NP-Completeness

Goal: We want some way to classify problems that are hard to solve, i.e. problems for which we can not find polynomial time algorithms.

For many interesting problems

- we cannot find a polynomial time algorithm
- we cannot prove that no polynomial time algorithm exists
- the best we can do is formalize a class of NP-complete problems that either all have polynomial time algorithms or none have polynomial time algorithms

NP-completeness arises in many fields including

- biology
- chemistry
- economics
- physics
- engineering
- sports
- etc.

Goal in class:

To learn how to prove that problems are NP-complete.

We need a formalism for proving problems hard.

Turing Machine (simplified description)

A Turing Machine has

- **Finite state control**
- **Infinite tape (each square can hold 0, 1, \$, or be blank.**
- **Read-Write head**

Each step of the finite state control is a function

$$f(\textbf{current state}, \textbf{tape symbol}) \rightarrow (\textbf{new state}, \textbf{symbol to write}, \textbf{movement of head})$$

Example

Program to test if a binary number is even. Input is \$ terminated. Output is written immediately after \$, 1 for yes, 0 for no.

- **Read until \$ (state** q_0**)**
- **Back up, check last digit (state** q_1**)**
- **if even, write a 1 (states** q_2, q_3, q_F**)**
- **if odd, write a 0 (states** q_4, q_5, q_F**)**

Here is a program. Each cell is (new state, write symbol move)

state	input 0	input 1	input \$
(q_0)	$(q_0, -, R)$	$(q_0, -, R)$	$(q_1, -, L)$
(q_1)	$(q_2, -, R)$	$(q_4, -, R)$	error
(q_2)	error	error	$(q_3, -, R)$
(q_3)	$(q_F, 1, -)$	$(q_F, 1, -)$	$(q_F, 1, -)$
(q_4)	error	error	$(q_5, -, R)$
(q_5)	$(q_F, 0, -)$	$(q_F, 0, -)$	$(q_F, 0, -)$
(q_F)	halt	halt	halt

Church Turing Thesis **The set of things that can be computed on a TM is the same as the set of things that can be computed on any digital computer.**

P

Definition Let P be defined as the set of problems that can be solved in polynomial time on a TM (On an input of size n, they can be solved in time $O(n^k)$ for some constant k)

Theorem P is the set of problems that can be solved in polynomial time on the model of computation used in CSOR 4231 and on every modern non-quantum digital computer.

Technicalities

- We assume a reasonable (binary) encoding of input
- Note that all computers are related by a polynomial time transformation. Think of this as a "compiler"

Further details

- We restrict attention to "yes-no" questions
- Shortest path is now "Given a graph G and a number b does the shortest path from s to t have length at most b.
- We do not use the language framework from the book in class

Verification

Verification Given a problem X and a possible solution S, is S a solution to X.

Example X is shortest paths and S is an s-t path in S that is claimed to have length at most b, check whether the path really is of length at most b

Example X is sorting and S is an allegedly sorted list. Is the list really sorted?

Claim Verification is no harder than solving a problem from scratch.
We write

$$\text{Verification} \le \text{Solving}$$

Def: NP is the set of problems that can be verified in polynomial time

Formally: Problem X with input of size n is in NP if there exists a "certificate" y, $|y| = \text{poly}(n)$ such that, using y, one can verify whether a solution x is really a solution in polynomial time. (Think of y as the "answer")

Some problems

Longest Path **Given a graph G, and number k is the longest simple path from s to t of length $\geq k$.**

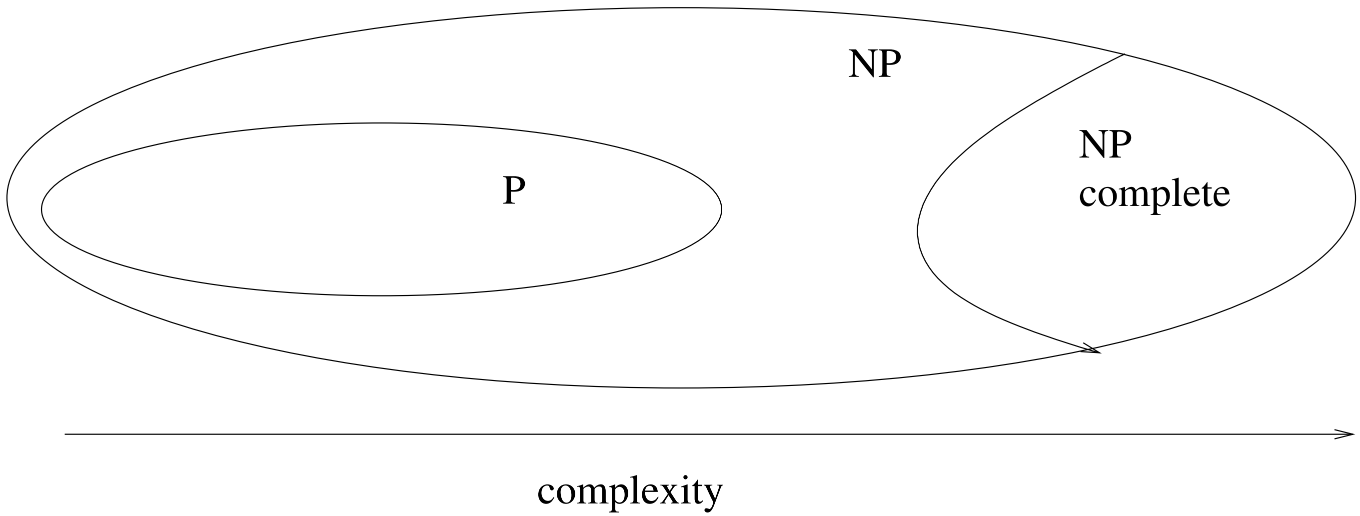
Satisfiability **Given a formula Φ in CNF (conjunctive normal form), does there exist a satisfying assignment to Φ, i.e. an assignment of the variables that evaluates to true.**

Big Question

$$P = NP??$$

Is solving a problem no harder than verifying?

Don't know answer. Instead we will identify "hardest" problems in NP. If any of these are in P then all of NP is in P.



NP-complete

Definition **Problem X is NP-complete if**

1. $X \in NP$

2. $Y \leq X \ \ \forall Y \in NP$

Definition $Y \leq X$ **means**

- **Y is polynomial time reducible to X, which means**

there exists a polynomial time computable function f **that maps inputs to** Y **to inputs to** X**, such that**

input y **to problem** Y **returns "Yes" iff input** $f(y)$ **to problem** X **returns "Yes"**

Informally $Y \leq X$ **means that Y is "not much harder than" ("easier than") X**

Theorem

If $Y \leq X$ **then** $X \in P \Rightarrow Y \in P$

Contrapositive

If $Y \leq X$ **then** $Y \notin P \Rightarrow X \notin P$

SAT

Theorem SAT is NP-complete

Proof idea: The turing machine program for any problem in NP can be verified by a polynomial sized SAT instance that encodes that the input is well formed and that each step follows legally from the next.

Implication We now have one NP-complete problem. We will now reduce other problems to it.

Reductions

- **If I want to show that X is easy, I show that in polynomial time I can reduce X to Y, where I already know that Y is easy.**
- **If I want to show that X is hard, then I reduce Y to X, where I already know that Y is hard.**
- **So if SAT** $\leq X$**, then** X **is hard.**

Showing X is NP-complete

To show that X is NP-complete, I show:

1. $X \in NP$
2. **For some problem Z that I know to be NP-complete** $Z \leq X$

Showing X is NP-complete

To show that X is NP-complete, I show:

1. $X \in NP$
2. **For some problem Z that I know to be NP-complete** $Z \leq X$

Expanded version: **To show that X is NP-complete, I show:**

1. $X \in NP$
2. **Find a known NP-complete problem Z.**
3. **Describe f, which maps input z to Z to input $f(z)$ to X.**
4. **Show that Z with input z returns "yes" iff X with input $f(z)$ returns "yes'**
5. **Show that f runs in polynomial time.**

3SAT

3SAT is SAT with exactly 3 literals per clause

Example:

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_4 \vee \overline{x_5}) \wedge (x_1 \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_5})$$

Comments

- n **variables,** m **clauses**
- **3SAT is a special case of SAT**
- **If SAT is easy, then 3SAT must be easy**
- **IS SAT is hard, then ???**
- **1-SAT is easy.**
- **2-SAT is easy.**

3SAT is NP-complete

Expanded version: **To show that X is NP-complete, I show:**

1. $X \in NP$
2. **Find a known NP-complete problem Z.**
3. **Describe f, which maps input z to Z to input $f(z)$ to X.**
4. **Show that Z with input z returns "yes" iff X with input $f(z)$ returns "yes'**
5. **Show that f runs in polynomial time.**

1) 3SAT is in NP. becasue SAT is in NP and 3SAT is a special case of SAT.

2) SAT
3,4, 5) Next slde..

Reduction

Approach We need to show how to convert an input to SAT into an input to 3SAT, while preserving yes/no instances. We will give a clause by clause conversion. Let k be the number of literals in a clause

Easy cases:

- $k = 1$. $x_1 \Rightarrow (x_1 \vee x_1 \vee x_1)$
- $k = 2$. $(x_1 \vee x_2) \Rightarrow (x_1 \vee x_1 \vee x_2)$
- $k = 3$. $(x_1 \vee x_2 \vee x_3) \Rightarrow (x_1 \vee x_2 \vee x_3)$

Easy to verify that transformation preserves satisfiability

k=4

- **Need to convert** $x_1 \vee x_2 \vee x_3 \vee x_4$ **to a 3SAT expression.**
- **Will need more than one clause**

First try:

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4)$$

Is this true for exactly the same settings as $x_1 \vee x_2 \vee x_3 \vee x_4$ **?**

k=4

- **Need to convert** $x_1 \vee x_2 \vee x_3 \vee x_4$ **to a 3SAT expression.**
- **Will need more than one clause**

First try:

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4)$$

Is this true for exactly the same settings as $x_1 \vee x_2 \vee x_3 \vee x_4$ **?**

No: **Consider**

$$\begin{aligned} x_1 &= T \\ x_2 &= F \\ x_3 &= F \\ x_4 &= F \end{aligned}$$

Lesson: **Need additional variables**

k=4

- **Need to convert** $\Phi = x_1 \vee x_2 \vee x_3 \vee x_4$ **to a 3SAT expression.**
- **Will need more than one clause**
- **Will need extra variables**

3SAT Expression:

$$\Phi' = (x_1 \vee x_2 \vee y_1) \wedge (\overline{y_1} \vee x_3 \vee x_4)$$

Claim: **There is a setting of** x_1, x_2, x_3, x_4 **that makes** Φ **true iff there is a setting of** x_1, x_2, x_3, x_4, y_1 **that makes** Φ' **true.**

k=5

- **Need to convert** $\Phi = x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$ **to a 3SAT expression.**
- **Will need more than one clause**
- **Will need extra variables**

3SAT Expression:

$$\Phi' = (x_1 \vee x_2 \vee y_1) \wedge (\overline{y_1} \vee x_3 \vee y_2) \wedge (\overline{y_2} \vee x_4 \vee x_5)$$

Claim: **There is a setting of** x_1, x_2, x_3, x_4, x_5 **that makes** Φ **true iff there is a setting of** $x_1, x_2, x_3, x_4, x_5, y_1, y_2$ **that makes** Φ' **true.**

General k

- Need to convert $\Phi = x_1 \vee x_2 \vee \ldots \vee x_k$ to a 3SAT expression.
- Will need more than one clause
- Will need extra variables

3SAT Expression:

$$\begin{array}{rrl} \Phi' = & & (x_1 \vee x_2 \vee y_1) \\ & \wedge & (\overline{y_1} \vee x_3 \vee y_2) \\ & \wedge & \ldots \\ & \wedge & (\overline{y_{i-2}} \vee x_i \vee y_{i-1}) \\ & \wedge & \ldots \\ & \wedge & (\overline{y_{k-4}} \vee x_{k-2} \vee y_{k-3}) \\ & \wedge & (\overline{y_{k-3}} \vee x_{k-1} \vee x_k)) \end{array}$$

Claim: **There is a setting of $x_1, x_2, \ldots, x_k$ that makes Φ true iff there is a setting of $x_1, x_2, \ldots, x_k, y_1, \ldots, y_{k-3}$ that makes Φ' true.**

Recap

- Described f
- f is polynomial time
  - A clause with k variables is mapped to $k-2$ clauses of 3 variables each.
  - Clauses blow up by a factor of at most n
  - Variables blow up by a factor of at most n
- We argued (clause by clause) that Φ is a yes instance to SAT iff Φ' is a yes instance to 3SAT.

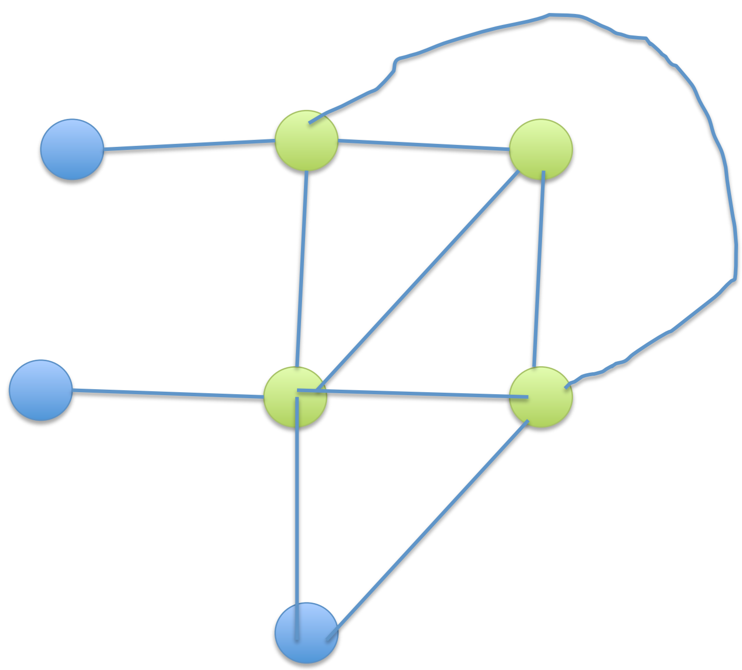
Sanity Checks

- **Why can't we prove that 2SAT is NP-complete via this reduction?**
- **What does the reduction from 2SAT to 3SAT tell us?**

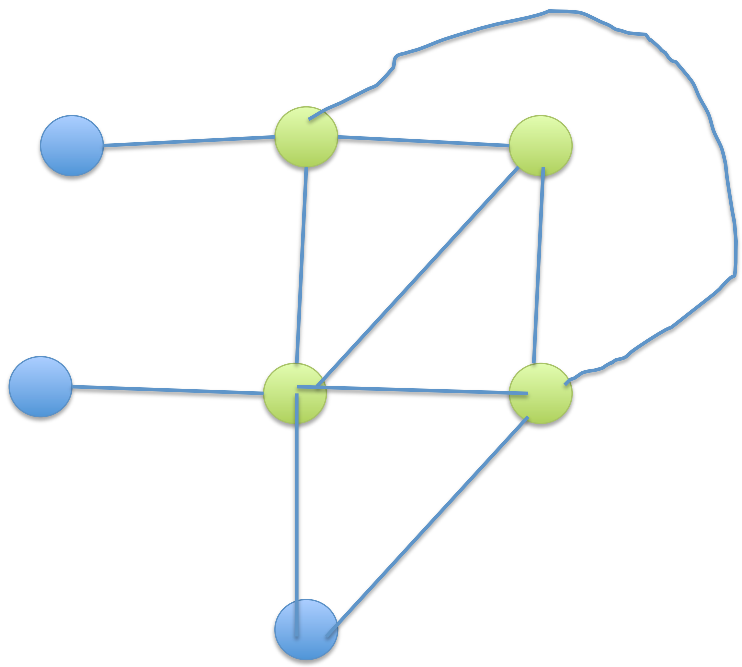
Clique

Definition A k -clique is a set of k vertices with all $\binom{k}{2}$ edges between them.

Clique Given a graph $G = (V, E)$ and an integer k , does G have a k -clique?



Clique



- G has a 4-clique
- G has no 5-clique.

Reduction

Goal We need to describe a function f that takes an instance Φ of 3SAT and produces instances $f(\Phi) = (G, k)$ of k-clique such that Φ is satisfiable iff $f(\Phi)$ has a k-clique.

Observation To make a 3SAT instance true, we need to make at least one literal in each clause true **Strategy:**

- A node for each appearance of a literal (a literal is a variable or its negation)
- An edge between literals that can be simultaneously true and in different clauses
- A k-clique will be a set of literals, one per clause, that can all be true simultaneously.

Example

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

Proof

Claim Φ is satisfiable iff G has a k -clique.

Proof

($\Rightarrow$) If Φ is satisfiable, then there is a setting of the variables with at least one literal per clause set to true. Let Z be such a set of literals. This set Z cannot contain both x_1 and $\overline{x_i}$, so in the graph G , the nodes in Z have an edge between each pair and therefore form a k -clique.

(($\Leftarrow$) If G has a k -clique, the clque must consist of k nodes, and they must be 1 per clause and must not have any pairs x_i and $\overline{x_i}$. Therefore you can set the corresponding literals to true and satisfy Φ

Reflections

- **We have actually shown that a "special case" with nodes in groups of 3 of clique is NP-complete. But if a special case is hard, there can't be a general algorithm for clique.**
- **In the proof, the function** f **goes one way, from 3SAT to clique, but the proof about yes instances has to go both ways.**
- **If the proof only went one way, it would be very easy (and incorrect)**

Vertex Cover

Defintion Given a graph $G = (V, E)$ and an integer k, a **vertex cover** $V' \subseteq V$ is a subset of the vertices such that for all edges (v, w), at least one of v and w is in V'. The vertex cover problem asks whether a graph G has a vertex cover of size at most k.

Claim Vertex cover is NP-complete.

- Vertex cover is in NP
- We will reduce from clique.
- What is the relationship between vertex covers and cliques, i.e. what does the vertex cover of a clique look like.

Reduction

Definition **Given a graph $G = (V, E)$ the complement G' is the graph in which edges are replaces by non-edges and vica versa.**

Claim: G **has a** k **-clique iff** G' **has a vertex cover of size** $|V| - k$ **.**

Subset Sum

Definition **Given a set of integers** $S = \{s_1, s_2, \ldots, s_n\}$ **and a target value** t **, is there a subset** $S' \subseteq S$ **such that** $\sum_{s_i \in S'} = t$ **.**

Example

$$S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\} \quad t = 3754$$

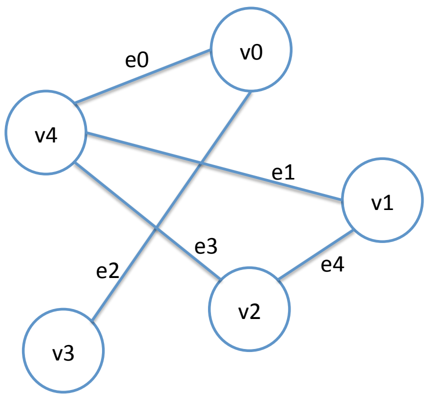
Solution

$$S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$$

Question **What about** $t = 3755$ **?**

Reduction

Claim **Vertex cover reduces to Subset Sum.**
Idea 1: **Look at the vertex edge adjacency matrix**



	e_4	e_3	e_2	e_1	e_0
v_0	**0**	**0**	**1**	**0**	**1**
v_1	**1**	**0**	**0**	**1**	**0**
v_2	**1**	**1**	**0**	**0**	**0**
v_3	**0**	**0**	**1**	**0**	**0**
v_4	**0**	**1**	**0**	**1**	**1**

We now have numbers!

Ideas

- A vertex cover is a subset R of rows, such that each column has at least one 1 in a row of R .
- Maybe we can think of the rows as binary numbers, can we say something about the sum of the numbers in R .
- Example, $R = \{v_1, v_3, v_4\}$

	e_4	e_3	e_2	e_1	e_0
v_0	**0**	**0**	**1**	**0**	**1**
v_1	**1**	**0**	**0**	**1**	**0**
v_2	**1**	**1**	**0**	**0**	**0**
v_3	**0**	**0**	**1**	**0**	**0**
v_4	**0**	**1**	**0**	**1**	**1**
$v_1 + v_3 + v_4$	**1**	**1**	**1**	**2**	**1**

- Sort of works:
  - If every edge had exactly one endpoint in R , then the binary sum would be 11111 , and we would choose $t = 11111$.
- Problems:
  - Edges can have one or two endpoints in R , which generates carries in base 2.
  - What should t be?
  - We ignored k .

Fixes

Problems:

- Edges can have one or two endpoints in R , which generates carries in base 2. Use base 4, and there won't be any carries
- What should t be?
- We ignored k . Add an extra column to "count". It will be the left-most column, so carries don't matter

	vert	e_4	e_3	e_2	e_1	e_0
x_0	1	0	0	1	0	1
x_1	1	1	0	0	1	0
x_2	1	1	1	0	0	0
x_3	1	0	0	1	0	0
x_4	1	0	1	0	1	1
$xv_1 + x_3 + x_4$	(3)	1	1	1	2	1

- Still have a problem, what should t be?
- We will introduce dummy rows to allow us to say that columns should sum to exactly 2 .

Final reduction

	vert	e_4	e_3	e_2	e_1	e_0	number converted to base 10
x_0	1	0	0	1	0	1	1041
x_1	1	1	0	0	1	0	1284
x_2	1	1	1	0	0	0	1344
x_3	1	0	0	1	0	0	1044
x_4	1	0	1	0	1	1	1093
y_0	0	0	0	0	0	1	1
y_1	0	0	0	0	1	0	4
y_2	0	0	0	1	0	0	16
y_3	0	0	1	0	0	0	64
y_4	0	1	0	0	0	0	256
t	(3)	2	2	2	2	2	3754

Claim G has a verex cover of size k iff the subset sum instance has a set that sums to t.

Hamiltonian Cycle

Definition Given a graph $G = (V, E)$, is there cycle visiting each vertex exactly once?

Fact Hamiltonian Cycle is NP-complete. See book for reduction.

Travelling Salesman Problem Given a graph $G = (V, E)$ with edge weights w and an integer B . Is there a Hamilonian Cycle C s.t.

$$\sum_{e \in C} w(e) \leq B$$

.

Claim Travelling Salesman Problem is NP-complete, via a reduction from Hamiltonian Cycle.

More NP-complete problems

Minimum Makespan Scheduling Given n jobs with processing times $p_1, \ldots, p_n$, and m identical machines and a number B. a schedule assigns each job to a machine. If J_i is the set of jobs assigned to machine i, then the load on machine i, $L_i = \sum_{j \in J_i} p_j$. The **makespan** of the schedule is the maximum machine load $M = \max_i L_i$. You want to know if there exists a schedule with makespan at most B.

3-partition Given a set of $3n$ numbers $x_1, \ldots, x_{3n}$, with $\sum_{i=1}^{3n} x_i = nB$, can you partition the numbers into n groups, each with 3 elements and each summing to B.