CSE 421
Introduction to Algorithms
Winter 2004

NP-Completeness

(Chapter 11)

Some Problems

- Independent-Set:
  - Given a graph G=(V,E) and an integer k, is there a subset U of V with |U| ≥ k such that no two vertices in U are joined by an edge.
- Clique:
  - Given a graph G=(V,E) and an integer k, is there a subset U of V with |U| ≥ k such that every pair of vertices in U is joined by an edge.

Some More Problems

- Hamilton Tour:
  - Given a graph G=(V,E) is there a simple cycle of length |V|, i.e.traversing each vertex once.
- Euler Tour:
  - Given a graph G=(V,E) is there a cycle traversing each edge once.
- TSP:
  - Given a weighted graph G=(V,E,w) and an integer k, is there a Hamilton tour of G with total weight ≤ k.

Satisfiability

- Boolean variables $x_1,...,x_n$
  - taking values in {0,1}. 0=false, 1=true
- Literals
  - x_i or $\neg x_i$ for i=1,...,n
- Clause
  - a logical OR of one or more literals
  - e.g. $(x_1 \vee \neg x_3 \vee x_7 \vee x_{12})$
- CNF formula
  - a logical AND of a bunch of clauses

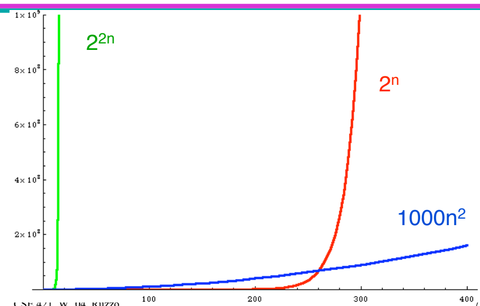
Satisfiability

- CNF formula example
  - (x1 ∨ ¬x3 ∨ x7 ∨ x12) ∧ (x2 ∨ ¬x4 ∨ x7 ∨ x5)
- If there is some assignment of 0's and 1's to the variables that makes it true then we say the formula is satisfiable
  - the one above is, the following isn't
  - x1 ∧ (¬x1 ∨ x2) ∧ (¬x2 ∨ x3) ∧ ¬x3
- Satisfiability: Given a CNF formula F, is it satisfiable?

Some History

- 1930's
  - What is (is not) computable
- 1960/70's
  - What is (is not) *feasibly* computable
  - Goal – a (largely) technology independent theory of time required by algorithms
  - Key modeling assumptions/approximations
    - Asymptotic (Big-O), worst case is revealing
    - Polynomial, exponential time – qualitatively different

Polynomial vs Exponential Growth



Another view of Poly vs Exp

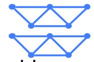
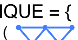
Next year's computer will be 2x faster. If I can solve problem of size N_0 today, how large a problem can I solve in the same time next year?

Complexity	Increase	E.g. $T=10^{12}$	
$O(n)$	$n_0 \rightarrow 2n_0$	10^{12}	2×10^{12}
$O(n^2)$	$n_0 \rightarrow \sqrt{2}\, n_0$	10^6	1.4×10^6
$O(n^3)$	$n_0 \rightarrow \sqrt[3]{2}\, n_0$	10^4	1.25×10^4
$2^{n/10}$	$n_0 \rightarrow n_0+10$	400	410
2^n	$n_0 \rightarrow n_0+1$	40	41

Polynomial versus exponential

- We'll say any algorithm whose run-time is
  - polynomial is good
  - bigger than polynomial is *bad*
- Note – of course there are exceptions:
  - n^{100} is bigger than $(1.001)^n$ for most practical values of n but usually such run-times don't show up
  - There are algorithms that have run-times like $O(2^{n/22})$ and these may be useful for small input sizes, but they're not too common either

Some Convenient Technicalities

- "Problem" – the general case
  - Ex: The Clique Problem: Given a graph G and an integer k, does G contain a k-clique?
- "Problem Instance" – the specific cases
  - Ex: Does  contain a 4-clique? (no)
  - Ex: Does  contain a 3-clique? (yes)
- Decision Problems – Just Yes/No answer
- Problems as Sets of "Yes" Instances
  - Ex: CLIQUE = { (G,k) | G contains a k-clique }
    - E.g., ( , 4) $\notin$ CLIQUE
    - E.g., ( , 3) $\in$ CLIQUE

Decision problems

- Computational complexity usually analyzed using decision problems
  - answer is just 1 or 0 (yes or no).
- Why?
  - much simpler to deal with
  - *deciding* whether G has a k-clique, is certainly no harder than *finding* a k-clique in G, so a *lower* bound on deciding is also a lower bound on finding
  - Less important, but if you have a good decider, you can often use it to get a good finder. (Ex.: does G still have a k-clique after I remove this vertex?)

Decision problem as a Language-recognition problem

- Let U be the set of all possible inputs to the decision problem.
- $L \subseteq U$ = the set of all inputs for which the answer to the problem is **yes**.
- We call L the **language** corresponding to the problem. (problem = language)
- The decision problem is thus:
  - to recognize whether or not a given input belongs to L = the language recognition problem.

Computational Complexity

- Classify problems according to the amount of computational resources used by the best algorithms that solve them
- Recall:
  - worst-case running time of an algorithm
    - max # steps algorithm takes on any input of size n
- Define:
  - TIME(f(n)) to be the set of all decision problems solved by algorithms having worst-case running time O(f(n))

Polynomial time

- Define P (polynomial-time) to be
  - the set of all decision problems solvable by algorithms whose worst-case running time is bounded by some polynomial in the input size.
- $P = \bigcup_{k\geq 0} \mathrm{TIME}(n^k)$

The class P

Definition: P = set of (decision) problems solvable by computers in polynomial time.

i.e. $T(n) = O(n^k)$ for some k.

- These problems are sometimes called **tractable** problems.

Examples: sorting, SCC, matching, max flow, shortest path, MST *– all of 421 up to now except Stamps/Knapsack/Partition*

Beyond P?

- There are many natural, practical problems for which we don't know any polynomial-time algorithms
- e.g. decisionTSP:
  - Given a weighted graph G and an integer k, does there exist a tour that visits all vertices in G having total weight at most k?

Solving TSP given a solution to decisionTSP

- Use binary search and several calls to decisionTSP to figure out what the exact total weight of the shortest tour is.
  - Upper and lower bounds to start are n times largest and smallest weights of edges, respectively
  - Call W the weight of the shortest tour.
- Now figure out which edges are in the tour
  - For each edge e in the graph in turn, remove e and see if there is a tour of weight at most W using decisionTSP
    - if not then e must be in the tour so put it back

More History – As of 1970

- Many of the above problems had been studied for decades
- All had real, practical applications
- *None* had poly time algorithms; exponential was best known
- But, it turns out they all have a very deep similarity under the skin

Some Problem Pairs

- Euler Tour
- 2-SAT
- Min Cut
- Shortest Path

- Hamilton Tour
- 3-SAT
- Max Cut
- Longest Path

Some Problem Pairs

- Euler Tour
- 2-SAT
- Min Cut
- Shortest Path

- Hamilton Tour
- 3-SAT
- Max Cut
- Longest Path

Superficially different; similar computationally

Similar pairs; seemingly different computationally

Common property of these problems

- There is a special piece of information, a short hint or proof, that allows you to efficiently verify (in polynomial-time) that the YES answer is correct. This hint might be very hard to find

- e.g.
  - DecisionTSP: the tour itself,
  - Independent-Set, Clique: the set U
  - Satisfiability: an assignment that makes F true.

The complexity class NP

NP consists of all decision problems where

- You can verify the YES answers efficiently (in polynomial time) given a short (polynomial-size) hint

And

- No hint can fool your polynomial time verifier into saying YES for a NO instance
- (implausible for all exponential time problems)

More Precise Definition of NP

- A decision problem is in NP iff there is a polynomial time procedure v(.,.), and an integer k such that
  - for every YES problem instance x there is a hint h with $|h| \le |x|^k$ such that v(x,h) = YES

  and

  - for every NO problem instance x there is *no* hint h with $|h| \le |x|^k$ such that v(x,h) = YES
- "Hints" sometimes called "Certificates"

Is it correct?

- For every x = (G,k) such that G contains a k-clique, there is a hint h that will cause v(x,h) to say YES, namely h = a list of the vertices in such a k-clique

and

- No hint can fool v into saying yes if either x isn't well-formed (the uninteresting case) or if x = (G,k) but G does not have any cliques of size k (the interesting case)

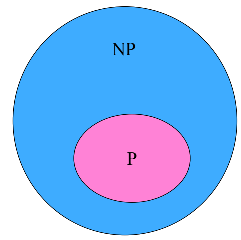
Keys to showing that a problem is in NP

- What's the output? (must be YES/NO)
- What's the input? Which are YES?
- For every given YES input, is there a hint that would help?
  - OK if some inputs need no hint
- For any given NO input, is there a hint that would trick you?

Complexity Classes

NP = Polynomial-time **verifiable**

P = Polynomial-time **solvable**



Example: CLIQUE is in NP

```
procedure v(x,h)
  if
    x is a well-formed representation of  a graph G =
    (V, E) and an integer k,
  and
    h is a well-formed representation of a k vertex
    subset U of V,
  and
    U is a clique in G,
  then output "YES"
  else output "I'm unconvinced"
```


Alternative Definition: NP = Nondeterministic P Time

- Imagine a nondeterministic algorithm: read input, compute, make nondeterministic choices, …, eventually arrive at “Accept” or “Quit” state.
- The language accepted = those inputs for which *some* (nondeterministically chosen) computation sequence leads to “Accept”
- NB: sequence ending in “Quit” does *not* mean input is rejected; only reject if *all* lead to “Quit.”

Equivalence of Definitions

- “hint” ⊆ “nondet”: nondeterministically guess the hint, then verify it determnistically
- “nondet” ⊆ “hint” : verify by running the nondet algorithm, using successive bits of the hint to determine the successive nondet choices to follow.

A problem NOT in NP; 2 bogus proofs to the contrary

- EEXP = {(p,x) | program p accepts input x in $< 2^{2^{|x|}}$ steps }

NON Theorem: EEXP in NP
- “Proof” 1: Hint = step-by-step trace of the computation of p on x; verify step-by-step
- “Proof” 2: nondeterministically guess whether accepts x, and accept if so.

Solving NP problems without hints/nondeterminism

- The only obvious algorithm for most of these problems is brute force:
  - try all possible hints and check each one to see if it works.
  - *Exponential* time:
    - 2^n truth assignments for n variables
    - n! possible TSP tours of n vertices
    - $\binom{n}{k}$ possible k element subsets of n vertices
    - etc.

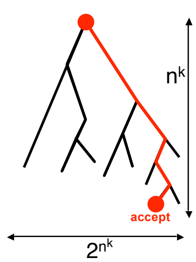
Problems in P can also be verified in polynomial-time

Shortest Path: Given a graph *G* with edge lengths, is there a path from *s* to *t* of length $\leq k$?
Verify: Given a path from *s* to *t*, is its length $\leq k$?

Small Spanning Tree: Given a weighted undirected graph *G*, is there a spanning tree of weight $\leq k$?
Verify: Given a spanning tree, is its weight $\leq k$?

P vs NP vs Exponential Time

- Theorem: Every problem in NP can be solved deterministically in exponential time
- Proof: the nondeterministic algorithm makes only n^k nd-choices. Try all 2^{n^k} possibilities; if any succeed, accept; if all fail, reject.

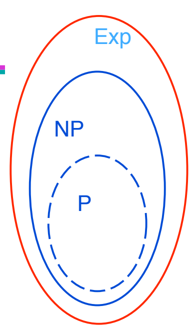


What We Know

- Nobody knows if all problems in NP can be done in polynomial time, i.e. does P=NP?
  - one of the most important open questions in all of science.
  - huge practical implications
- Every problem in P is in NP
  - one doesn't even need a hint for problems in P so just ignore any hint you are given
- Every problem in NP is in exponential time

P and NP

- Every problem in P is in NP
  - one doesn't even need a hint for problems in P so just ignore any hint you are given
  - Equivalently, a "nondet" algorithm doesn't need to use nondeterminism
- Every problem in NP is in exponential time



P vs NP

- Theory
  - P = NP ?
  - Open Problem!
  - I bet against it
- Practice
  - Many interesting, useful, natural, well-studied problems known to be NP-complete
  - With rare exceptions, no one routinely succeeds in finding exact solutions to large, arbitrary instances

More Connections

- Some Examples in NP
  - Satisfiability
  - Independent-Set
  - Clique
  - Vertex Cover
- All hard to solve; hints seem to help on all
- Very surprising fact:
  - Fast solution to *any* gives fast solution to *all!*

Nondeterminism

- A **nondeterministic algorithm** has all the "regular" operations of any other algorithm available to it.
- *In addition*, it has a powerful primitive, the **nd-choice primitive**.
- The **nd-choice primitive** is associated with a fixed number of choices, such that each choice causes the algorithm to follow a different computation path.

Nondeterminism (cont.)

- A **nondeterministic algorithm** consists of an interleaving of regular deterministic steps and uses of the **nd-choice primitive**.
- Definition: the algorithm accepts a language L if and only if
  - It has at least one "good" (accepting) sequence of choices for every $x \in L$, and
  - For all $x \notin L$, it reaches a reject outcome on **all** paths.

The class NP-complete

We are pretty sure that no problem in NP – P can be solved in polynomial time.

Non-Definition: NP-complete = the **hardest** problems in the class NP. (Formal definition later.)

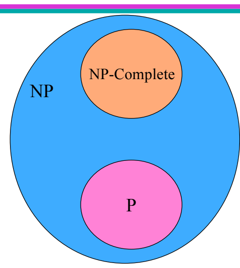
Interesting fact: If any one NP-complete problem could be solved in polynomial time, then ***all*** NP-complete problems could be solved in polynomial time.

Complexity Classes

NP = Poly-time **verifiable**

P = Poly-time **solvable**

NP-Complete = "Hardest" problems in **NP**



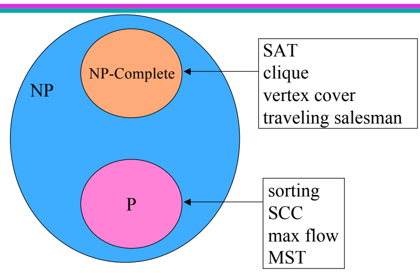
The class NP-complete (cont.)

Thousands of important problems have been shown to be NP-complete.

Fact (Dogma): The general belief is that there is no efficient algorithm for any **NP-complete** problem, but no proof of that belief is known.

Examples: SAT, clique, vertex cover, Hamiltonian cycle, TSP, bin packing.

Complexity Classes of Problems



Does P = NP?

- This is an open question.
- To show that **P = NP**, we have to show that *every* problem that belongs to NP can be solved by a polynomial time deterministic algorithm.
- No one has shown this yet.
- (It seems unlikely to be true.)

Is all of this useful for anything??!?

Earlier in this class we learned techniques for solving problems in **P**.

Question: Do we just throw up our hands if we come across a problem we suspect **not to be in P**?

Dealing with NP-complete Problems

What if I think my problem is <u>not in P</u>?

<u>Here is what you might do</u>:

1) Prove your problem is **NP-hard** or **-complete**
   (a common, but not guaranteed outcome)
2) Come up with an algorithm to solve the problem **usually** or **approximately**.

Reductions: a useful tool

Definition: To **reduce** A to B means to figure out how to solve A, given a subroutine solving B.

Example: reduce MEDIAN to SORT
  Solution: sort, then select (n/2) th

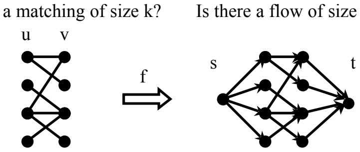
Example: reduce SORT to FIND_MAX
  Solution: FIND_MAX, remove it, repeat

Example: reduce MEDIAN to FIND_MAX
  Solution: transitivity: compose solutions above.

More Examples of reductions

Example:
  reduce BIPARTITE_MATCHING to MAX_FLOW

Is there a matching of size k? Is there a flow of size k?



All capacities = 1

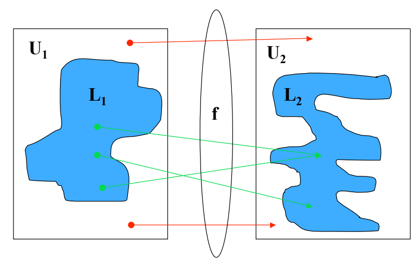
Polynomial-Time Reductions

Definition: Let L_1 and L_2 be two languages from the input spaces U_1 and U_2.

We say that L_1 is **polynomially reducible** to L_2 if there exists a polynomial-time algorithm f that converts each input $u_1 \in U_1$ to another input $u_2 \in U_2$ such that $u_1 \in L_1$ iff $u_2 \in L_2$.

$$u_1 \in L_1 \Leftrightarrow f(u_1) \in L_2$$

Polynomial-time Reduction from language L_1 to language L_2 via reduction function f.



$$u_1 \in L_1 \Leftrightarrow f(u_1) \in L_2$$

Polynomial-Time Reductions (cont.)

Define: $A \le_p B$ *"A is polynomial-time reducible to B", iff there is a polynomial-time computable function f such that:* $x \in A \Leftrightarrow f(x) \in B$

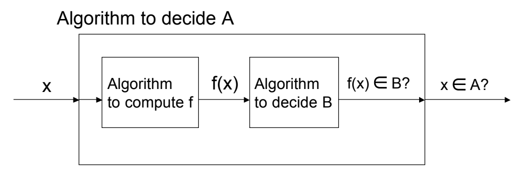
"complexity of A" $\le$ *"complexity of B" + "complexity of f"*

(1) $A \le_p B$ and $B \in P \Rightarrow A \in P$

(2) $A \le_p B$ and $A \notin P \Rightarrow B \notin P$

(3) $A \le_p B$ and $B \le_p C \Rightarrow A \le_p C$ (transitivity)

Using an Algorithm for *B* to Decide *A*



"If $A \le_p B$, and we can solve B in polynomial time, then we can solve A in polynomial time also."

Ex: suppose f takes $O(n^3)$ and algorithm for B takes $O(n^2)$. How long does the above algorithm for A take?

Definition of NP-Completeness

Definition: Problem *B* is **NP-hard** if *every* problem in NP is polynomially reducible to *B*.

Definition: Problem *B* is **NP-complete** if:

(1) *B* belongs to NP, and

(2) *B* is NP-hard.

Proving a problem is NP-complete

- Technically,for condition (2) we have to show that **every** problem in NP is reducible to B. (yikes!) This sounds like a lot of work.
- For the **very first NP-complete problem** (SAT) this had to be proved directly.
- However, once we have one NP-complete problem, then we don't have to do this every time.
- Why? Transitivity.

Re-stated Definition

Lemma 11.3: Problem *B* is **NP-complete** if:

(1) *B* belongs to NP, and

(2') *A* is polynomial-time reducible to *B*, for some problem *A* that is NP-complete.

That is, to show (2') given a new problem *B*, it is sufficient to show that SAT or any other NP-complete problem is polynomial-time reducible to *B*.

Usefulness of Transitivity

Now we only have to show $L' \leq_p L$, for some problem $L' \in$ ***NP-complete***, in order to show that **L** is NP-hard. Why is this equivalent?

1) Since $L' \in$ ***NP-complete****, we know that* L' is *NP-hard.* That is:

$$\forall L'' \in NP, \text{ we have } L'' \leq_p L'$$

2) If we show $L' \leq_p L$, then by transitivity we know that: $\forall L'' \in NP$, we have $L'' \leq_p L$.

Thus L is NP-hard.

The growth of the number of NP-complete problems

- Steve Cook (1971) showed that SAT was NP-complete.
- Richard Karp (1972) found 24 more NP-complete problems.
- Today there are thousands of known NP-complete problems.
  - Garey and Johnson (1979) is an excellent source of NP-complete problems.

SAT is NP-complete

Cook's theorem: **SAT is NP-complete**

Satisfiability (SAT)

A Boolean formula in conjunctive normal form (CNF) is **satisfiable** if there exists a truth assignment of 0's and 1's to its variables such that the value of the expression is 1. Example:

$$S=(x+y+\neg z)\cdot(\neg x+y+z)\cdot(\neg x+\neg y+\neg z)$$

Example above is satisfiable. (We an see this by setting x=1, y=1 and z=0.)

SAT is NP-complete

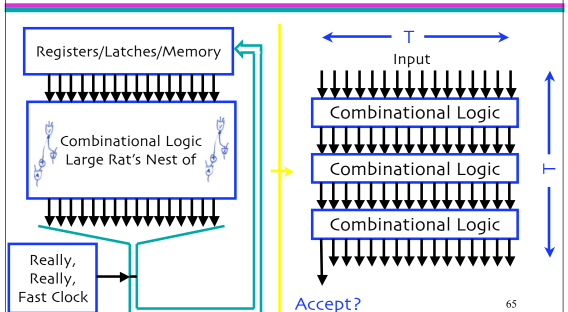
Rough idea of proof:

(1) **SAT is in NP** because we can guess a truth assignment and check that it satisfies the expression in polynomial time.

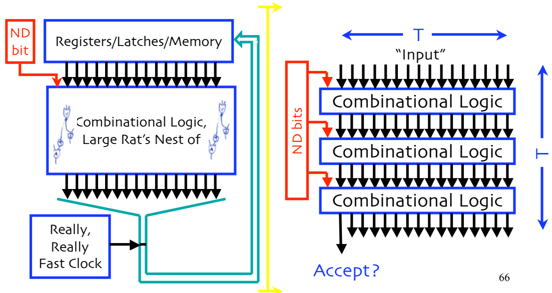
(2) **SAT is NP-hard** because …..

Cook proved it directly, but easier to see via an intermediate problem – **Circuit-SAT**

P Is Reducible To The Circuit Value Problem



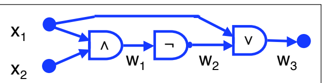
NP Is Reducible To The Circuit Satisfiability Problem



To Prove SAT is NP-complete

- Show it's *in* NP: Exercise
  (Hint: what's an easy-to-check certificate of satisfiability?)
- Pick a known NP-complete problem
  & reduce it to SAT
  - Gee, How about Circuit-SAT?
    Good idea; it's the only NP-complete problem we have so far
  - What we need:
    a fast, mechanical way to "simulate" a circuit by a formula

Circuit-SAT $\leq_p$ 3-SAT



$$\underbrace{(w_1 \Leftrightarrow (x_1 \wedge x_2))}\wedge(w_2 \Leftrightarrow (\neg w_1))\wedge(w_3 \Leftrightarrow (w_2 \vee x_1))\wedge w_3$$

Replace with 3-CNF Equivalent:

x_1	x_2	w_1	$(x_1 \wedge x_2)$	$\neg(w_1 \Leftrightarrow (x_1 \wedge x_2))$	
0	0	0	0	0	
0	0	1	0	1	$\leftarrow \neg x_1 \wedge \neg x_2 \wedge w_1$
0	1	0	0	0	
0	1	1	0	1	$\leftarrow \neg x_1 \wedge x_2 \wedge w_1$
1	0	0	0	0	
1	0	1	0	1	$\leftarrow x_1 \wedge \neg x_2 \wedge w_1$
1	1	0	1	1	$\leftarrow x_1 \wedge \neg x_2 \wedge \neg w_1$
1	1	1	1	0	

$$(x_1 \vee x_2 \vee \neg w_1)\wedge(x_1 \vee \neg x_2 \vee \neg w_1)\wedge(\neg x_1 \vee x_2 \vee \neg w_1)\wedge(\neg x_1 \vee \neg x_2 \vee w_1)$$

¬clause → Truth Table → DNF; DeMorgan → CNF

Correctness of "Circuit-SAT $\leq_p$ 3-SAT"

Summary of reduction function f:
Given circuit, add variable for every gate's value, build clause for each gate, satisfiable iff gate value variable is appropriate logical function of its input variables, convert each to CNF via standard truth-table construction. Output conjunction of all, plus output variable. Note: f does *not* know whether circuit or formula are satisfiable or not; does *not* try to find satisfying assignment.

Correctness:

1. Show f poly time computable: A key point is that formula size is linear in circuit size; mapping basically straightforward.
2. Show c in Circuit-SAT iff f(c) in SAT:
   (⇒) Given an assignment to x_i's satisfying c, extend it to w_i's by evaluating the circuit on x_i's gate by gate. Show this satisfies f(c).
   (⇐) Given an assignment to x_i's & w_i's satisfying f(c), show x_i's satisfy c (with gate values given by w_i's).

How do you prove problem *A* is NP-complete?

1) **Prove *A* is in NP:** show that given a solution, it can be verified in polynomial time.

2) **Prove that *A* is NP-hard:**

a) Select a **known** NP-complete problem *B*.

b) Describe a polynomial time computable algorithm that computes a function *f*, **mapping *every* instance of *B* to an instance of *A*.** (that is: ***B*** $\leq_p$ ***A***)

c) Prove that if b is a ***yes***-instance of *B* then f(b) is a ***yes***-instance of *A*. Conversely, if f(b) is a ***yes***-instance of *A,* then b must be ***yes***-instance of *B*.

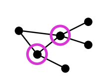
d) Prove that the algorithm computing ***f* runs in polynomial time**.

NP-complete problem: **Vertex Cover**

Input: Undirected graph *G = (V, E)*, integer *k*.

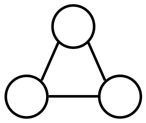
Output: True iff there is a subset *C* of *V* of size ≤ *k* such that every edge in *E* is incident to at least one vertex in *C*.

Example: Vertex cover of size ≤ 2.

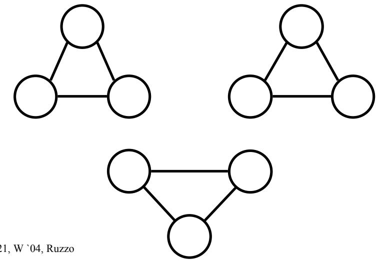


In NP? Exercise

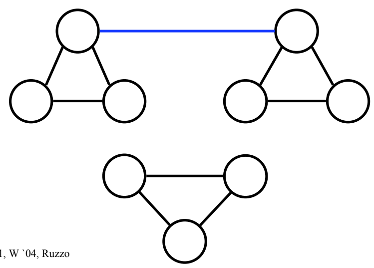
3SAT $\leq_p$ VertexCover



3SAT $\leq_p$ VertexCover

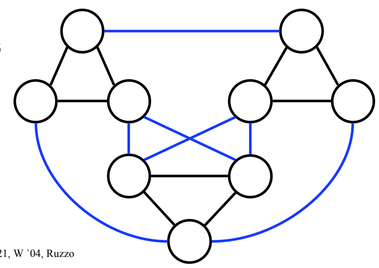


3SAT $\leq_p$ VertexCover



3SAT $\leq_p$ VertexCover

k=6



3SAT $\leq_p$ VertexCover

f (

3-SAT Instance:
- Variables: $x_1, x_2, \ldots$
- Literals: $y_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, $1 \leq i \leq q$
- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

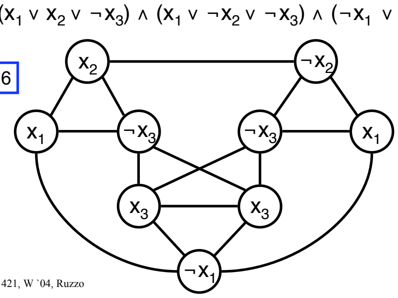
) =

VertexCover Instance:
- $k = 2q$
- $G = (V, E)$
- $V = \{ [i,j] \mid 1 \leq i \leq q, 1 \leq j \leq 3 \}$
- $E = \{ ([i,j], [k,l]) \mid i = k \text{ or } y_{ij} = \neg y_{kl} \}$

3SAT $\leq_p$ VertexCover

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

k=6



Correctness of "3-SAT $\leq_p$ VertexCover"

Summary of reduction function f:
Given formula, make graph G with one group per clause, one node per literal. Connect each to all nodes in *same* group, *plus* complementary literals (x, ¬x). Output graph G plus integer k = 2 * number of clauses.
Note: f does *not* know whether formula is satisfiable or not; does *not* know if G has k-cover; does *not* try to find satisfying assignment or cover.

Correctness:

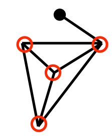
1. Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
2. Show c in 3-SAT iff f(c)=(G,k) in VertexCover:
(⇒) Given an assignment satisfying c, pick one true literal per clause. Add *other* 2 nodes of each triangle to cover. Show it is a cover: 2 per triangle cover triangle edges; only true literals uncovered, so at least one end of every (x, ¬x) edge is covered.
(⇐) Given a k-vertex cover in G, *uncovered* labels define a valid (perhaps partial) truth assignment since no (x, ¬x) pair uncovered. It satisfies c since there is one uncovered node in each clause triangle (else some other clause triangle has > 1 uncovered node, hence an uncovered edge.)

NP-complete problem: **Clique**

Input: Undirected graph $G = (V, E)$, integer k.

Output: True iff there is a subset C of V of size $\geq k$ such that all vertices in C are connected to all other vertices in C.

Example: Clique of size ≥ 4



In NP? Exercise

3SAT $\leq_p$ Clique

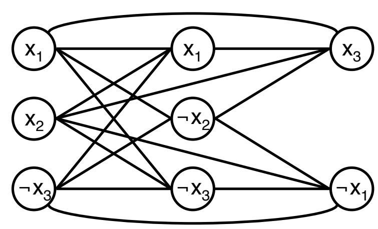
f (3-SAT Instance:
- Variables: $x_1, x_2, \ldots$
- Literals: $y_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, $1 \leq i \leq q$
- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

) =

VertexCover Instance:
- K = q
- G = (V, E)
- V = { [i,j] | $1 \leq i \leq q$, $1 \leq j \leq 3$ }
- E = { ([i,j], [k,l]) | $i \neq k$ and $y_{ij} \neq \neg y_{kl}$ }

3SAT $\leq_p$ **Clique**

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



k=3

Correctness of "3-SAT $\leq_p$ Clique"

Summary of reduction function f:
Given formula, make graph G with column of nodes per clause, one node per literal. Connect each to all nodes in *other* columns, *except* complementary literals (x, ¬x). Output graph G plus integer k = number of clauses. Note: f does *not* know whether formula is satisfiable or not; does *not* know if G has k-clique; does *not* try to find satisfying assignment or clique.

Correctness:

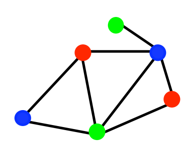
1. Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
2. Show c in 3-SAT iff f(c)=(G,k) in Clique:
(⇒) Given an assignment satisfying c, pick one true literal per clause. Show corresponding nodes in G are k-clique.
(⇐) Given a k-clique in G, clique labels define a truth assignment; show it satisfies c. Note: literals in a clique are a valid truth assignment [no "(x, ¬x)" edges] & k nodes must be 1 per column, [no edges within columns].

NP-complete problem: **3-Coloring**

Input: An undirected graph G=(V,E).

Output: True iff there is an assignment of at most 3 colors to the vertices in G such that no two adjacent vertices have the same color.

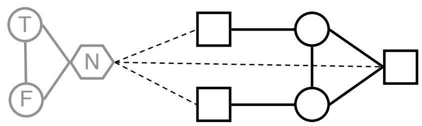
Example:



In NP? Exercise

A 3-Coloring Gadget:

In what ways can this be 3-colored?

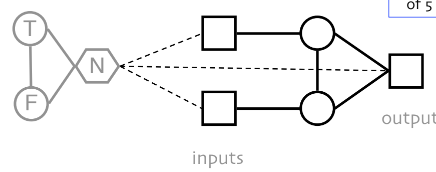


A 3-Coloring Gadget: "Sort of an OR gate"

(1) if any input is T, the output can be T

(2) if output is T, some input must be T

Exercise: find all colorings of 5 nodes



3SAT $\leq_p$ 3Color

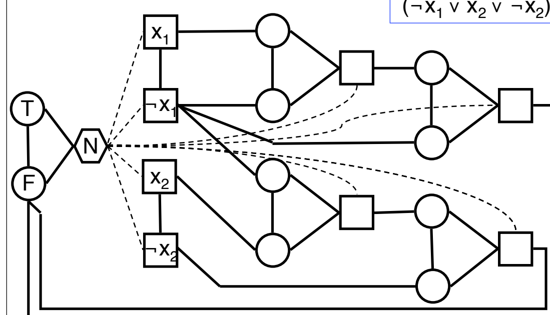
$f\Bigg($ 3-SAT Instance:
- Variables: $x_1, x_2, \ldots$
- Literals: $y_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, $1 \leq i \leq q$
- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

$\Bigg) =$

3Color Instance:
- G = (V, E)
- 6 q + 2 n + 3 vertices
- 13 q + 3 n + 3 edges
- (See Example for details)

3SAT $\leq_p$ 3Color Example

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



Correctness of "3-SAT $\leq_p$ 3Coloring"

Summary of reduction function f:
Given formula, make G with T-F-N triangle, 1 pair of literal nodes per variable, 2 "or" gadgets per clause, connected as in example.
Note: again, f does *not* know or construct satisfying assignment or coloring.

Correctness:

1. Show f poly time computable: A key point is that graph size is polynomial in formula size; graph looks messy, but pattern is basically straightforward.
2. Show c in 3-SAT iff f(c) is 3-colorable:
   ($\Rightarrow$) Given an assignment satisfying c, color literals T/F as per assignment; can color "or" gadgets so output nodes are T since each clause is satisfied.
   ($\Leftarrow$) Given a 3-coloring of f(c), name colors T-N-F as in example. All square nodes are T or F (since all adjacent to N). Each variable pair (x_i, $\neg x_i$) must have complementary labels since they're adjacent. Define assignment based on colors of x_i's. Clause "output" nodes must be colored T since they're adjacent to both N & F. By fact noted earlier, output can be T only if at least one input is T, hence it is a satisfying assignment.

Common Errors in NP-completeness Proofs

- Backwards reductions
  E.g., Biconnectivity $\leq_p$ SAT is true, but not so useful. (XYZ $\leq_p$ SAT shows XYZ in NP, does *not* show that it's hard.)
- Slooow Reductions
  "Find a satisfying assignment, then output…"
- Half Reductions
  e.g. delete dashed edges in 3Color reduction. It's still true that "c satisfiable $\Rightarrow$ G is 3 colorable", but 3-colorings don't necessarily give good assignments

Coping with NP-Completeness

- Is your real problem a special subcase?
  - E.g. 3-SAT is NP-complete, but 2-SAT is not;
  - Ditto 3- vs 2-coloring
  - E.g. maybe you only need planar graphs, or degree 3 graphs, or …
- Guaranteed approximation good enough?
  - E.g. Euclidean TSP within 1.5 * Opt in poly time
- Clever exhaustive search, e.g. Branch & Bound
- Heuristics – usually a good approximation and/or usually fast

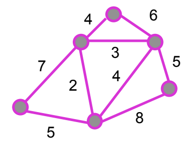
NP-complete problem: **TSP**

Input: An undirected graph G=(V,E) with integer edge weights, and an integer b.

Output: True iff there is a simple cycle in G passing through all vertices (once), with total cost ≤ b.

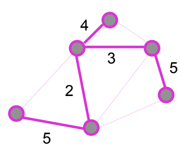
Example:

b = 34



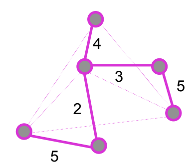
2x Approximation to EuclideanTSP

- A TSP tour visits all vertices, so contains a spanning tree, so TSP cost is > cost of min spanning tree.
- Find MST
- Double all edges
- Find Euler Tour
- Shortcut
- Cost of shortcut < ET = 2 * MST < 2 * TSP



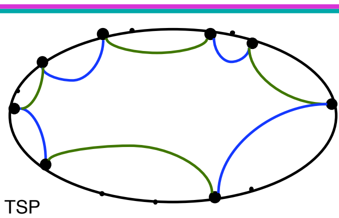
1.5x Approximation to EuclideanTSP

- Find MST
- Find min cost matching among odd-degree tree vertices
- Cost of matching ≤ TSP/2
- Find Euler Tour
- Shortcut
- Shortcut ≤ ET ≤ MST + TSP/2 < 1.5* TSP



Matching ≤ TSP/2

- Oval=TSP
- Big dots= odd tree nodes
- Blue, Green = 2 matchings
- Blue + Green ≤ TSP (by triangle inequality)
- So min matching ≤ TSP/2



Summary

- Big-O – good
- P – good
- Exp – bad
- Hints help? NP
- NP-hard, NP-complete – bad (I bet)
- To show NP-complete – reductions
- NP-complete = hopeless? – no, but you need to lower your expectations: heuristics & approximations.