NP Completeness

Reporting to the Boss

Suppose you fail to find a fast algorithm. What can you tell your boss?

Reporting to the Boss

Suppose you fail to find a fast algorithm. What can you tell your boss?

- "I guess I'm too dumb…" (dangerous confession)

Reporting to the Boss

Suppose you fail to find a fast algorithm. What can you tell your boss?

- "I guess I'm too dumb..." (dangerous confession)
- "There is no fast algorithm!" (lower bound proof)

Reporting to the Boss

Suppose you fail to find a fast algorithm. What can you tell your boss?

- "I guess I'm too dumb. . . " (dangerous confession)
- "There is no fast algorithm!" (lower bound proof)
- "I can't solve it, but no one else in the world can, either. . . " (NP-completeness reduction)

Polynomial vs. Exponential Time

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f(n) = n!$
10	0.01 μs	0.1 μs	1 μs	3.63 ms
20	0.02 μs	0.4 μs	1 ms	77.1 years
30	0.03 μs	0.9 μs	1 sec	8.4×10^{15} years
40	0.04 μs	1.6 μs	18.3 min	
50	0.05 μs	2.5 μs	13 days	
100	0.1 μs	10 μs	4×10^{13} years	
1,000	1.00 μs	1 ms		

The Theory of NP-Completeness

- We've seen a few problems for which we couldn't find efficient algorithms, such as TSP.

The Theory of NP-Completeness

- We've seen a few problems for which we couldn't find efficient algorithms, such as TSP.
- We also couldn't prove exponential-time lower bounds for these problems.

The Theory of NP-Completeness

- We've seen a few problems for which we couldn't find efficient algorithms, such as TSP.
- We also couldn't prove exponential-time lower bounds for these problems.
- By the early 1970s, literally hundreds of problems were stuck in this limbo.

The Theory of NP-Completeness

- We've seen a few problems for which we couldn't find efficient algorithms, such as TSP.
- We also couldn't prove exponential-time lower bounds for these problems.
- By the early 1970s, literally hundreds of problems were stuck in this limbo.
- The theory of NP-Completeness, developed by Stephen Cook and Richard Karp, provided the tools to show that all of these problems were really the same problem.

Turing Awards

The Main Idea

Consider the following algorithm to solve Problem A using an algorithm for Problem B:

Alg-For-A(X)

  Convert X to an instance of Problem B, Y.
  Call Alg-For-B on Y to solve this instance.
  Return the answer of Alg-For-B(Y) as the answer.

Such a translation from instances of one type of problem to instances of another type such that answers are preserved is called a *reduction*.

What Does this Imply?

Now suppose my reduction translates X to Y in $O(P(n))$:

1. If Alg-For-B ran in $O(Q(n))$ I can solve Problem A in $O(P(n) + Q(n))$

What Does this Imply?

Now suppose my reduction translates X to Y in $O(P(n))$:

1. If Alg-For-B ran in $O(Q(n))$ I can solve Problem A in $O(P(n) + Q(n))$
2. If I know that $\Omega(L(n))$ is a lower-bound to solve Problem A, then $\Omega(L(n) - P(n))$ must be a lower-bound to solve Problem B.

What Does this Imply?

Now suppose my reduction translates X to Y in $O(P(n))$:

1. If Alg-For-B ran in $O(Q(n))$ I can solve Problem A in $O(P(n) + Q(n))$
2. If I know that $\Omega(L(n))$ is a lower-bound to solve Problem A, then $\Omega(L(n) - P(n))$ must be a lower-bound to solve Problem B.

What Does this Imply?

Now suppose my reduction translates X to Y in $O(P(n))$:

1. If Alg-For-B ran in $O(Q(n))$ I can solve Problem A in $O(P(n) + Q(n))$
2. If I know that $\Omega(L(n))$ is a lower-bound to solve Problem A, then $\Omega(L(n) - P(n))$ must be a lower-bound to solve Problem B.

The second argument is the idea we use to prove problems hard!

What is a Problem?

A problem is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution.

What is a Problem?

A problem is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution.

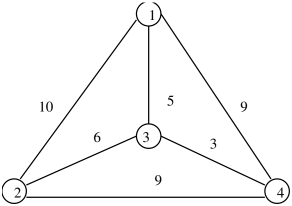
Example: TSP

Problem: Given a weighted graph G, what tour $\{v_1, v_2, \ldots, v_n\}$ minimizes $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$.

What is an Instance?

An instance is a problem with the input specified.

TSP instance: $d[v_1, v_2] = 10$, $d[v_1, v_3] = 5$, $d[v_1, v_4] = 9$, $d[v_2, v_3] = 6$, $d[v_2, v_4] = 9$, $d[v_3, v_4] = 3$



Solution: $\{v_1, v_3, v_4, v_2\}$ cost$= 27$

Input Encodings

- Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc.

Input Encodings

- Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc.
- All reasonable encodings will be within polynomial size of each other.

Input Encodings

- Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc.
- All reasonable encodings will be within polynomial size of each other.
- The fact that we can ignore minor differences in encoding is important.

Input Encodings

- Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc.
- All reasonable encodings will be within polynomial size of each other.
- The fact that we can ignore minor differences in encoding is important.
- We are concerned with the difference between algorithms which are polynomial and exponential in the size of the input.

Decision Problems

- A problem with answers restricted to *yes* and *no* is called a decision problem.

Decision Problems

- A problem with answers restricted to *yes* and *no* is called a decision problem.
- Most interesting optimization problems can be phrased as decision problems which capture the essence of the computation.

Decision Problems

- A problem with answers restricted to *yes* and *no* is called a decision problem.
- Most interesting optimization problems can be phrased as decision problems which capture the essence of the computation.
- For convenience, from now on we will talk *only* about decision problems.

The TSP Decision Problem

- Given a weighted graph G and integer k, does there exist a traveling salesperson tour with cost $\leq k$?

The TSP Decision Problem

- Given a weighted graph G and integer k, does there exist a traveling salesperson tour with cost $\leq k$?
- Using binary search and the decision version of the problem we can find the optimal TSP solution.

Reductions

- Reducing (transforming) one algorithm problem A to another problem B is an argument that if you can figure out how to solve B then you can solve A.

Reductions

- Reducing (transforming) one algorithm problem A to another problem B is an argument that if you can figure out how to solve B then you can solve A.
- We showed that many algorithmic problems are reducible to sorting (e.g. element uniqueness, mode, etc.).

Reductions

- Reducing (transforming) one algorithm problem A to another problem B is an argument that if you can figure out how to solve B then you can solve A.
- We showed that many algorithmic problems are reducible to sorting (e.g. element uniqueness, mode, etc.).
- A computer scientist and an engineer wanted some tea...

Satisfiability

Consider the following logic problem:
Instance: A set V of variables and a set of clauses C over V.
Question: Is there a truth assignment to V such that each clause in C is (simultaneously) satisfied?

Example 1

$V = v_1, v_2$ and $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$

- A clause is satisfied when at least one literal in it is TRUE.

Example 1

$V = v_1, v_2$ and $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$

- A clause is satisfied when at least one literal in it is TRUE.
- C is satisfied when $v_1 = v_2 =$TRUE.

Example 2

$V = v_1, v_2,$

$$C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$$

- Although you try, and you try, you can get no satisfaction!

Example 2

$$V = v_1, v_2,$$

$$C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$$

- Although you try, and you try, you can get no satisfaction!
- There is no satisfying assignment since v_1 must be FALSE (third clause), so v_2 must be FALSE (second clause), but then the first clause is unsatisfiable!

Satisfiability is Hard

- Satisfiability is known/assumed to be a hard problem.

Satisfiability is Hard

- Satisfiability is known/assumed to be a hard problem.
- Every top-notch algorithm expert in the world has tried and failed to come up with a fast algorithm to test whether a given set of clauses is satisfiable.

Satisfiability is Hard

- Satisfiability is known/assumed to be a hard problem.
- Every top-notch algorithm expert in the world has tried and failed to come up with a fast algorithm to test whether a given set of clauses is satisfiable.
- Satisfiability shown to be NP-complete by Cook.

3-Satisfiability

Instance: Same as SAT except that each clause contains exactly 3 literals.
Question: Is there a truth assignment to V so that each clause is satisfied?

- More restricted problem than SAT.

3-Satisfiability

Instance: Same as SAT except that each clause contains exactly 3 literals.
Question: Is there a truth assignment to V so that each clause is satisfied?

- More restricted problem than SAT.
- If 3-SAT is NP-complete, it implies SAT is NP-complete but not vice-versa, perhaps long clauses are what makes SAT difficult?!

3-Satisfiability

Instance: Same as SAT except that each clause contains exactly 3 literals.
Question: Is there a truth assignment to V so that each clause is satisfied?

- More restricted problem than SAT.
- If 3-SAT is NP-complete, it implies SAT is NP-complete but not vice-versa, perhaps long clauses are what makes SAT difficult?!
- After all, 1-SAT is trivial!

3-SAT is NP-Complete

To prove it is NP-complete, we give a reduction from $SAT \propto 3-SAT$. We will transform each clause independently based on its *length*. Suppose clause C_i contains k literals.

- If $k = 1$, meaning $C_i = \{z_1\}$, create two new variables v_1, v_2 and four new 3-literal clauses:

3-SAT is NP-Complete

To prove it is NP-complete, we give a reduction from $SAT \propto 3 - SAT$. We will transform each clause independently based on its *length*. Suppose clause C_i contains k literals.

- If $k = 1$, meaning $C_i = \{z_1\}$, create two new variables v_1, v_2 and four new 3-literal clauses: $\{v_1, v_2, z_1\}$, $\{v_1, \overline{v}_2, z_1\}$, $\{\overline{v}_1, v_2, z_1\}$, $\{\overline{v}_1, \overline{v}_2, z_1\}$.

3-SAT is NP-Complete

To prove it is NP-complete, we give a reduction from $SAT \propto 3 - SAT$. We will transform each clause independently based on its *length*. Suppose clause C_i contains k literals.

- If $k = 1$, meaning $C_i = \{z_1\}$, create two new variables v_1, v_2 and four new 3-literal clauses: $\{v_1, v_2, z_1\}$, $\{v_1, \overline{v}_2, z_1\}$, $\{\overline{v}_1, v_2, z_1\}$, $\{\overline{v}_1, \overline{v}_2, z_1\}$.
  Note that the only way all four of these can be satisfied is if z_1 is TRUE.

3-SAT is NP-Complete (Pg 2)

- If $k = 2$, meaning $\{z_1, z_2\}$, create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$

3-SAT is NP-Complete (Pg 2)

- If $k = 2$, meaning $\{z_1, z_2\}$, create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$
- If $k = 3$, meaning $\{z_1, z_2, z_3\}$, copy into the 3-SAT instance as it is.

Difficult Case: $k > 3$

Clause is $\{z_1, z_2, ..., z_k\}$: create $k - 3$ new variables and $k - 2$ new clauses in a chain:
$\{z_1, z_2, \overline{v_1}\}$, $\{v_1, z_3, \overline{v_2}\}$, $\{v_2, z_4, \overline{v_3}\}$, ...,
$\{v_{k-4}, z_{k-2}, \overline{v_{k-3}}\}$, $\{v_{k-3}, z_{k-1}, z_k\}$

Why does the Chain Work?

If none of the original variables in a clause are TRUE, there is no way to satisfy all of them using the additional variable:

$$(F, F, T), (F, F, T), \ldots, (F, F, F)$$

Why does the Chain Work?(2)

But if any literal is TRUE, we have $n - 3$ free variables and $n - 3$ remaining 3-clauses, so we can satisfy each of them.
$(F, F, T), (F, F, T), \ldots, (\mathbf{F}, \mathbf{T}, \mathbf{F}), \ldots, (T, F, F),$
(T, F, F)

SAT and 3-SAT instances are equivalent

Any SAT solution will also satisfy the 3-SAT instance and any 3-SAT solution sets ups a SAT solution, so the problems are equivalent.

Class Exercise

$(\overline{x}_1 \vee x_2 \vee x_4 \vee \overline{x}_7) \wedge (x_3 \vee \overline{x_5}) \wedge$
$(\overline{x}_2 \vee \overline{x}_3 \vee x_4 \vee \overline{x}_6 \vee x_8)$

- Find a satisfying truth assignment for the SAT instance above.
- Reduce the SAT instance above to a 3SAT instance using the method described in class.
- Find a satisfying truth assignment for the 3SAT instance.
- In your own time: repeat for SAT instance that is not satisfiable.

4-SAT and 2-SAT

- A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.

4-SAT and 2-SAT

- A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.
- However, it breaks down when we try to use it for 2-SAT.

4-SAT and 2-SAT

- A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.
- However, it breaks down when we try to use it for 2-SAT.
- Now that we've shown 3-SAT is NP-complete, we may use it for further reductions.

4-SAT and 2-SAT

- A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.
- However, it breaks down when we try to use it for 2-SAT.
- Now that we've shown 3-SAT is NP-complete, we may use it for further reductions.
- Since the set of 3-SAT instances is smaller and more regular than the *SAT* instances, it will be easier to use 3-SAT for future reductions.

4-SAT and 2-SAT

- A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.
- However, it breaks down when we try to use it for 2-SAT.
- Now that we've shown 3-SAT is NP-complete, we may use it for further reductions.
- Since the set of 3-SAT instances is smaller and more regular than the *SAT* instances, it will be easier to use 3-SAT for future reductions.
- Remember the direction of the reduction! SAT $\propto$ 3-SAT $\propto$ X

A Perpetual Point of Confusion

- We must transform *every* instance of a known NP-complete problem to an instance of the problem we are interested in.

A Perpetual Point of Confusion

- We must transform *every* instance of a known NP-complete problem to an instance of the problem we are interested in.
- If we do the reduction the other way, all we get is a slow way to solve X, by using a subroutine which probably will take exponential time.

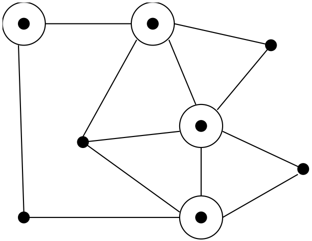
A Perpetual Point of Confusion

- We must transform *every* instance of a known NP-complete problem to an instance of the problem we are interested in.
- If we do the reduction the other way, all we get is a slow way to solve X, by using a subroutine which probably will take exponential time.
- This always is confusing at first - it seems backwards. Make sure you understand the direction of reduction now - and think back to this when you get confused.

Vertex Cover

Instance: A graph $G = (V, E)$, and integer $k \leq V$
Question: Is there a subset of at most k vertices such that every $e \in E$ has at least one vertex in the subset?

Example



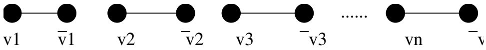
- It is trivial to find *a* vertex cover of a graph – just take all the vertices.
- The tricky part is to cover with as small a set as possible.

Vertex cover is NP-complete

To prove completeness, we reduce 3-SAT to VC. From a 3-SAT instance with N variables and C clauses, we construct a graph with $2N + 3C$ vertices.

Variable Gadgets

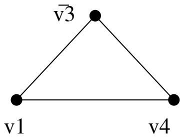
For each variable, we create two vertices connected by an edge:



To cover each of these edges, at least n vertices must be in the cover, one for each pair.

Clause Gadgets

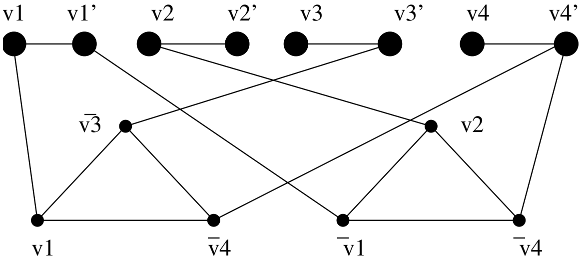
For each clause, we create three new vertices, one for each literal in each clause. Connect these in a triangle.



At least two vertices per triangle must be in the cover to take care of edges in the triangle, for a total of at least $2C$ vertices.

Putting it Together

Finally, we will connect each literal in the flat structure to the corresponding vertices in the triangles which share the same literal.



Claim: G has a vertex cover of size $N + 2C$ iff S is Satisfiable

Any cover of G must have at least $N + 2C$ vertices.
To show that our reduction is correct, we must show that:

1. *Every satisfying truth assignment gives a cover of size $N + 2C$.*
2. *Every vertex cover of size $N + 2C$ gives a satisfying truth assignment.*

Every satisfying truth assignment gives a cover of size $N + 2C$.

- Select the N vertices corresponding to the TRUE literals to be in the cover.
- Since it is a satisfying truth assignment, at least one of the three cross edges associated with each clause must already be covered - pick the other two vertices to complete the cover.

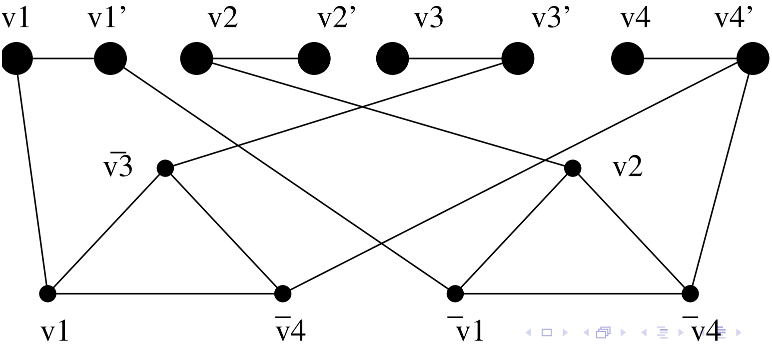
Every vertex cover of size $N + 2C$ gives a satisfying truth assignment

- Any vertex cover of size $N + 2C$ must contain N first stage vertices and $2C$ second stage vertices.
- Let the first stage vertices define the truth assignment.
- To give the cover, at least one cross-edge must be covered, so the truth assignment satisfies.

Example Reduction

Every SAT defines a cover and Every Cover Truth values for the SAT!

Example: $V_1 = V_2 = True$, $V_3 = V_4 = False$.



Starting from the Right Problem

$$3 - SAT \propto VC$$

As you can see, the reductions can be very clever and complicated. While theoretically any NP-complete problem will do, choosing the correct one can make it much easier.

NP-Completeness Tree

P and NP

- Precise distinction is somewhat technical, requiring formal language theory and Turing machines to state correctly.

P and NP

- Precise distinction is somewhat technical, requiring formal language theory and Turing machines to state correctly.
- However, intuitively a problem is in P, (ie. polynomial) if it can be solved in time polynomial in the size of the input.

P and NP

- Precise distinction is somewhat technical, requiring formal language theory and Turing machines to state correctly.
- However, intuitively a problem is in P, (ie. polynomial) if it can be solved in time polynomial in the size of the input.
- A problem is in NP (i.e., non-deterministically polynomial) if, given the answer, it is possible to verify that the answer is correct within time polynomial in the size of the input.

NP examples

- Satisfiability: if we are given an assignment of T/F to the variables, can we check for satisfiability in polynomial time?

NP examples

- Satisfiability: if we are given an assignment of T/F to the variables, can we check for satisfiability in polynomial time?
- Vertex Cover: given a set of vertices, can we check whether it is a vertex cover of size $\leq k$.

P versus NP: intuition

"If you have ever attempted to solve a crossword puzzle, you know that it is much harder to solve it from scratch than to verify a solution provided by someone else. The usual explanation for this difference of effort is that finding a solution to a crossword puzzle requires *creative* effort. Verifying a solution is much easier since someone else has already done the creative part."

P versus NP: what we know

- Technically, P and NP are sets of problems.

P versus NP: what we know

- Technically, P and NP are sets of problems.
- P: problems that can be solved in poly time.

P versus NP: what we know

- Technically, P and NP are sets of problems.
- P: problems that can be solved in poly time.
- NP: problems that can be verified in poly time.

P versus NP: what we know

- Technically, P and NP are sets of problems.
- P: problems that can be solved in poly time.
- NP: problems that can be verified in poly time.
- If you can solve a problem in polynomial time, you can also verify it in polynomial time. (Actual argument is a bit more technical, but this is the essence of the idea.)

P versus NP: what we know

- Technically, P and NP are sets of problems.
- P: problems that can be solved in poly time.
- NP: problems that can be verified in poly time.
- If you can solve a problem in polynomial time, you can also verify it in polynomial time. (Actual argument is a bit more technical, but this is the essence of the idea.)
- So $P \subseteq NP$.

P versus NP: what we don't know

If $P \subseteq NP$, then either

1. $P = NP$
2. $P \subset NP$

We don't know which one of these is true.

NP-Complete Definition

A problem X is NP-complete if

1. ALL problems in $NP \propto X$.
2. X is in NP.

The “Get Rich and Famous” Theorem

Theorem

If any NPC problem is in P, then $P = NP$

The "Get Famous" Theorem

Theorem

If any NP problem can be proven to not be in P, then every NPC problem is not in P.

Formal Basis for our NPC Proofs

Theorem

If an NPC problem $Y \propto X$ and X is in NP, then X is NPC.