EE 441 – CH11

COMPUTATIONAL COMPLEXITY

Brute Force Solution

Consider the following problem:

Satisfiability problem: given a boolean expression Φ find an assignment of values (0, 1) to variables x_i that causes Φ to evaluate to true (i.e. logic value 1)

An instance of the problem: $\Phi = (x_1 + x_2' + x_3)(x_1' + x_2' + x_3 + (x_4 x_5))x_2 x_3$

(another notation $\Phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee (x_4 \wedge x_5)) \wedge x_2 \wedge x_3$

Brute force: Likewise satisfiability problem, for many non-trivial problems, there is a natural brute force search algorithm that **checks every possible solution.**

For example, for the satisfiability problem, all possible value combinations for the variables x_1, x_2, x_3, x_4 and x_5 such as {00000, 00001, …, 11111} may be checked in order reach a solution.

Brute force

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

Brute Force Solution

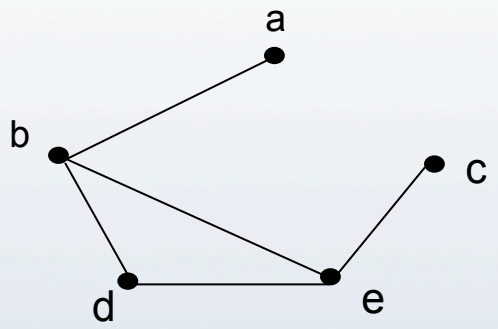
Vertex cover problem:

Let $G=(V,E)$ be a graph where $V=\{v_1, v_2,..,v_N\}$ is the vertices and $E=\{(v_i,v_j)\}$ is the edges of the graph.

Example:

$V=\{a,b,c,d,e\}$

$E=\{(a,b),(b,d),(b,e),(c,e),(d,e)\}$



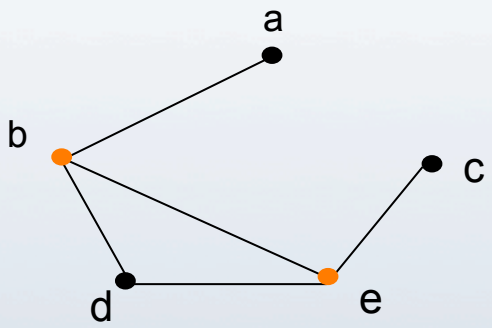
Cover C of G is a subset of V such that for each edge (v_i, v_j) in E, either v_i or v_j is in C.

A ***minimum cover*** of G is a set C^* such that the number of nodes in C^* is the minimum among all the covers of G, that is $|C^*| \le |C|$.

Given a graph G, the **Vertex Cover** problem tries to find a minimal cover.

Brute Force Solution

- **Vertex cover problem brute force solution:**



All Covers:

C_1={a,b,c,d,e}
C_2={a,b,c,d}
C_3={a,b,c,e}
C_4={a,b,d,e}
C_5={a,c,d,e}
C_6={b,c,d,e}
C_7={a,b,e}
C_8={a,d,e}
C_9={b,c,d}
C_{10}={b,c,e}
C_{11}={b,d,e}
C_{12}={b,e}

Minimal Cover: C_{12}={b,e}

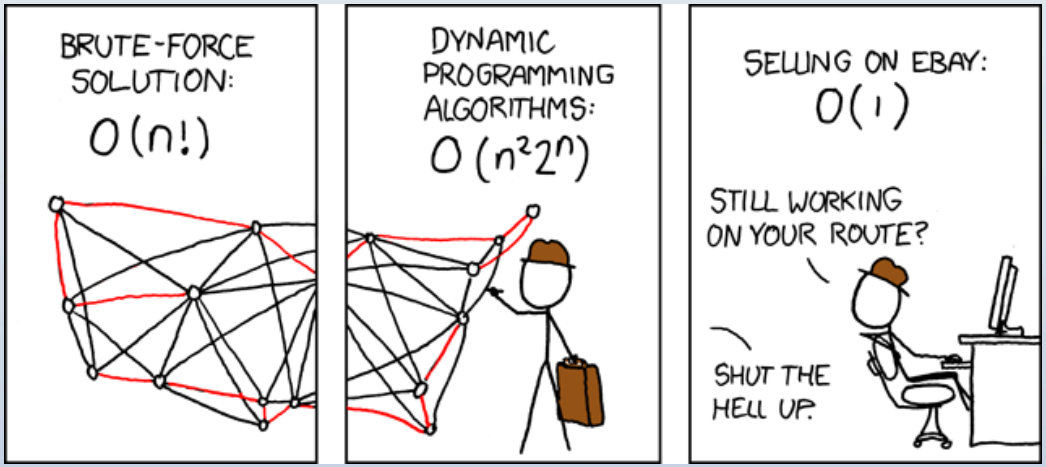
Brute Force Solution

Traveling salesman (TSP):

Given a list of N cities and their pairwise distances, the task is to find the shortest possible route that visit a each city exactly once and return to original city.

Brute force solution: Consider each permutaion of n cities and calculate the length of the correponding tour, then choose the minimal one.

If it took 1 microsecond to calculate each possibility then it takes 10^{140} centuries to calculate all possibilities when n = 100



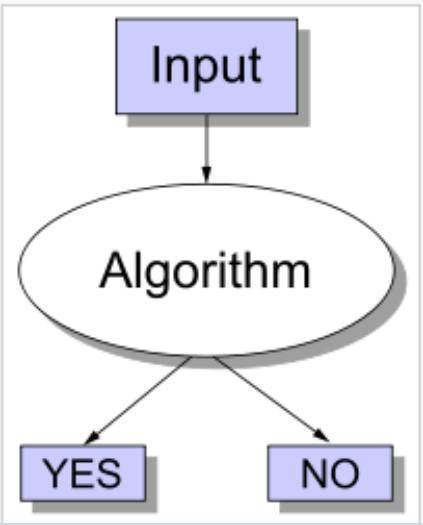
Decision problems

Usually, we've considered optimization problems: given some input instance, output some answer that maximizes or minimizes a particular objective function.

Most of computational complexity deals with a seemingly simpler type of problem: the decision problem.

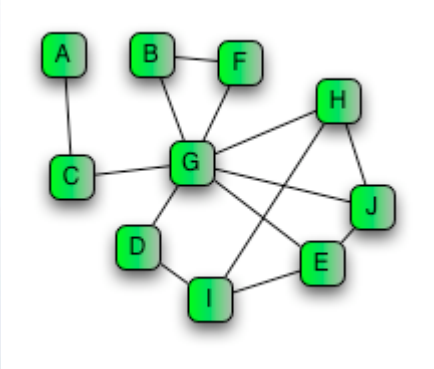
A decision problem just asks for a yes or a no.

Optimization problems can be converted to decision problems



Decision problems

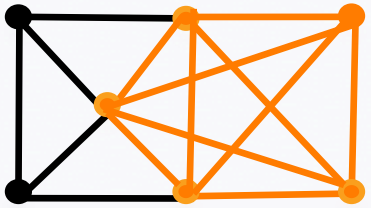
Shortest Path



Optimization problem: Find a path between A and E that uses the fewest edges

Decision Problem: Does a path exist between node A and E with at most 3 edges?

Decision problems

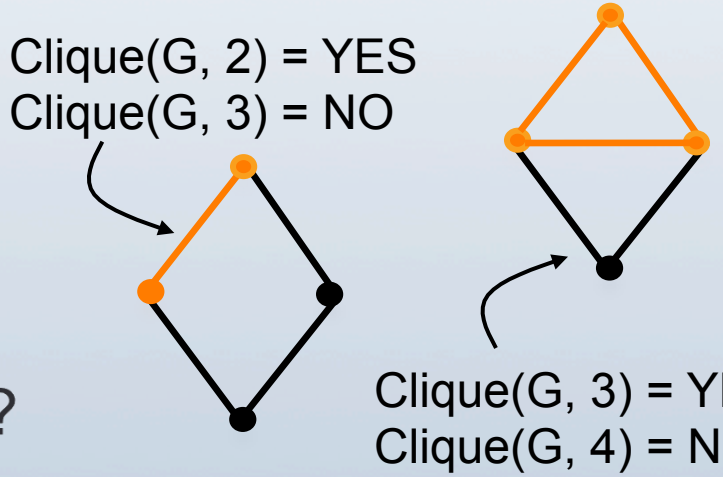


Clique Problem:

- Undirected graph G = (V, E)
- **Clique:** a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph)
- **Size of a clique:** number of vertices it contains

Optimization problem:

- Find a clique of maximum size



Decision problem:

- Does G have a clique of size k?

Encoding an Instance

- We can encode an instance of a decision problem as a string.

Example: Does the given G has a cover of size at most k?

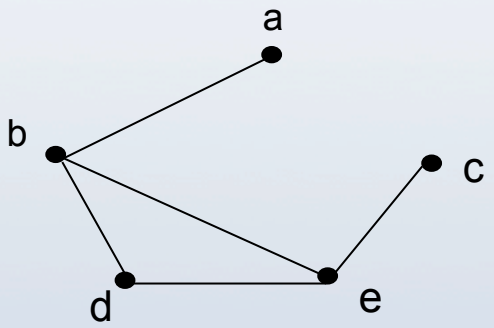
The encoding the vertex cover problem as a string for the instance given (G, k=3) might be

"V:a,b,c,d,e;E:(a,b),(b,d),(b,e),(c,e),(d,e);3?"

or

"3;5;(1,2),(2,4),(2,5),(3,5),(4,5)"

(in fact, there might be a better encoding)



- How do we "know" intuitively that all of the problems we've considered so far can be encoded as a single string?

- Because we can represent them in RAM as a string of bits!

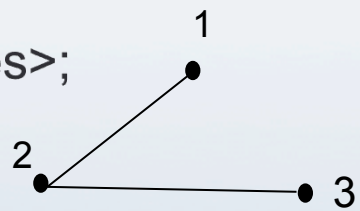
Language

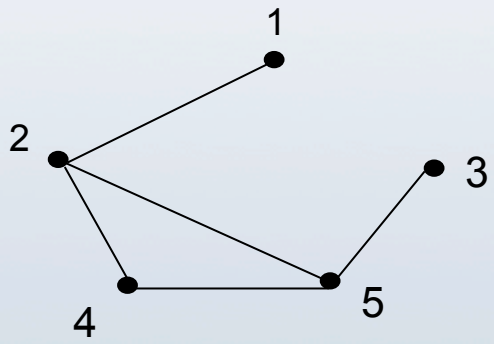
Definition: A language is a set of valid strings.
(Analogy: English is the set of valid sequences of English words.)

A decision problem X can be represented just sets of valid strings (i.e. answer is yes):

Example: vertex cover
syntax: <k>;<n>;<list of edges>;

…
1;3;(1,2),(2,3);
…
1;5;(1,2),(2,4),(2,5),(3,5),(4,5);
2;5;(1,2),(2,4),(2,5),(3,5),(4,5);
….
1;6;(1,2),(2,4),(2,5),(3,5),(4,5);
2;6;(1,2),(2,4),(2,5),(3,5),(4,5);





Hence, any decision problem is equivalent to deciding membership in some language.

We talk about “decision problems” and “languages” pretty much interchangably.

Class P problems

Class P consists of **decision** problems that are solvable in polynomial time

Polynomial-time algorithms

Worst-case running time is $O(n^k)$, for some constant k

Examples of polynomial time:

$O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$

Examples of non-polynomial time:

$O(2^n)$, $O(n^n)$, $O(n!)$

Class P problems

Let

- X be a set of strings (i.e. a language as given for vertex cover)
- s be an instance of X
- Algorithm A solves problem such that

  X: A(s) = `yes` iff $s \in X$.

Polynomial time: Algorithm A runs in **poly-time** if for every string s, A(s) terminates in at most p(|s|) "steps", where p(·) is some polynomial and |s| is the problem size represented by s.

Certifier

Certifier Algorithm: X: C(s,t) is a certifier on problem X if for any instance s such that, $s \in X$ iff there exists a string t such that C(s,t)=Yes.

Intution about Certifier algorithm

- Certifier doesn't determine whether $s \in X$ on its own.
- Certifier checks only a proposed proof t,

Certificate: Instance which leads the true answer.

If t is a certificate of s then

C(s,t)=YES, which in turn implies $s \in X$,

otherwise

C(s,t)=NO, and therefore whether $s \in X$ is still unknown

Certifier

Remarks:

X is a solution set for a specific decision problem
(i.e. all the instances resulting in YES answer)
s: a specific case of a decision problem (instance)
t : a specific certificate candidate

We want to know if $s \in X$ using a certifier

- Certifier algorithm: C(s,t) produces a YES/NO answers the question: “Is $s \in X$ according to t?”
- If C(s,t)=YES for some t then t is certificate indicating that $s \in X$
- If $s \notin X$ then there is no certificate and C(s,t)=NO for any t,
- If C(s,t)=NO, t is not a certificate and we still do not know if $s \in X$ or $s \notin X$

Certifier

Example: Composite numbers

Composites: Given integer s, is s composite (i.e. not a prime) ?

Certificate: A nontrivial factor t of s. Note that such a certificate exists iff s is composite.

Certifier: C(s,t) that if s is a multiple of t, return true, but otherwise return false.

Instance: s = 437669

Certificate: t = 541 or t=809 results in
X:C("437669","541")=yes
X:C("437669", "809")=yes

Instance
s: 437669

Candidate
t: 541

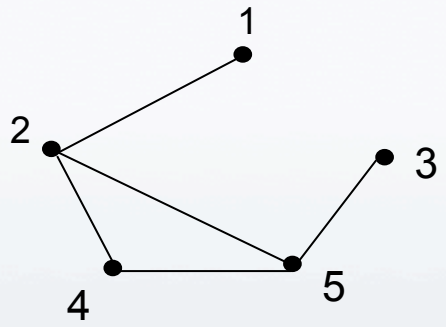
Certifier:
Is s multiple of t ?

YES:
t is a certificate
so s∈X

NO:
t is not a
certificate
if s∈X or s∉X
still not known

Certifier

Example: Vertex cover



Consider the following instances of the vertex cover problem:
s_1="2;5;(1,2),(2,4),(2,5),(3,5),(4,5)"
s_2="3;5;(1,2),(2,4),(2,5),(3,5),(4,5)"

t="{2,5}" is a certificate for s_1, so certifier produces:
X: C(s_1,t)=YES, so it can be concluded $s_1 \in X$

But given t is not a certificate for s_2:
X: C(s_2,t)=NO,
whether $s_2 \in X$ is true or not can not be concluded by using t.

Certifier

Example : Satisfiability

SAT: Given a boolean formula, is there a satisfying assignment?

Certificate: An assignment of truth values

Instance: $\Phi = x_1(x_2' + x_3x_4)(x_1' + x_4x_5')$

Certificate: $x_1=1$, $x_2=0$, $x_3=1$, $x_4=1$, $x_5=0$, there may exist such other certificates

Instance: $\Phi = x_1(x_2' + x_3x_4)(x_1' + x_2x_4')$

Certificate: No certificate, since no assignment is resulting in YES answer

Certifier

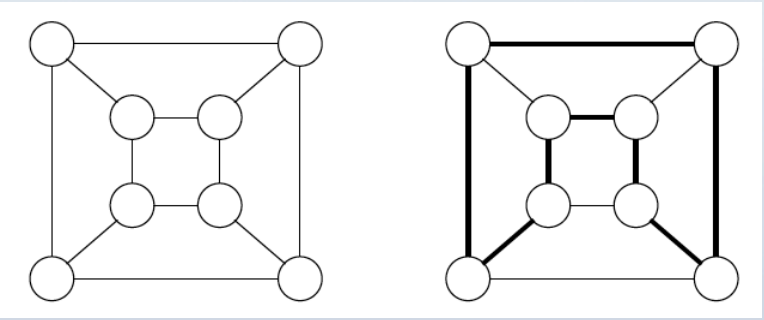
Example: Hamiltonian Cycle

HAM-CYCLE: Given an undirected graph G=(V,E), does there exist a simple cycle C that visits every node?

Certificate: A permutation of the n nodes.

Certifier: Check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.

An **Instance** and **Certificate** as in the figure



Instance s certificate t

Algorithm vs Problem Complexity

- The ***algorithmic complexity*** of a computation is some measure of how *difficult* is to perform the computation (i.e., specific to an algorithm)
- The **complexity of a computational *problem*** or *task* is the complexity of the algorithm with the **lowest** order of growth of complexity for solving that problem or performing that task.
  - *e.g.* the problem of searching an ordered list has O(log n) time complexity.
- **Computational Complexity**: deals with **classifying** problems by how hard they are.

Tractable/Intractable Problems

Problems in P are also called **tractable**

Problems **not** in P are **intractable or unsolvable**

- Can be solved in reasonable time only for small inputs
- Or, can not be solved at all

Are non-polynomial algorithms always worst than polynomial algorithms?

- $n^{1,000,000}$ is *technically* tractable, but really impossible
- $n^{\log \log \log n}$ is *technically* intractable, but easy

Example of Unsolvable Problem

Turing discovered in the 1930's that there are problems **unsolvable** by *any* algorithm.

The most famous of them is the ***halting problem***

- Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an "*infinite loop*?"

Example:

- `main(){while(true){}; }` does not halt; rather, it goes on forever in an infinite loop
- `main(){cout<<"EE441";}` halts very quickly.

A more complex program might be more difficult to analyze. The program could be run for some fixed time. If the program does not halt, there is in general no way to know if the program will eventually halt or run forever.

Intractable Problems

Decision problem can be classified in various categories based on their degree of difficulty, e.g.,

- P: decision problems solvable in poly time (tractable)
- NP: decision problems verifiable in poly time.
- NP-complete (NP-C)
- NP-hard

Warning: NP does **not** mean “non-polynomial” but it means “**Nondeterministic Polinomial”**

Nondeterministic and NP Algorithms

Let s be an instance of a decision problem X

Nondeterministic algorithm is a two stage procedure:

1. Nondeterministic ("guessing") stage:
   generate randomly an arbitrary string "t" that can be thought of as a candidate "certificate"

2. Deterministic ("verification") stage:
   Certifier C(s,t) produces YES iff t is a certificate for the problem instance s

NP algorithms (Nondeterministic polynomial)

Verification stage ie. C(s,t) is polynomial

Class of "NP" Problems

Class NP consists of problems that could be solved by NP algorithms, i.e., verifiable in polynomial time

If we were given a "certificate" of a solution, we could verify that the certificate is correct in time polynomial to the size of the input

Note that:

The verifier-based definition of NP requires easy-to-verify certificates for YES answers to show $s \in X$, but does *not require certificate* to show that $s \notin X$

The class of problems also with certificates to show that $s \notin X$ called co-NP.

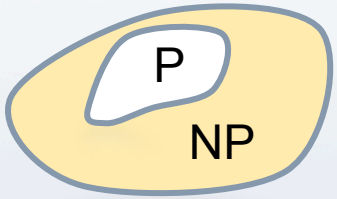
In fact it is an open question whether all problems in NP also have certificates for $s \notin X$ and thus are in co-NP.

Relationships

P: Decision problems for which there is a **poly-time** algorithm to decide on s.

NP: Decision problems for which there is **a poly-time certifier.**

$P \subseteq NP$



Proof: Consider any problem X in P.

By definition, there exists a poly-time algorithm A(s) that decides on whether $s \in X$ or $s \notin X$. Then use A(s) itself as certifier: C(s,t)=A(s).

Relationships

- The big (and **open question**) is whether

  $P \subseteq NP$ or $P = NP$?

  i.e., if it is always easy to check a solution, should it also be easy to find a solution?

Most computer scientists believe that this is false but we do not have a proof …

Princeton CS Building, West Wall

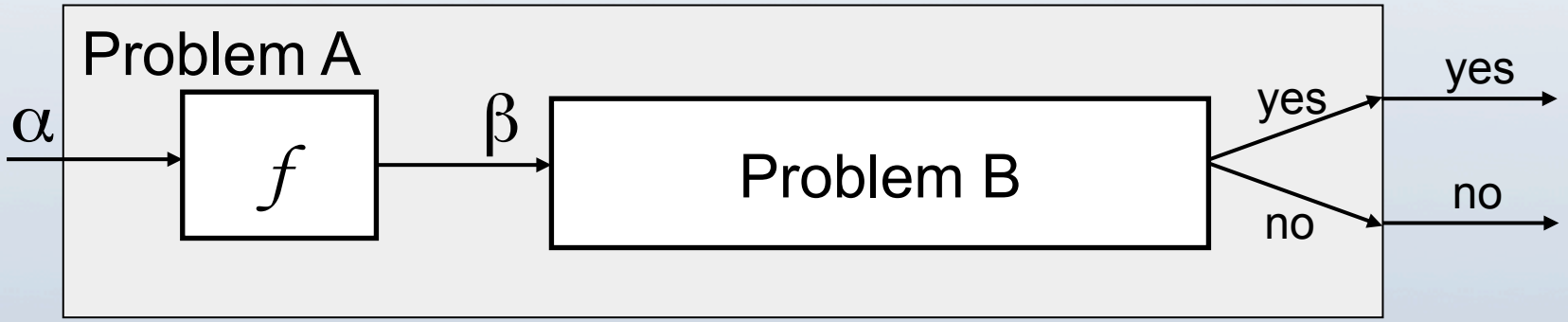
Character	ASCII	Bits
P	80	1010000
=	61	0111101
N	78	1001110
P	80	1010000
?	63	0111111

Reductions

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 Reductions

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

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

2. $A(i) = YES \Leftrightarrow B(f(i)) = YES$

Remark: $\leq_p$ is a transitive relation, that is $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$

Polynomial Reductions

Example:

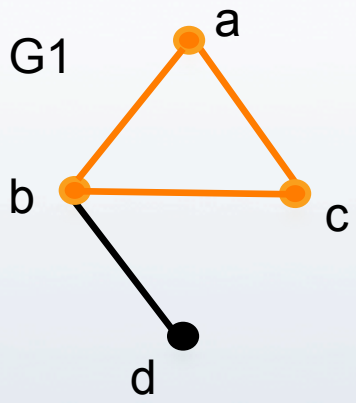
Reducing *k*-clique to vertex cover in P

- The *complement* G_C of a graph G contains exactly those edges not in G
- Compute G_C in polynomial time
- G has a clique of size *k* iff G_C has a vertex cover of size $|V| - k$

The reverse is also true for this example, that is vertex cover can be reduced to k-clique by applying the same algorithm.

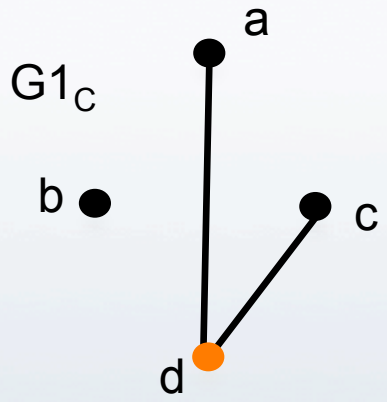
Polynomial Reductions

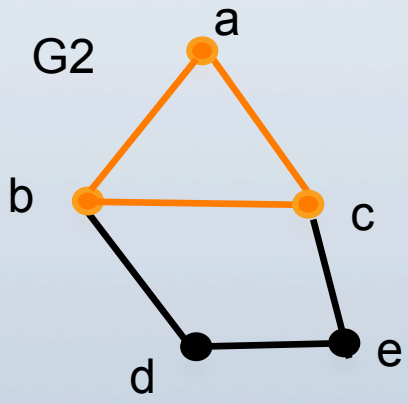
Examples for Reducing *k*-clique to vertex cover



G1 has a clique: {a,b,c}
size 3

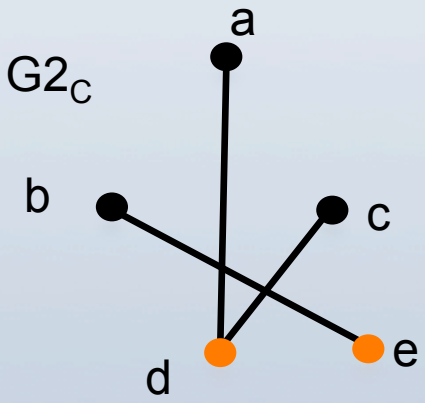
$G1_C$ has a vertex cover
{d} of 1 which is
|V| - *k=4-3*





G2 has a clique: {a,b,c}
size 3

$G2_C$ has a vertex cover
{d,e} of size 2=5-3



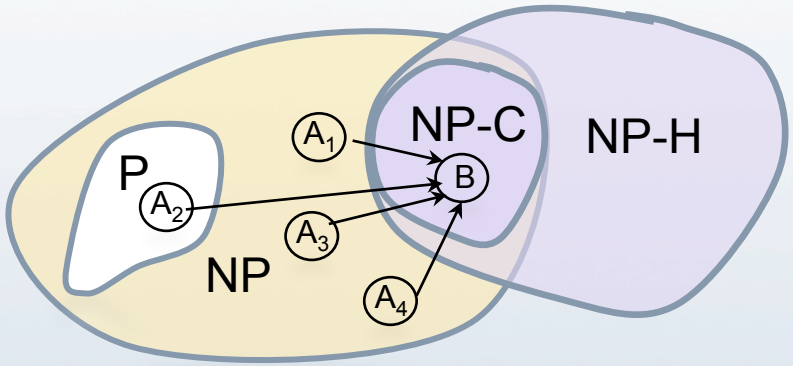
NP-Completeness

NP-complete problems are defined as the *hardest* problems in NP
Most practical problems turn out to be either P or NP-complete.

A problem B is **NP-complete (NP-C)** 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**

NP-Completeness

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

Cook-Levin Theorem

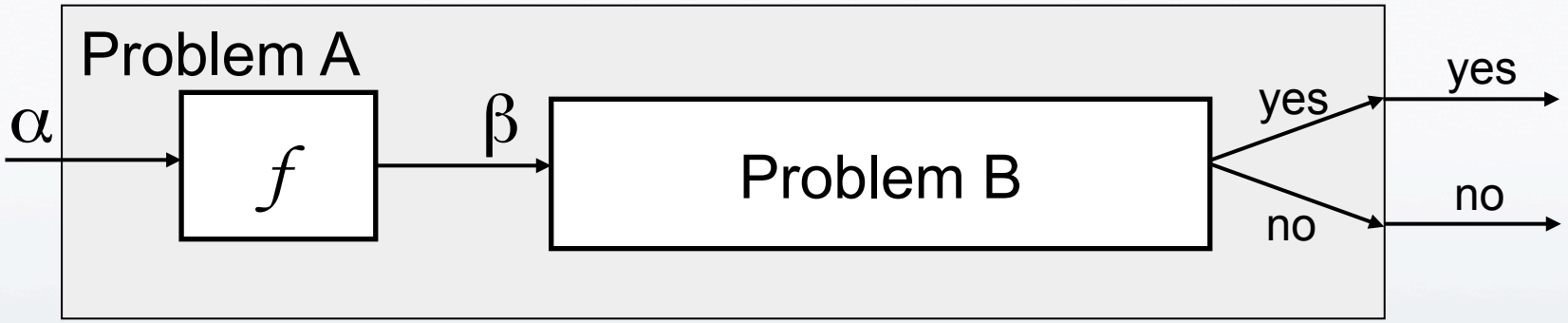
Theorem: SAT is NP complete

Proven in 1971 by Cook. Slightly different proof by Levin independently.

Idea of proof: There are two parts to proving that the Boolean satisfiability problem (SAT) is NP-complete. One is to show that SAT is an NP problem. The other is to show that **any NP problem can be reduced to an instance of a SAT problem** by a polynomial time many-one reduction. In the proof by Cook, the operation of a Turing Machine for an instance s of any problem $X \in NP$ encoded as a SAT formula so that the formula is satisfiable if and only if the Non-Deterministic Turing machine would accept the instance s, i.e. $X \leq_p SAT$

The proof will not be covered here since Turing Machine out of scope of the course.

Implications of Reduction

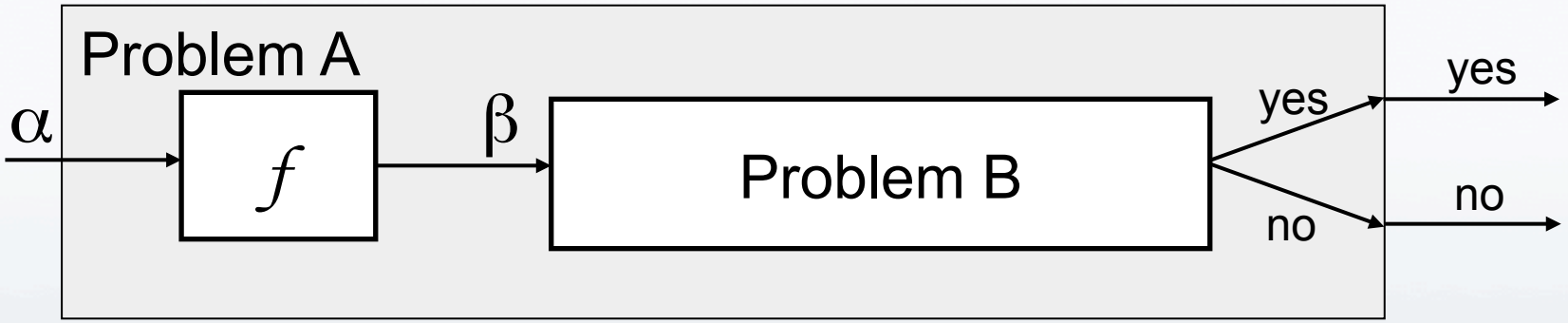


If $A \leq_p B$ and $B \in P$, then $A \in P$

Proof: The following algorithm that solves A is polinomial 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

Implications of Reduction



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

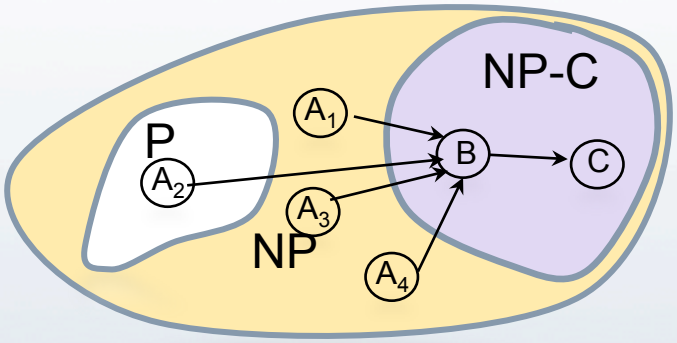
Proof: by contradiction
Let $A \leq_p B$ and $A \notin P$, but $B \in P$
$A \leq_p B$ and $B \in P$, then $A \in P$ (see previous page)
contradiction

NP-Completeness

Theorem: If B is NP-complete, and

- $C \in NP$
- $B \leq_p C$

then C is NP-complete.



In other words, we can prove a new problem is NP-complete by reducing some other NP-complete problem to it.

Proof: Let A be any problem in NP. Since B is NP-complete, $A \leq_p B$. By assumption, $B \leq_p C$. Therefore: $A \leq_p B \leq_p C$, which implies by transitivity $A \leq_p C$

NP-Completeness

Remarks:

- B∈NP-Complete is still in NP-Complete after introduction of the new problem C ∈ NP, that is $C \leq_p B$. $C \leq_p$SAT by Cook-Levin Theorem and SAT $\leq_p$ B because B was proved to be NP-complete previously, therefore $C \leq_p B$
- If any NP-Complete problem can be solved in polynomial time ⇒ then P = NP.
- If $B \leq_p C$ where B∈NP-Complete, but C is not NP (i.e. only second condition is satisfied in the theorem) then C is NP-hard (by definition).

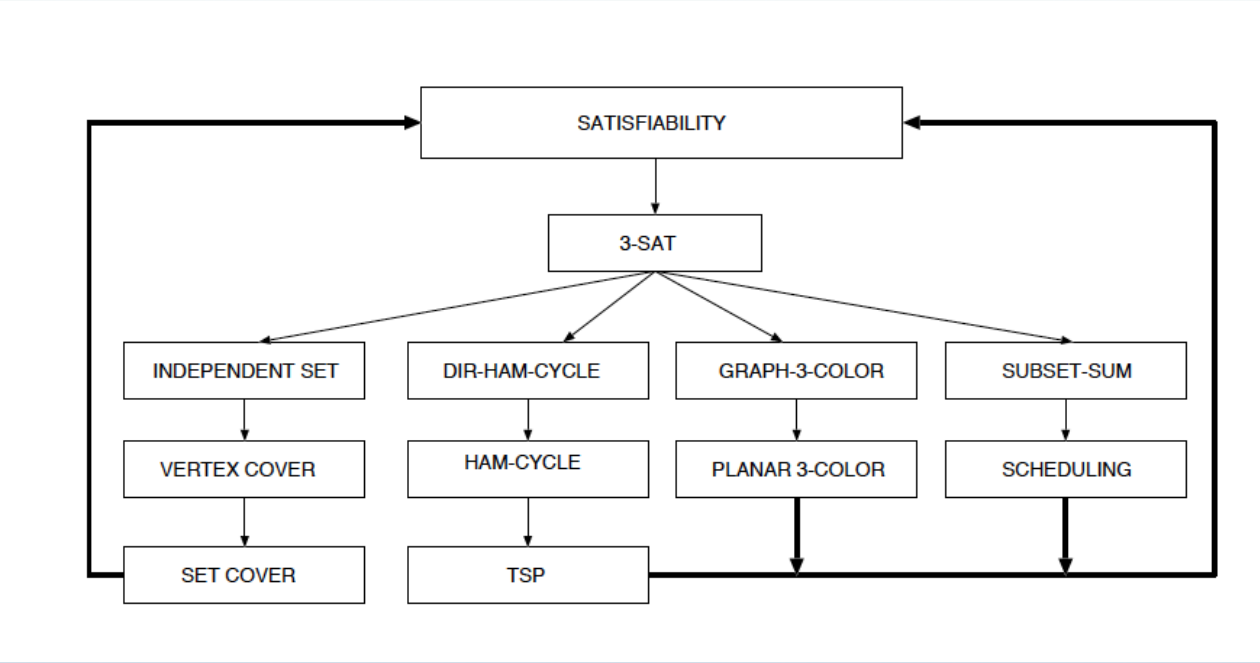
Proving NP-Completeness In Practice

In order to prove that a problem C is in NP

- Show that **one known** NP-Complete problem can be reduced to C in polynomial time
- No need to check that **all** NP-Complete problems are reducible to C

Proving NP-Completeness In Practice

Observation: All problems in figure below are NP-complete and polynomially reduce to one another!



Reduction Graph

Why discussion on NP-Completeness

If a problem is proved to be NP-Complete, a good evidence for its intractability (hardness).

Do not waste time on trying to find efficient algorithm for it

Instead, focus on designing an approximate algorithm or a solution for a special case of the problem

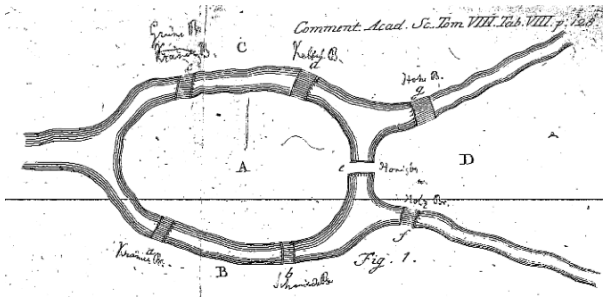
Some problems looks very easy on the surface, but in fact, is hard (NP-C).

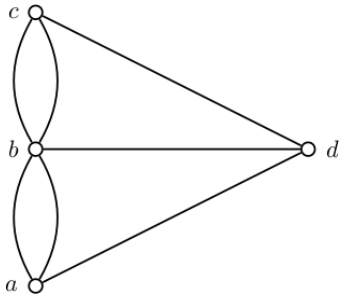
P & NP-Complete Problems

- **Shortest simple path**
  - Given a graph G = (V, E) find a **shortest** path from a source to all other vertices
  - Polynomial solution: See chapter on graphs
- **Longest simple path**
  - Given a graph G = (V, E) find a **longest** path from a source to all other vertices
  - **Decision version:** is there a path longer than k
  - NP-complete

P and NP Problems

- **Euler tour**
  - G = (V, E) a connected, directed graph find a cycle that traverses each edge of G exactly once (may visit a vertex multiple times)
  - Polynomial solution O(E)





Seven bridges of Königsberg. Euler was asked is it possible to cross all of these bridges while only crossing them only once each?
In 1735, Euler proved his answer by modelling the seven bridges of Königsberg in a diagram of four dots connected by lines.

P and NP Problems

- **Hamiltonian cycle**
  - G = (V, E) a connected graph find a cycle that visits each vertex of G exactly once
  - NP-complete
- **Traveling salesman**
  - G = (V, E) a connected graph with weighted edges, find a cycle that visits each vertex of G exactly once and its length is minimum
  - NP_hard (Decision version NP_complete)
  - Note that Hamiltonian cyle is a special case of traveling salesman where all the edges has weight 1.

SAT, CNF, 3CNF

CNF: A Boolean formula is in **conjunctive normal form** (**CNF)**, if it is an AND of clauses, each of which is an OR of literals (i.e. it is product of sums normal form)

- Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$

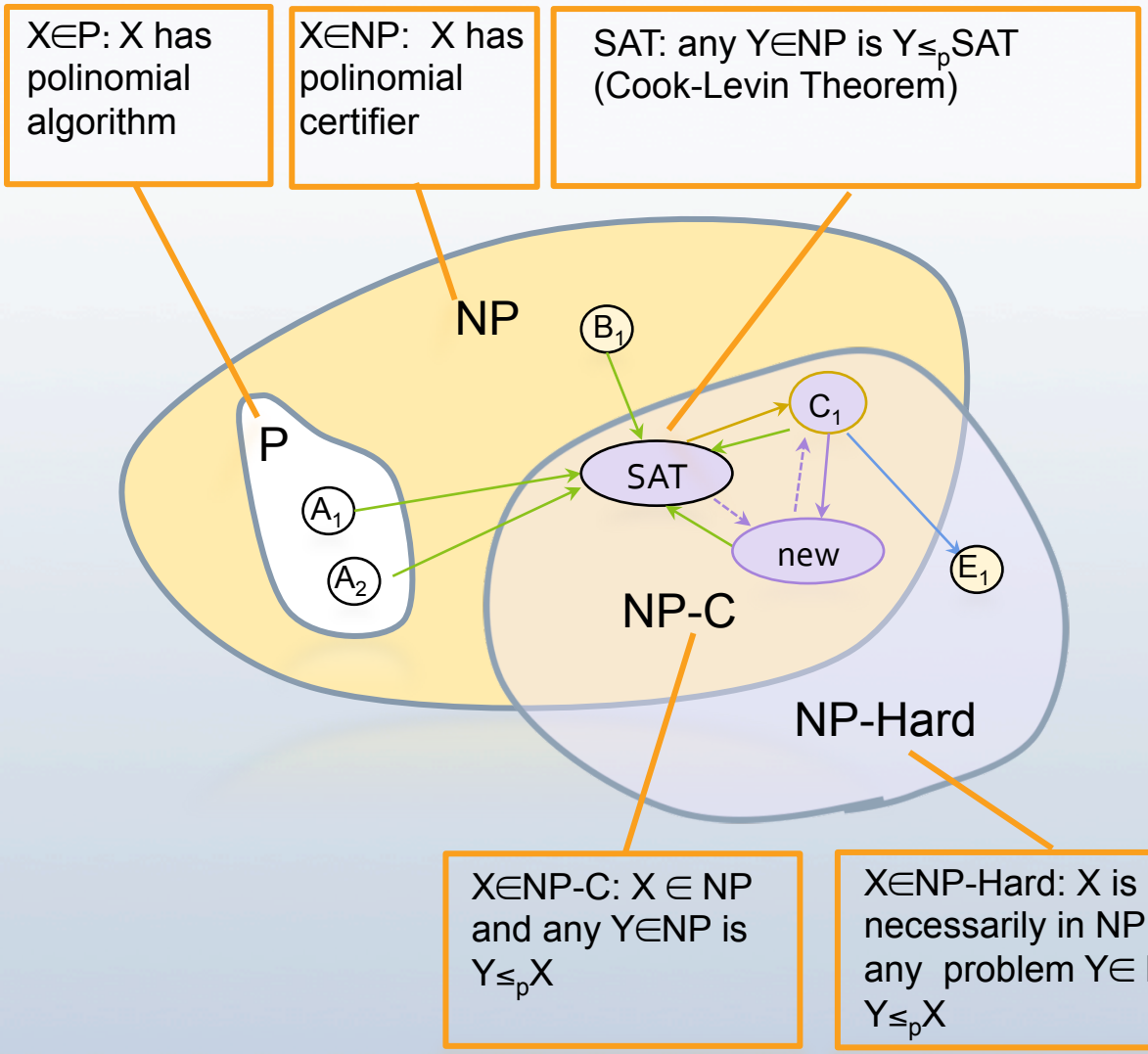
3-CNF: each clause has exactly 3 distinct literals

- Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$

Notice that the expression is true if at least one literal in each clause is true

- The first problem proved to be NP-complete was SAT .
- CNF problem is a special case of SAT problem. Even if the boolean expression is in CNF (Conjunctive Normal Form,) the problem is still NP-complete.
- 3-CNF is also proved to be NP-complete. (3-CNF is also called 3-SAT in literature)
- Interestingly enough, **2-CNF** is in P!

Computational Complexity Summary



$\leq_p$ relations:

- by Cook-Levine theorem
- introduced when a problem X is shown to be NP-C
- by transitivity
- introduced when a problem X is shown to be NP Hard