NP-Completeness

Andreas Klappenecker

[based on slides by Prof. Welch]

Prelude: Informal Discussion

(Incidentally, we will never get very formal in this course)

Polynomial Time Algorithms

- Most of the algorithms we have seen so far run in time that is upper bounded by a polynomial in the input size
  - sorting: $O(n^2)$, $O(n \log n)$, …
  - matrix multiplication: $O(n^3)$, $O(n^{\log_2 7})$
  - graph algorithms: $O(V+E)$, $O(E \log V)$, …
- In fact, the running time of these algorithms are bounded by small polynomials.

Categorization of Problems

We will consider a computational problem tractable if and only if it can be solved in polynomial time.

Decision Problems and the class P

A computational problem with yes/no answer is called a decision problem.

We shall denote by P the class of all decision problems that are solvable in polynomial time.

Why Polynomial Time?

It is convenient to define decision problems to be tractable if they belong to the class P, since

- the class P is closed under composition.
- the class P becomes more or less independent of the computational model.

[Typically, computational models can be transformed into each other by polynomial time reductions.]

Of course, no one will consider a problem requiring an $\Omega(n^{100})$ algorithm as efficiently solvable. However, it seems that most problems in P that are interesting in practice can be solved fairly efficiently.

The Class NP

We shall denote by NP the class of all decision problems for which a candidate solution can be verified in polynomial time.

[We may not be able to find the solution, but we can verify the solution in polynomial time if someone is so kind to give us the solution.]

Sudoku

- The problem is given as an n^2 x n^2 array which is divided into blocks of n x n squares.
- Some array entries are filled with an integer in the range [1.. n^2].
- The goal is to complete the array such that each row, column, and block contains each integer from [1..n^2].

Sudoku

Problem

			4
3		2	
	1		3
4			

→

Solution

1	2	3	4
3	4	2	1
2	1	4	3
4	3	1	2

- Finding the solution might be difficult, but verifying the solution is easy.
- The Sudoku decision problem is whether a given Sudoku problem has a solution.

The Class NP

The decision problems in NP can be solved on a nondeterministic Turing machine in polynomial time. Thus, NP stands for nondeterministic polynomial time.

Obviously, the class P is a subset of NP.

NP does not stand for not-P. Why?

Verifying a Candidate Solution

- Difference between solving a problem and verifying a candidate solution:
- Solving a problem: is there a path in graph G from node u to node v with at most k edges?
- Verifying a candidate solution: is $v_0, v_1, \ldots, v_\ell$ a path in graph G from node u to node v with at most k edges?

Verifying a Candidate Solution

- A Hamiltonian cycle in an undirected graph is a cycle that visits every node exactly once.
- Solving a problem: Is there a Hamiltonian cycle in graph G?
- Verifying a candidate solution: Is v_0, v_1, …, v_ℓ a Hamiltonian cycle of graph G?

Verifying a Candidate Solution

- Intuitively it seems much harder (more time consuming) in some cases to solve a problem from scratch than to verify that a candidate solution actually solves the problem.
- If there are many candidate solutions to check, then even if each individual one is quick to check, overall it can take a long time

Verifying a Candidate Solution

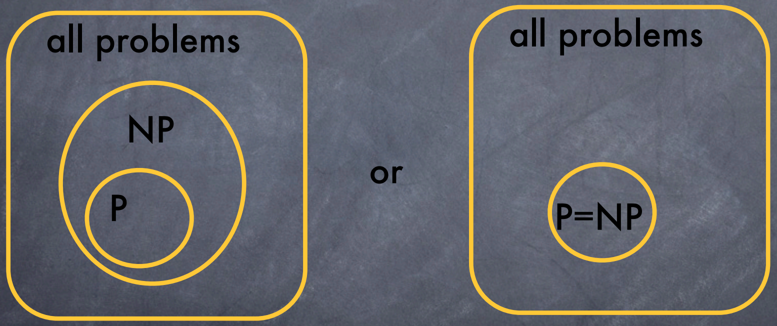
- Many practical problems in computer science, math, operations research, engineering, etc. are poly time verifiable but have no known poly time algorithm
  - Wikipedia lists problems in computational geometry, graph theory, network design, scheduling, databases, program optimization and more

P versus NP

P vs. NP

- Although poly-time verifiability seems like a weaker condition than poly time solvability, no one has been able to prove that it is weaker (i.e., describes a larger class of problems)
- So it is unknown whether P = NP.

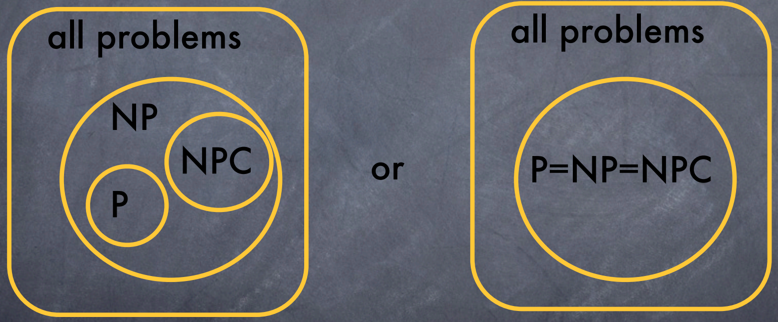
P and NP



NP-Complete Problems

- NP-complete problems is class of "hardest" problems in NP.
- If an NP-complete problem can be solved in polynomial time, then all problems in NP can be, and thus P = NP.

Possible Worlds



all problems

NP

NPC

P

or

all problems

P=NP=NPC

NPC = NP-complete

P = NP Question

- Open question since about 1970
- Great theoretical interest
- Great practical importance:
    - If your problem is NP-complete, then don't waste time looking for an efficient algorithm
    - Instead look for efficient approximations, heuristics, etc.

Decision Problems and Formal Languages

NP-Completeness Theory

As we have already mentioned, the theory is based considering decision problems.

Example:

-Does there exist a path from node u to node v in graph G with at most k edges.

- Instead of: What is the length of the shortest path from u to v? Or even: What is the shortest path from u to v?

Decision Problems

Why focus on decision problems?

- Solving the general problem is at least as hard as solving the decision problem version
- For many natural problems, we only need polynomial additional time to solve the general problem if we already have a solution to the decision problem
- We can use "language acceptance" notions

Languages and Decision

- Language: A set of strings over some alphabet
- Decision problem: A decision problem can be viewed as the formal language consisting of exactly those strings that encode YES instances of the problem
- What do we mean by encoding Yes instances?

Encodings

- Every abstract problem has to be represented somehow for the computer to work on it - ultimately a binary representation
- Consider the problem: "Is x prime?"
- Each input is a positive integer
- Common way to encode an integer is in binary
- Primes decision problem is {10,11,101,111,...} since 10 encodes 2, 11 encodes 3, 101 encodes 5, 111 encodes 7, etc.

More Complicated Encodings

- Suggest an encoding for the shortest path decision problem
- Represent G, u, v and k somehow in binary
- Decision problem is all encodings of a graph G, two nodes u and v, and an integer k such that G really does have a path from u to v of length at most k

Definition of P

- P is the set of all decision problems that can be computed in time $O(n^k)$, where n is the length of the input string and k is a constant
- "Computed" means there is an algorithm that correctly returns YES or NO whether the input string is in the language

Example of a Decision Problem

- "Given a graph G, nodes u and v, and integer k, is there a path in G from u to v with at most k edges?"
- Why is this a decision problem?
  - Has YES/NO answers
- We are glossing over the particular encoding (tedious but straightforward)
- Why is this problem in P?
  - Do BFS on G in polynomial time

Definition of NP

- NP = set of all decision problems for which a candidate solution can be verified in polynomial time
- Does *not* stand for "not polynomial"
  - in fact P is a subset of NP
- NP stands for "nondeterministic polynomial"
  - more info on this in CPSC 433

Example of a Decision Problem

- Decision problem: Is there a path in G from u to v of length at most k?
- Candidate solution: a sequence of nodes $v_0, v_1, \ldots, v_\ell$
- To verify:
  - check if $\ell \leq k$
  - check if $v_0 = u$ and $v_\ell = v$
  - check if each (v_i, v_{i+1}) is an edge of G

Example of a Decision Problem

- Decision problem: Does G have a Hamiltonian cycle?
- Candidate solution: a sequence of nodes $v_0, v_1, \ldots, v_\ell$
- To verify:
  - check if ℓ = number of nodes in G
  - check if $v_0 = v_\ell$ and there are no repeats in $v_0, v_1, \ldots, v_{\ell-1}$
  - check if each (v_i, v_{i+1}) is an edge of G

Going From Verifying to Solving

- for each candidate solution do
  - verify if the candidate really works
  - if so then return YES
- return NO

Difficult to use in practice, though, if number of candidate solutions is large

Number of Candidate Solutions

- "Is there a path from u to v in G of length at most k?": more than n! candidate solutions where n is the number of nodes in G
- "Does G have a Hamiltonian cycle?": n! candidate solutions

Trying to be Smarter

- For the length-k path problem, we can do better than the brute force approach of trying all possible sequences of nodes
  - use BFS
- For the Hamiltonian cycle problem, no one knows a way that is significantly faster than trying all possibilities
  - but no one has been able to prove that

Polynomial Reduction

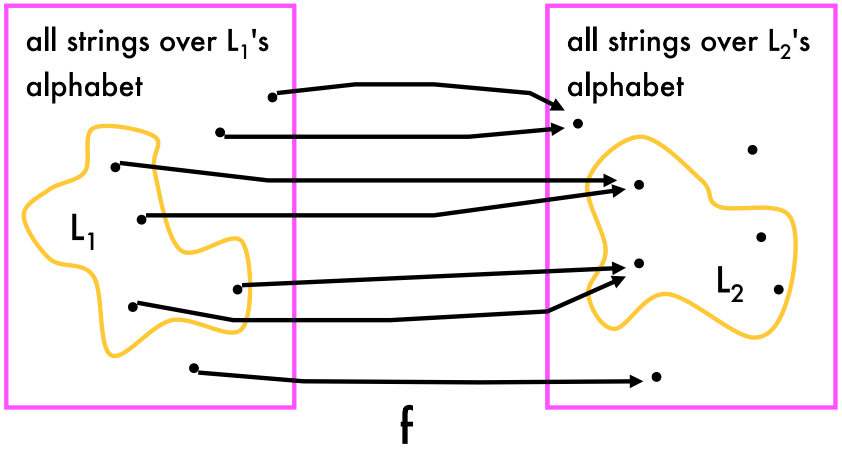
Polynomial Reduction

A polynomial reduction (or transformation) from language L_1 to language L_2 is a function f from strings over L_1's alphabet to strings over L_2's alphabet such that

(1) f is computable in polynomial time

(2) for all x, x is in L_1 if and only if f(x) is in L_2

Polynomial Reduction



Polynomial Reduction

- YES instances map to YES instances
- NO instances map to NO instances
- computable in polynomial time
- Notation: $L_1 \leq_p L_2$
- [Think: L_2 is at least as hard as L_1]

Polynomial Reduction Theorem

Theorem: If $L_1 \le_p L_2$ and L_2 is in P,

then L_1 is in P.

Proof: Let A_2 be a polynomial time algorithm for L_2. Here is a polynomial time algorithm A_1 for L_1.

- input: x
- compute f(x)
- run A_2 on input f(x)

$|x| = n$

takes p(n) time

takes q(p(n)) time

takes O(1) time

Implications

- Suppose that $L_1 \leq_p L_2$
- If there is a polynomial time algorithm for L_2, then there is a polynomial time algorithm for L_1.
- If there is no polynomial time algorithm for L_1, then there is no polynomial time algorithm for L_2.
- Note the asymmetry!

HC $\le_p$ TSP

Traveling Salesman Problem

- Given a set of cities, distances between all pairs of cities, and a bound B, does there exist a tour (sequence of cities to visit) that returns to the start and requires at most distance B to be traveled?
- TSP is in NP:
  - given a candidate solution (a tour), add up all the distances and check if total is at most B

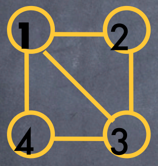
Example of Polynomial

- Theorem: HC (Hamiltonian circuit problem) $\leq_p$ TSP.
- Proof: Find a way to transform ("reduce") any HC input (G) into a TSP input (cities, distances, B) such that
  - the transformation takes polynomial time
  - the HC input is a YES instance (G has a HC) if and only if the TSP input constructed is a YES instance (has a tour that meets the bound).

The Reduction

- Given undirected graph G = (V,E) with m nodes, construct a TSP input like this:
  - set of m cities, labeled with names of nodes in V
  - distance between u and v is 1 if (u,v) is in E, and is 2 otherwise
  - bound B = m
- Why can this TSP input be constructed in time polynomial in the size of G?

Figure for Reduction



HC input

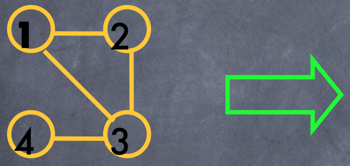
HC: 1,2,3,4,1

dist(1,2) = 1
dist(1,3) = 1
dist(1,4) = 1
dist(2,3) = 1
dist(2,4) = 2
dist(3,4) = 1
bound = 4

TSP input

tour w/ distance 4: 1,2,3,4,1

Figure for Reduction



dist(1,2) = 1
dist(1,3) = 1
dist(1,4) = 2
dist(2,3) = 1
dist(2,4) = 2
dist(3,4) = 1
bound = 4

HC input

TSP input

no HC

no tour w/ distance at most 4

Correctness of the Reduction

- Check that input G is in HC (has a Hamiltonian cycle) if and only if the input constructed is in TSP (has a tour of length at most m).
- => Suppose G has a Hamiltonian cycle v_1, v_2, …, v_m, v_1.
  - Then in the TSP input, v_1, v_2, …, v_m, v_1 is a tour (visits every city once and returns to the start) and its distance is $1 \cdot m = B$.

Correctness of the Reduction

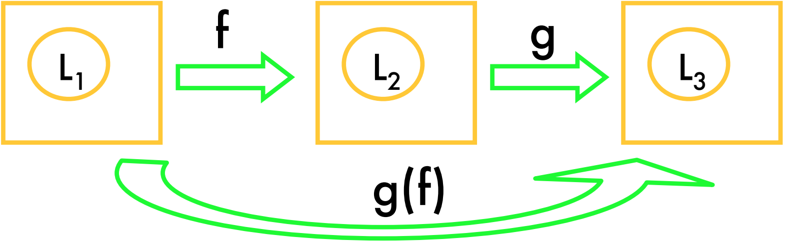
- <=: Suppose the TSP input constructed has a tour of total length at most m.
  - Since all distances are either 1 or 2, and there are m of them in the tour, all distances in the tour must be 1.
  - Thus each consecutive pair of cities in the tour correspond to an edge in G.
  - Thus the tour corresponds to a Hamiltonian cycle in G.

Implications:

- If there is a polynomial time algorithm for TSP, then there is a polynomial time algorithm for HC.
- If there is no polynomial time algorithm for HC, then there is no polynomial time algorithm TSP.
- Note the asymmetry!

Transitivity of Polynomial

- Theorem: If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.
- Proof:



NP-Completeness

Definition of NP-Complete

L is NP-complete if and only if

(1) L is in NP and

(2) for all L' in NP, L' $\leq_p$ L.

In other words, L is at least as hard as every language in NP.

Implication of NP-Completeness

Theorem: Suppose L is NP-complete.

(a) If there is a poly time algorithm for L, then P = NP.

(b) If there is no poly time algorithm for L, then there is no poly time algorithm for any NP-complete language.

Showing NP-Completeness

- How to show that a problem (language) L is NP-complete?
- Direct approach: Show
  - (1) L is in NP
  - (2) every other language in NP is polynomially reducible to L.
- Better approach: once we know some NP-complete problems, we can use reduction to show other problems are also NP-complete. How?

Showing NP-Completeness with

To show L is NP-complete:

- (1) Show L is in NP.
- (2.a) Choose an appropriate known NP-complete language L'.
- (2.b) Show L' $\leq_p$ L.

Why does this work? By transitivity: Since every language L'' in NP is polynomially reducible to L', L'' is also polynomially reducible to L.

The First NP-Complete Problem: Satisfiability - SAT

First NP-Complete Problem

How do we get started? Need to show via brute force that some problem is NP-complete.

- Logic problem "satisfiability" (or SAT).
- Given a boolean expression (collection of boolean variables connected with ANDs and ORs), is it satisfiable, i.e., is there a way to assign truth values to the variables so that the expression evaluates to TRUE?

Conjunctive Normal Form (CNF)

- boolean variables: take on values T or F
  - Ex: x, y
- literal: variable or negation of a variable
  - Ex: x, x
- clause: disjunction (OR) of several literals
  - Ex: $x \vee y \vee z \vee w$
- CNF formula: conjunction (AND) of several clauses
  - Ex: $(x \vee y) \wedge (z \vee w \vee x)$

Satisfiable CNF Formula

- Is $(x \vee \neg y)$ satisfiable?
  - yes: set $x = T$ and $y = F$ to get overall T
- Is $x \wedge \neg x$ satisfiable?
  - no: both $x = T$ and $x = F$ result in overall F
- Is $(x \vee y) \wedge (z \vee w \vee x)$ satisfiable?
  - yes: $x = T$, $y = T$, $z = F$, $w = T$ result in overall T
- If formula has n variables, then there are 2^n different truth assignments.

Definition of SAT

- SAT = all (and only) strings that encode satisfiable CNF formulas.

SAT is NP-Complete

- Cook's Theorem: SAT is NP-complete.
- Proof ideas:
- (1) SAT is in NP: Given a candidate solution (a truth assignment) for a CNF formula, verify in polynomial time (by plugging in the truth values and evaluating the expression) whether it satisfies the formula (makes it true).

SAT is NP-Complete

- How to show that every language in NP is polynomially reducible to SAT?
- Key idea: the common thread among all the languages in NP is that each one is solved by some nondeterministic Turing machine (a formal model of computation) in polynomial time.
- Given a description of a poly time TM, construct in poly time, a CNF formula that simulates the computation of the TM.

Proving NP-Completeness By

To show L is NP-complete:

(1) Show L is in NP.

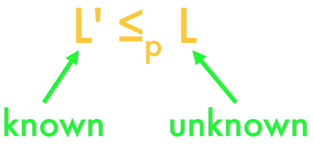
(2.a) Choose an appropriate known NP-complete language L'.

(2.b) Show $L' \leq_p L$: Describe an algorithm to compute a function f such that

- f is poly time
- f maps inputs for L' to inputs for L s.t. x is in L' if and only if f(x) is in L

Get the Direction Right!

- We want to show that L is at least as hard (time-consuming) as L'.
- So if we have an algorithm A for L, then we can solve L' with polynomial overhead
- Algorithm for L':
  - input: x
  - compute y = f(x)
  - run algorithm A for L on y
  - return whatever A returns



$L' \leq_p L$

3SAT

Definition of 3SAT

- 3SAT is a special case of SAT: each clause contains exactly 3 literals.
- Is 3SAT in NP?
  - Yes, because SAT is in NP.
- Is 3SAT NP-complete?
  - Not obvious. It has a more regular structure, which can perhaps be exploited to get an efficient algorithm
  - In fact, 2SAT does have a polynomial time algorithm

Showing 3SAT is NP-Complete

(1) To show 3SAT is in NP, use same algorithm as for SAT to verify a candidate solution (truth assignment)

(2.a) Choose SAT as known NP-complete problem.

(2.b) Describe a reduction from

SAT inputs to 3SAT inputs

- computable in poly time
- SAT input is satisfiable iff constructed 3SAT input is satisfiable

Reduction from SAT to 3SAT

- We're given an arbitrary CNF formula $C = c_1 \wedge c_2 \wedge \dots \wedge c_m$ over set of variables U
  - each c_i is a clause (disjunction of literals)
- We will replace each clause c_i with a set of clauses C_i', and may use some extra variables U_i' just for this clause
- Each clause in C_i' will have exactly 3 literals
- Transformed input will be conjunction of all the clauses in all the C_i'
- New clauses are carefully chosen…

Reduction from SAT to 3SAT

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 1: $k = 1$.
  - Use extra variables y_i^1 and y_i^2.
  - Replace c_i with 4 clauses:

    $(z_1 \vee y_i^1 \vee y_i^2)$

    $(z_1 \vee \neg y_i^1 \vee y_i^2)$

    $(z_1 \vee y_i^1 \vee \neg y_i^2)$

    $(z_1 \vee \neg y_i^1 \vee \neg y_i^2)$

Reduction from SAT to 3SAT

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 2: $k = 2$.
  - Use extra variable y_i^1.
  - Replace c_i with 2 clauses:

    $(z_1 \vee z_2 \vee \neg y_i^1)$

    $(z_1 \vee z_2 \vee y_i^1)$

Reduction from SAT to 3SAT

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 3: $k = 3$.
  - No extra variables are needed.
  - Keep c_i:

    $(z_1 \vee z_2 \vee z_3)$

Reduction from SAT to 3SAT

Let $c_i = z_1 \lor z_2 \lor \ldots \lor z_k$

- Case 4: k > 3.
  - Use extra variables $y_i^1, \ldots, y_i^{k-3}$.
  - Replace c_i with k-2 clauses:

$(z_1 \lor z_2 \lor y_i^1)$

$(\neg y_i^1 \lor z_3 \lor y_i^2)$

$(\neg y_i^2 \lor z_4 \lor y_i^3)$

. . .

. . .

$(\neg y_i^{k-5} \lor z_{k-3} \lor y_i^{k-4})$

$(\neg y_i^{k-4} \lor z_{k-2} \lor y_i^{k-3})$

$(\neg y_i^{k-3} \lor z_{k-1} \lor z_k)$

Correctness of Reduction

- Show that CNF formula C is satisfiable iff the 3-CNF formula C' constructed is satisfiable.
- =>: Suppose C is satisfiable. Come up with a satisfying truth assignment for C'.
- For variables in U, use same truth assignments as for C.
- How to assign T/F to the new variables?

Truth Assignment for New Variables

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 1: $k = 1$.
  - Use extra variables y_i^1 and y_i^2.
  - Replace c_i with 4 clauses:

$(z_1 \vee y_i^1 \vee y_i^2)$

$(z_1 \vee \neg y_i^1 \vee y_i^2)$

$(z_1 \vee y_i^1 \vee \neg y_i^2)$

$(z_1 \vee \neg y_i^1 \vee \neg y_i^2)$

Since z_1 is true, it does
not
matter how we assign
y_i^1 and y_i^2

Truth Assignment for New Variables

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 2: $k = 2$.
  - Use extra variable y_i^1.
  - Replace c_i with 2 clauses:

$(z_1 \vee z_2 \vee \neg y_i^1)$

$(z_1 \vee z_2 \vee y_i^1)$

Since either z_1 or z_2 is true, it does not matter how we assign y_i^1

Truth Assignment for New Variables

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 3: k = 3.
  - No extra variables are needed.
  - Keep c_i:

    $(z_1 \vee z_2 \vee z_3)$

No new variables.

Truth Assignment for New Variables

Let $c_i = z_1 \lor z_2 \lor \ldots \lor z_k$

- Case 4: $k > 3$.
  - Use extra variables $y_i^1, \ldots, y_i^{k-3}$.
  - Replace c_i with k-2 clauses:

If first true literal is z_1 or z_2, set all y_i's to false: then all later clauses have a true literal

$(z_1 \lor z_2 \lor y_i^1)$

$(\neg y_i^1 \lor z_3 \lor y_i^2)$

$(\neg y_i^2 \lor z_4 \lor y_i^3)$

. . .

. . .

$(\neg y_i^{k-5} \lor z_{k-3} \lor y_i^{k-4})$

$(\neg y_i^{k-4} \lor z_{k-2} \lor y_i^{k-3})$

$(y_i^{k-3} \lor z_{k-1} \lor z_k)$

Truth Assignment for New Variables

Let $c_i = z_1 \lor z_2 \lor \ldots \lor z_k$

- Case 4: $k > 3$.
  - Use extra variables $y_i^1, \ldots, y_i^{k-3}$.
  - Replace c_i with $k-2$ clauses:

$(z_1 \lor z_2 \lor y_i^1)$

$(\neg y_i^1 \lor z_3 \lor y_i^2)$

$(\neg y_i^2 \lor z_4 \lor y_i^3)$

. . .

. . .

$(\neg y_i^{k-5} \lor z_{k-3} \lor y_i^{k-4})$

$(\neg y_i^{k-4} \lor z_{k-2} \lor y_i^{k-3})$

$(\neg y_i^{k-3} \lor z_{k-1} \lor z_k)$

If first true literal is z_{k-1} or z_k, set all y_i's to true: then all earlier clauses have a true literal

Truth Assignment for New Variables

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 4: $k > 3$.
  - Use extra variables $y_i^1, \ldots, y_i^{k-3}$.
  - Replace c_i with k-2 clauses:

$(z_1 \vee z_2 \vee y_i^1)$

$(\neg y_i^1 \vee z_3 \vee y_i^2)$

$(\neg y_i^2 \vee z_4 \vee y_i^3)$

. . .

. . .

$(\neg y_i^{k-5} \vee z_{k-3} \vee y_i^{k-4})$

$(\neg y_i^{k-4} \vee z_{k-2} \vee y_i^{k-3})$

$(\neg y_i^{k-3} \vee z_{k-1} \vee z_k)$

If first true literal is in between, set all earlier y_i's to true and all later y_i's to false

Correctness of Reduction

- <=: Suppose the newly constructed 3SAT formula C' is satisfiable. We must show that the original SAT formula C is also satisfiable.
- Use the same satisfying truth assignment for C as for C' (ignoring new variables).
- Show each original clause has at least one true literal in it.

Original Clause Has a True Literal

Let $c_i = z_1 \lor z_2 \lor \ldots \lor z_k$

- Case 1: $k = 1$.
  - Use extra variables y_i^1 and y_i^2.
  - Replace c_i with 4 clauses:

$(z_1 \lor y_i^1 \lor y_i^2)$

$(z_1 \lor \neg y_i^1 \lor y_i^2)$

$(z_1 \lor y_i^1 \lor \neg y_i^2)$

$(z_1 \lor \neg y_i^1 \lor \neg y_i^2)$

For every assignment of y_i^1 and y_i^2, in order for all 4 clauses to have a true literal, z_1 must be true.

Original Clause Has a True Literal

Let $c_i = z_1 \lor z_2 \lor \ldots \lor z_k$

- Case 2: $k = 2$.
  - Use extra variable y_i^1.
  - Replace c_i with 2 clauses:

$(z_1 \lor z_2 \lor y_i^1)$

$(z_1 \lor z_2 \lor \neg y_i^1)$

For either assignment of y_i^1, in order for both clauses to have a true literal, z_1 or z_2 must be true.

Original Clause Has a True

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 3: $k = 3$.
  - No extra variables are needed.
  - Keep c_i:

    $(z_1 \vee z_2 \vee z_3)$

No new variables.

Original Clause Has a True Literal

Let $c_i = z_1 \vee z_2 \vee \ldots \vee z_k$

- Case 4: $k > 3$.
  - Use extra variables $y_i^1, \ldots, y_i^{k-3}$.
  - Replace c_i with $k-2$ clauses:

$(z_1 \vee z_2 \vee y_i^1)$

$(\neg y_i^1 \vee z_3 \vee y_i^2)$

$(\neg y_i^2 \vee z_4 \vee y_i^3)$

. . .

. . .

$(\neg y_i^{k-5} \vee z_{k-3} \vee y_i^{k-4})$

$(\neg y_i^{k-4} \vee z_{k-2} \vee y_i^{k-3})$

$(\neg y_i^{k-3} \vee z_{k-1} \vee z_k)$

Suppose in contradiction all z_i's are false. Then y_i^1 must be true, y_i^2 must be true,... Impossible!

Why is Reduction Poly Time?

- The running time of the reduction (the algorithm to compute the 3SAT formula C', given the SAT formula C) is proportional to the size of C'
- rules for constructing C' are simple to calculate

Size of New Formula

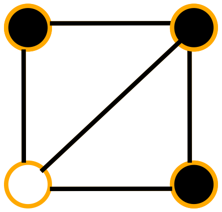
- original clause with 1 literal becomes 4 clauses with 3 literals each
- original clause with 2 literals becomes 2 clauses with 3 literals each
- original clause with 3 literals becomes 1 clause with 3 literals
- original clause with k > 3 literals becomes k-2 clauses with 3 literals each
- So new formula is only a constant factor larger than the original formula

Vertex Cover

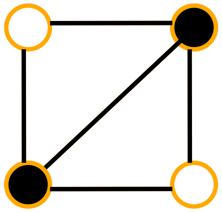
Vertex Cover of a Graph

- Given undirected graph G = (V,E)
- A subset V' of V is a vertex cover if every edge in E has at least one endpoint in V'
- Easy to find a big vertex cover: let V' be all the nodes
- What about finding a small vertex cover?

Vertex Cover Example



vertex cover
of size 3



vertex cover
of size 2

Vertex Cover Decision Problem

- VC: Given a graph G and an integer K, does G have a vertex cover of size at most K?
- Theorem: VC is NP-complete.
- Proof: First, show VC is in NP:
- Given a candidate solution (a subset V' of the nodes), check in polynomial time if |V'| ≤ K and if every edge has at least one endpoint in V'.

VC is NP-Complete

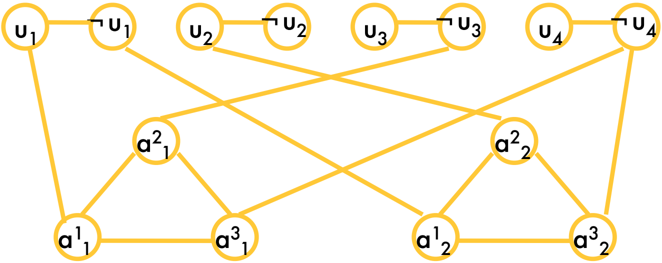
- Now show some known NP-complete problem is polynomially reducible to VC.
- So far, we have two options, SAT and 3SAT.
- Let's try 3SAT: since inputs to 3SAT have a more regular structure than inputs to SAT, maybe it will be easier to define a reduction from 3CNF formulas to graphs.

Reducing 3SAT to VC

- Let $C = c_1 \wedge \ldots \wedge c_m$ be any 3SAT input over set over variables $U = \{u_1, \ldots, u_n\}$.
- Construct a graph G like this:
  - two nodes for each variable, u_i and $\neg u_i$, with an edge between them ("literal" nodes)
  - three nodes for each clause c_j, "placeholders" for the three literals in the clause: a^1_j, a^2_j, a^3_j, with edges making a triangle
  - edges connecting each placeholder node in a triangle to the corresponding literal node
- Set K to be n + 2m.

Example of Reduction

- 3SAT input has variables u1, u2, u3, u4 and clauses $(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$.
- K = 4 + 2*2 = 8



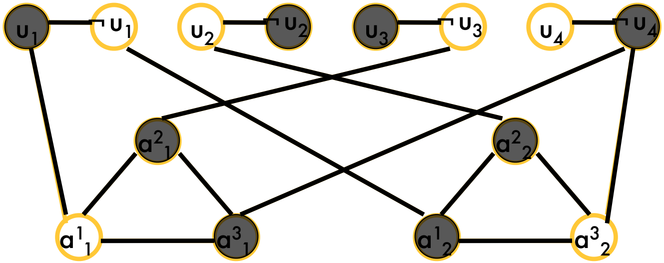
Correctness of Reduction

- Suppose the 3SAT input (with m clauses over n variables) has a satisfying truth assignment.
- Show there is a VC of G of size n + 2m:
  - pick the node in each pair corresponding to the true literal w.r.t. the satisfying truth assignment
  - pick two of the nodes in each triangle such that the excluded node is connected to a true literal

Example of Reduction

$(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$

u_1= T
u_2= F
u_3= T
u_4= F



Correctness of Reduction

- Since one from each pair is chosen, the edges in the pairs are covered.
- Since two from each triangle are chosen, the edges in the triangles are covered.
- For edges between triangles and pairs:
  - edge to a true literal is covered by pair choice
  - edges to false literals are covered by triangle choices

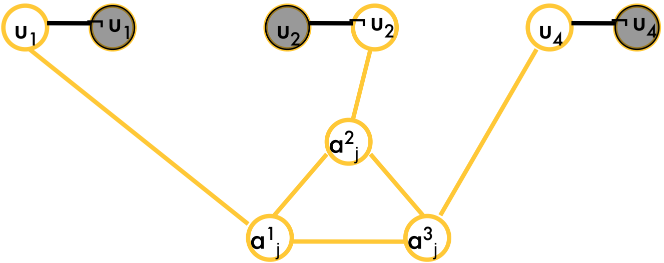
Correctness of Reduction

- Suppose G has a vertex cover V' of size at most K.
- To cover the edges in the pairs, V' must contain at least one node in each pair
- To cover the edges in the triangles, V' must contain at least two nodes in each triangle
- Since there are n pairs and m triangles, and since K = n + 2m, V' contains exactly one from each pair and two from each triangle.

Correctness of Reduction

- Use choice of nodes in pairs to define a truth assignment:
  - if node u_i is chosen, then set variable u_i to T
  - if node $\neg u_i$ is chosen, then set variable u_i to F
- Why is this a satisfying truth assignment?
- Seeking a contradiction, suppose that some clause has no true literal….

Correctness of Reduction



In order to cover the triangle-to-literal edges, all three nodes in this triangle must be chosen, contradicting fact that only two can be chosen (since size is n + 2m).

Running Time of the Reduction

- Show graph constructed is not too much bigger than the input 3SAT formula:
  - number of nodes is 2n + 3m
  - number of edges is n + 3m + 3m
- Size of VC input is polynomial in size of 3SAT input, and rules for constructing the VC input are quick to calculate, so running time is polynomial.

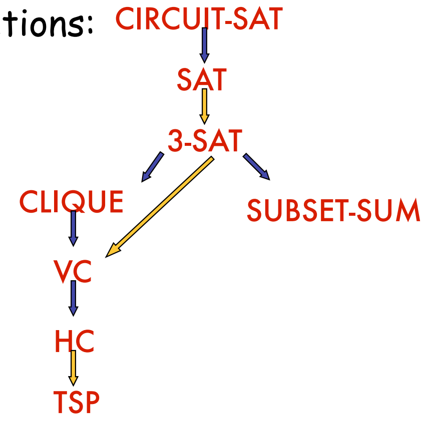
Further Examples

Some NP-Complete Problems

- SAT
- 3-SAT
- VC
- TSP
- CLIQUE (does G contain a completely connected subgraph of size at least K?)
- HC (does G have a Hamiltonian cycle?)
- SUBSET-SUM (given a set S of natural numbers and integer t, is there a subset of S that sum to t?)

Relationship Between Some NP-

- Textbook shows NP-completeness using this tree of reductions:

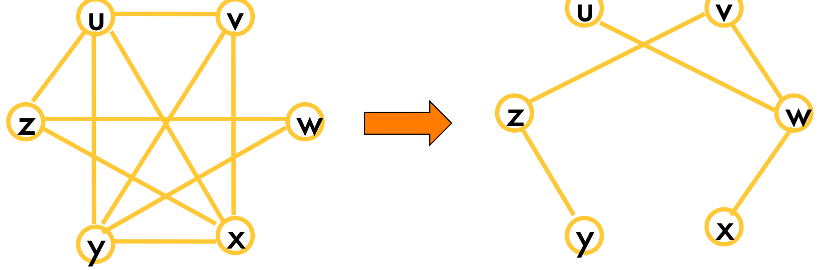


We've seen
the orange
reductions

Clique

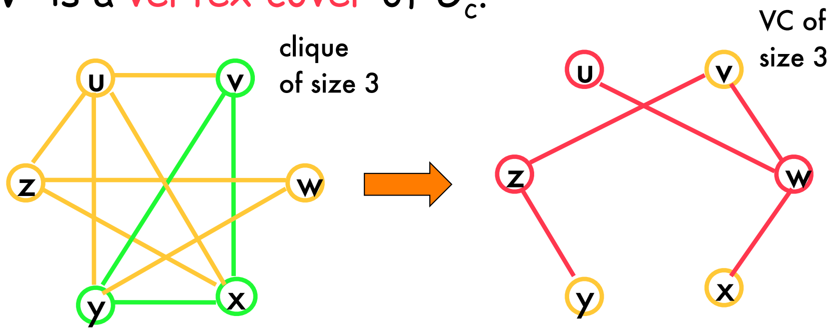
CLIQUE vs. VC

- The complement of graph G = (V,E) is the graph G_c = (V,E_c), where E_c consists of all the edges that are missing in G.



CLIQUE vs. VC

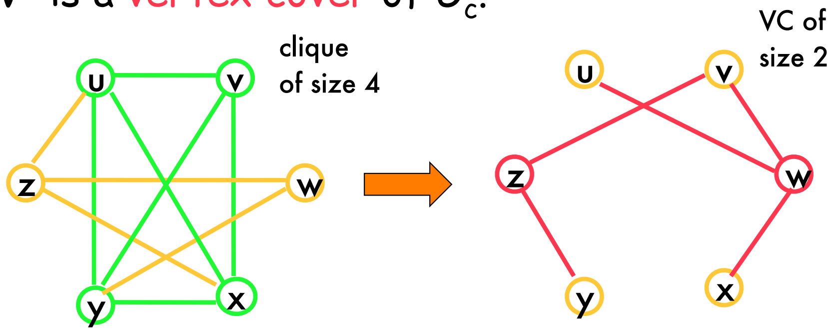
- Theorem: V' is a clique of G if and only if V - V' is a vertex cover of G_c.



the nodes in V' would only "cover" missing edges and thus are not needed in G_c

CLIQUE vs. VC

- Theorem: V' is a clique of G if and only if V - V' is a vertex cover of G_c.



clique of size 4

VC of size 2

the nodes in V' would only "cover" missing edges and thus are not needed in G_c

VC and CLIQUE

- Can use previous observation to show that VC $\leq_p$ CLIQUE and also to show that CLIQUE $\leq_p$ VC.

Useful Reference

- Additional source: Computers and Intractability, A Guide to the Theory of Intractability, M. Garey and D. Johnson, W. H. Freeman and Co., 1979

Dealing with NP-Complete Problems

Dealing with NP-Completeness

- Suppose the problem you need to solve is NP-complete. What do you do next?
- hope/show bad running time does not happen for inputs of interest
- find heuristics to improve running time in many cases (but no guarantees)
- find a polynomial time algorithm that is guaranteed to give an answer close to optimal

Optimization Problems

- Concentrate on approximation algorithms for optimization problems:
  - every candidate solution has a positive cost
- Minimization problem: goal is to find smallest cost solution
  - Ex: Vertex cover problem, cost is size of VC
- Maximization problem: goal is to find largest cost solution
  - Ex: Clique problem, cost is size of clique

Approximation Algorithms

An approximation algorithm for an optimization problem

- runs in polynomial time and
- always returns a candidate solution

Ratio Bound

Ratio bound: Bound the ratio of the cost of the solution returned by the approximation algorithm and the cost of an optimal solution

- minimization problem:

  cost of approx solution / cost of optimal solution

- maximization problem:

  cost of optimal solution / cost of approx solution

So ratio is always at least 1, goal is to get it as close to 1 as we can

Approximation Algorithms

A poly-time algorithm A is called a δ-approximation algorithm for a minimization problem P if and only if for every problem instance I of P with an optimal solution value OPT(I), it delivers a solution of value A(I) satisfying $A(I) \leq \delta OPT(I)$.

Approximation Algorithms

A poly-time algorithm A is called a δ-approximation algorithm for a maximization problem P if and only if for every problem instance I of P with an optimal solution value OPT(I), it delivers a solution of value A(I) satisfying $A(I) \geq \delta OPT(I)$.

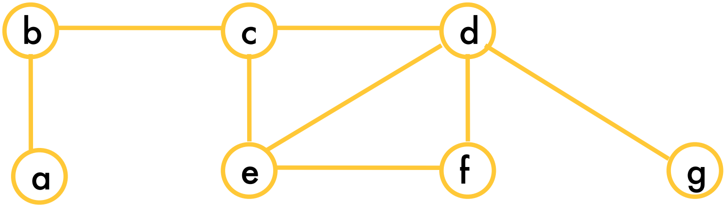
Approximation Algorithm for Minimum Vertex Cover Problem

```
input:  G = (V,E)
C := ∅
E' := E
while E' ≠ ∅ do
   pick any (u,v) in E'
   C := C U {u,v}
   remove from E' every edge incident on u or v
endwhile
return C
```

Min VC Approx Algorithm

- Time is O(E), which is polynomial.
- How good an approximation does it provide?
- Let's look at an example.

Min VC Approx Alg Example



choose (b,c): remove (b,c), (b,a), (c,e), (c,d)
choose (e,f): remove (e,f), (e,d), (d,f)
Answer: {b,c,e,f,d,g}
Optimal answer: {b,d,e}
Algorithm's ratio bound is 6/3 = 2.

Ratio Bound of Min VC Alg

Theorem: Min VC approximation algorithm has ratio bound of 2.

Proof: Let A be the total set of edges chosen to be removed.

- Size of VC returned is 2*|A| since no two edges in A share an endpoint.
- Size of A is at most size of a min VC since min VC must contain at least one node for each edge in A.
- Thus cost of approx solution is at most twice cost of optimal solution

More on Min VC Approx Alg

- Why not run the approx alg and then divide by 2 to get the optimal cost?
- Because answer is not always exactly twice the optimal, just never more than twice the optimal.
- For instance, a different choice of edges to remove gives a different answer:
  - Choosing (d,e) and then (b,c) produces answer {b,c,d,e} with cost 4 as opposed to optimal cost 3

Triangle Inequality

- Assume TSP inputs with the triangle inequality:
  - distances satisfy property that for all cities a, b, and c, dist(a,c) ≤ dist(a,b) + dist(b,c)
  - i.e., shortest path between 2 cities is direct route
- Depending on what you are modeling

TSP Approximation Algorithm

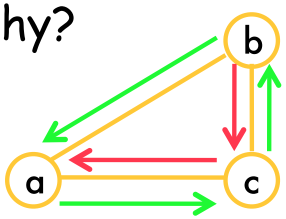
- input: set of cities and distances b/w them that satisfy the triangle inequality
- create complete graph G = (V,E), where V is set of cities and weight on edge (a,b) is dist (a,b)
- compute MST of G
- Go twice around the MST to get a tour (that will have duplicates)
- Remove duplicates to avoid visiting a city more than once

Analysis of TSP Approx Alg

- Running time is polynomial (creating complete graph takes $O(V^2)$ time, Kruskal's MST algorithm takes time $O(E \log E) = O(V^2 \log V)$.
- How good is the quality of the solution?

Analysis of TSP Approx Alg

- cost of approx solution ≤ 2*weight of MST, by triangle inequality
- Why?



when tour created by going around the MST is adjusted to remove duplicate nodes, the two red edges are replaced with the green diagonal edge

Analysis of TSP Approx Alg

- weight of MST < length of min tour
- Why?
- Min tour minus one edge is a spanning tree T, whose weight must be at least the weight of MST.
- And weight of min tour is greater than weight of T.

Analysis of TSP Approx Alg

- Putting the pieces together:
- cost of approx solution ≤ 2*weight of MST

  ≤ 2*cost of min tour
- So approx ratio is at most 2.

Suppose we don't have triangle inequality.

TSP Without Triangle Inequality

Theorem: If P ≠ NP, then no polynomial time approximation algorithm for TSP (w/o triangle inequality) can have a constant ratio bound.

Proof: We will show that if there is such an approximation algorithm, then we could solve a known NP-complete problem (Hamiltonian cycle) in polynomial time, so P would equal NP.

HC Exact Algorithm using TSP

input: G = (V,E)

1. convert G to this TSP input:
   - one city for each node in V
   - distance between cities u and v is 1 if (u,v) is in E
   - distance between cities u and v is r*|V| if (u,v) is not in E, where r is the ratio bound of the TSP approx alg
   - Note: This TSP input does not satisfy the triangle inequality

HC (Exact) Algorithm Using

2. run TSP approx alg on the input just created
3. if cost of approx solution returned in step 2 is ≤ r*|V| then return YES else return NO

Running time is polynomial.

Correctness of HC Algorithm

- If G has a HC, then optimal tour in TSP input constructed corresponds to that cycle and has weight |V|.
- Approx algorithm returns answer with cost at most r*|V|.
- So if G has HC, then algorithm returns YES.

Correctness of HC Algorithm

- If G has no HC, then optimal tour for TSP input constructed must use at least one edge not in G, which has weight r*|V|.
- So weight of optimal tour is > r*|V|, and answer returned by approx alg has weight > r*|V|.
- So if G has not HC, then algorithm returns NO.

Set Cover

Given a universe U of n elements, a collection $S = \{S_1, S_2, \ldots, S_k\}$ of subsets of U, and a cost function $c: S \to Q^+$, find a minimum cost subcollection of S that covers all the elements of U.

Example

We might want to select a committee consisting of people who have combined all skills.

Cost-Effectiveness

We are going to pick a set according to its cost effectiveness.

Let C be the set of elements that are already covered.

The cost effectiveness of a set S is the average cost at which it covers new elements: $c(S)/|S-C|$.

Greedy Set Cover

```
$C := \emptyset$
while $C \neq U$ do
    Find most cost effective set in current iteration, say S, and pick it.
    For each e in S, set price(e)= $c(S)/|S-C|$.
    $C := C \cup S$
Output C
```

Theorem

Greedy Set Cover is an H_m-approximation algorithm, where $m = \max\{ |S_i| : 1<=i<=k\}$.

Lemma

For all sets T in S, we have

$\sum_{e \text{ in } T}$ price(e) <= c(T) H_x with x=|T|

Proof: Let e in $T \cap (S_i \setminus \bigcup_{j<i} S_j)$ and

$V_i = T \setminus \bigcup_{j<i} S_j$ be the remaining part of T before being covered by the greedy cover.

Lemma (2)

Then the greedy property implies that

price(e) <= $c(T)/|V_i|$

Let $e_1,\dots,e_m$ be the elements of T in the order chosen by the greedy algorithm.

It follows that

price(e_k) <= $w(T)/(|T|-k+1)$.

Summing over all k yields the claim.

Proof of the Theorem

- Let A be the optimal set cover and B the set cover returned by the greedy algorithm.
- $\sum$ price(e) <= $\sum_{S \text{ in } A} \sum_{e \text{ in } S}$ price(e)
  By the lemma, this is bounded by
- $\sum_{T \text{ in } A} c(T)H_{|T|}$
- The latter sum is bounded by $\sum_{T \text{ in } A} c(T)$ times the Harmonic number of the cardinality of the largest set in S.

Example

- Let $U = \{e_1,\dots,e_n\}$
- $S = \{ \{e_1\},\dots,\{e_n\}, \{e_1,\dots,e_n\} \}$
- $c(\{e_i\}) = 1/i$
- $c(\{e_1,\dots,e_n\})=1+\varepsilon$
- The greedy algorithm computes a cover of cost H_n and the optimal cover is $1+\varepsilon$