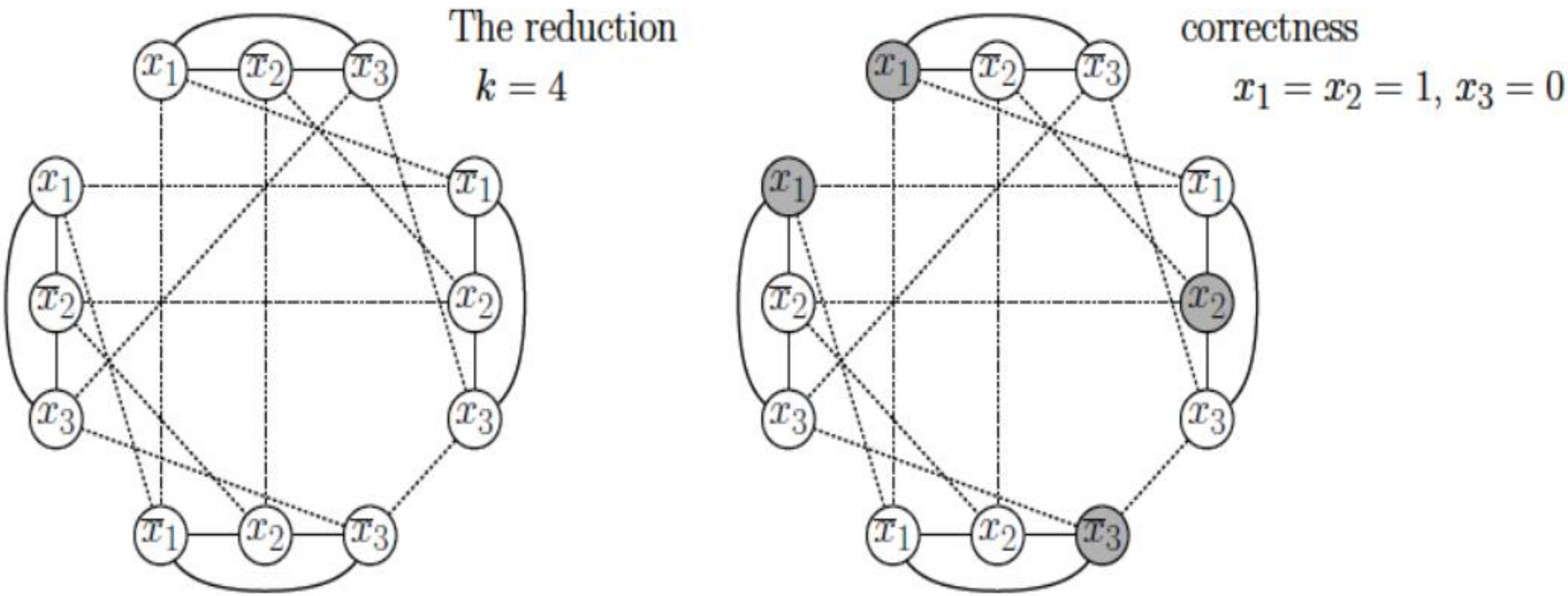
3-satisfiability reduces to independent set





3-satisfiability reduces to independent set

$$F = (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3).$$



(a) (b)

Cook-Levin theorem shows that 3-SAT is a "universal" problem