BBM402-Lecture 8: NP Completeness

Lecturer: Lale Özkahya

Resources for the presentation:
https://courses.engr.illinois.edu/cs473/fa2016/lectures.html
https://courses.engr.illinois.edu/cs374/fa2015/lectures.html

Part I

NP-Completeness

P and **NP** and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time verification algorithms.

- Many natural problems we would like to solve are in NP.
- Every problem in NP has an exponential time algorithm (try verifying each possible certificate).
- $P \subseteq NP$
- So some problems in NP are in P (example, shortest path problem)

Big Question: Does every problem in NP have an efficient algorithm? Same as asking whether $P = NP$.

“Hardest” Problems

Question

What is the hardest problem in **NP**? How do we define it?

Towards a definition

1. Hardest problem must be in **NP**.
2. Hardest problem must be at least as “difficult” as every other problem in **NP**.

NP-Complete Problems

Definition

A problem X is said to be **NP-Complete** if

1. $X \in \mathbf{NP}$, and
2. (Hardness) For any $Y \in \mathbf{NP}$, $\mathbf{Y} \leq_P \mathbf{X}$.

NP-Complete Problems

Definition

A problem X is said to be **NP-Complete** if

1. $X \in \mathbf{NP}$, and
2. (Hardness) For any $Y \in \mathbf{NP}$, $\mathbf{Y} \leq_P \mathbf{X}$.

Recall reduction: $\mathbf{Y} \leq_P \mathbf{X}$ means that an instance of Y can be efficiently modeled as an instance of X.

Solving NP-Complete Problems

Proposition

*Suppose X is **NP-Complete**. Then X can be solved in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.*

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

NP-Hard Problems

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in \mathbf{NP}$, we have that $\mathbf{Y} \leq_P \mathbf{X}$.

An **NP-Hard** problem need not be in **NP**!

Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

Consequences of proving **NP-Complete**ness

If X is **NP-Complete**

1. Since we believe **P** $\neq$ **NP**,
2. and solving X implies **P** $=$ **NP**.

Consequences of proving **NP-Complete**ness

If X is **NP-Complete**

1. Since we believe **P** $\neq$ **NP**,
2. and solving X implies **P** $=$ **NP**.

X is unlikely to be efficiently solvable.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.
(This is proof by mob opinion — take with a grain of salt.)

NP-Complete Problems

Question

Are there any "natural" problems that are **NP-Complete**?

Answer

Yes! Many, many important problems are **NP-Complete**.

Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* **NP-Complete**.

Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* **NP-Complete**.

Need to show

1. **SAT** is in **NP**.
2. *every* **NP** problem X reduces to **SAT**.

Will see proof in next lecture.

Steve Cook won the Turing award for his theorem.

Proving that a problem X is NP-Complete

To prove X is **NP-Complete**, show

1. Show that X is in **NP**.
2. Give a polynomial-time reduction *from* a known **NP-Complete** problem such as **SAT** *to* X

Proving that a problem X is NP-Complete

To prove X is **NP-Complete**, show

1. Show that X is in **NP**.
2. Give a polynomial-time reduction *from* a known **NP-Complete** problem such as **SAT** *to* X

SAT $\leq_P X$ implies that every **NP** problem $Y \leq_P X$. Why?

Proving that a problem X is NP-Complete

To prove X is **NP-Complete**, show

1. Show that X is in **NP**.
2. Give a polynomial-time reduction *from* a known **NP-Complete** problem such as **SAT** *to* X

SAT $\leq_P X$ implies that every **NP** problem $Y \leq_P X$. Why?
Transitivity of reductions:

$Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

3-SAT is NP-Complete

- **3-SAT** is in NP
- **SAT** $\leq_P$ **3-SAT** as we saw

NP-Completeness via Reductions

1. **SAT** is **NP-Complete** due to Cook-Levin theorem
2. **SAT** $\leq_P$ **3-SAT**
3. **3-SAT** $\leq_P$ **Independent Set**
4. **Independent Set** $\leq_P$ **Vertex Cover**
5. **Independent Set** $\leq_P$ **Clique**
6. **3-SAT** $\leq_P$ **3-Color**
7. **3-SAT** $\leq_P$ **Hamiltonian Cycle**

NP-Completeness via Reductions

1. **SAT** is **NP-Complete** due to Cook-Levin theorem
2. **SAT** $\leq_P$ **3-SAT**
3. **3-SAT** $\leq_P$ **Independent Set**
4. **Independent Set** $\leq_P$ **Vertex Cover**
5. **Independent Set** $\leq_P$ **Clique**
6. **3-SAT** $\leq_P$ **3-Color**
7. **3-SAT** $\leq_P$ **Hamiltonian Cycle**

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be **NP-Complete**.

A surprisingly frequent phenomenon!

NP-Completeness via Reductions

Part II

Reducing **3-SAT** to **Independent Set**

Independent Set

Problem: Independent Set

> **Instance:** A graph G, integer k.
> **Question:** Is there an independent set in G of size k?

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ

Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.
G_φ should be constructable in time polynomial in size of φ

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.
G_φ should be constructable in time polynomial in size of φ

Importance of reduction: Although **3SAT** is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.

Notice: We handle only $3\mathrm{CNF}$ formulas – reduction would not work for other kinds of boolean formulas.

Interpreting 3SAT

There are two ways to think about **3SAT**

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick x_i and $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

The Reduction

1. G_φ will have one vertex for each literal in a clause

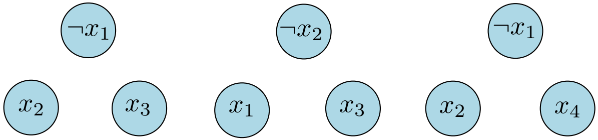


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

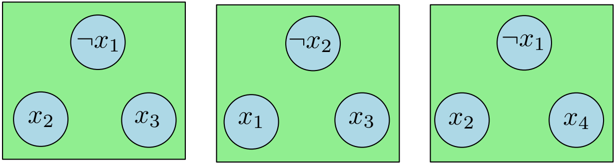


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

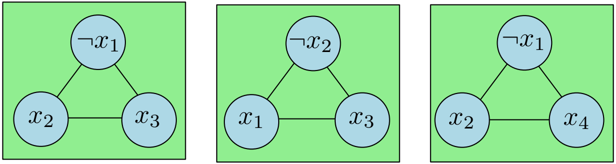


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict

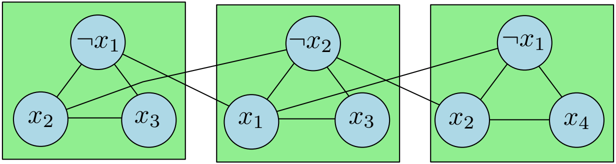


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
4. Take k to be the number of clauses

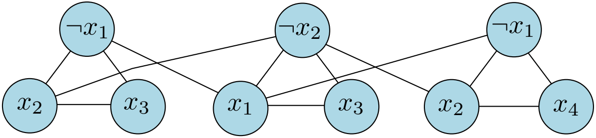


Figure: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

1. Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size. Why? $\square$

Correctness (contd)

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Leftarrow$ Let S be an independent set of size k

1. S must contain *exactly* one vertex from each clause
2. S cannot contain vertices labeled by conflicting literals
3. Thus, it is possible to obtain a truth assignment that makes in the literals in S true; such an assignment satisfies one literal in every clause $\square$

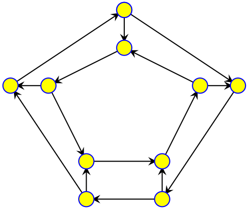
Part III

NP-Completeness of Hamiltonian Cycle

Directed Hamiltonian Cycle

Input Given a directed graph $G = (V, E)$ with n vertices

Goal Does G have a Hamiltonian cycle?

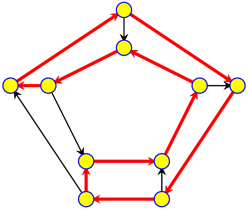


Directed Hamiltonian Cycle

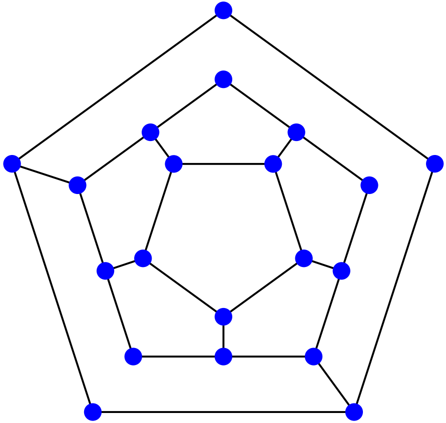
Input Given a directed graph $G = (V, E)$ with n vertices

Goal Does G have a Hamiltonian cycle?

- A Hamiltonian cycle is a cycle in the graph that visits every vertex in G exactly once



Is the following graph Hamiltonian?



(A) Yes.

(B) No.

Directed Hamiltonian Cycle is **NP-Complete**

- Directed Hamiltonian Cycle is in NP: Why?
- Hardness: We will show
  3-SAT $\leq_P$ Directed Hamiltonian Cycle

Reduction

Given 3-SAT formula φ create a graph G_φ such that

- G_φ has a Hamiltonian cycle if and only if φ is satisfiable
- G_φ should be constructible from φ by a polynomial time algorithm $\mathcal{A}$

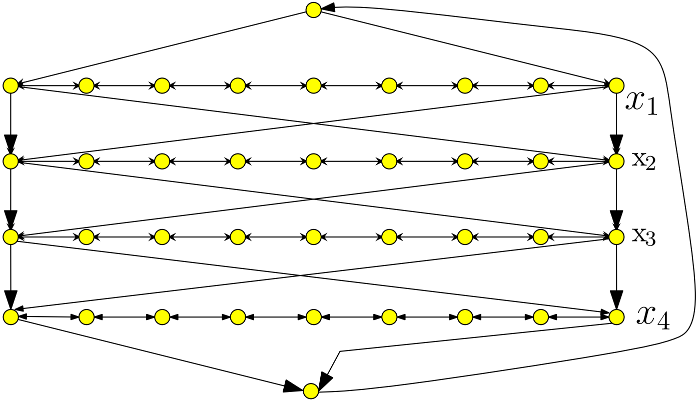
Notation: φ has n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$.

Reduction: First Ideas

- Viewing SAT: Assign values to n variables, and each clause has multiple ways in which it can be satisfied.
- Construct graph with 2^n Hamiltonian cycles, where each cycle corresponds to some boolean assignment.
- Then add more graph structure to encode constraints on assignments imposed by the clauses.

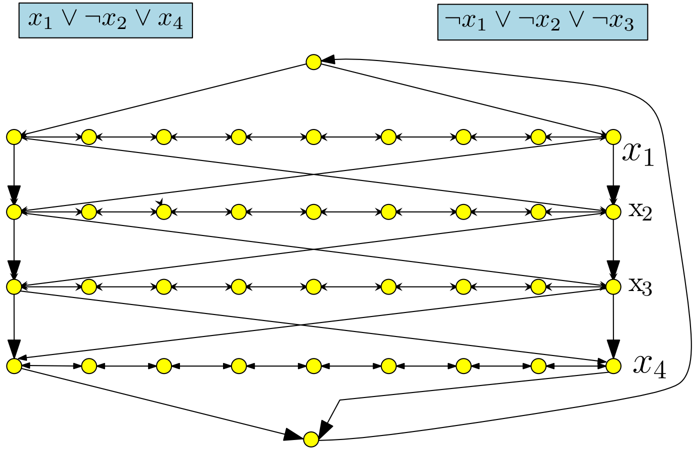
The Reduction: Phase I

- Traverse path i from left to right iff x_i is set to true
- Each path has $3(m+1)$ nodes where m is number of clauses in φ; nodes numbered from left to right (1 to $3m+3$)



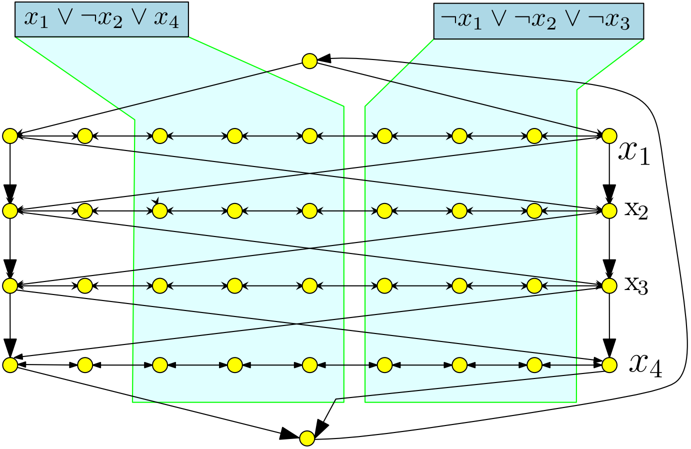
The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.



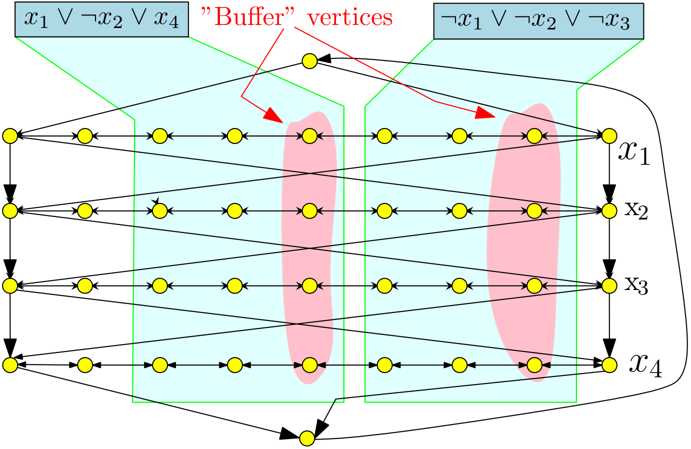
The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path i if x_i appears in clause C_j, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\neg x_i$ appears in C_j.



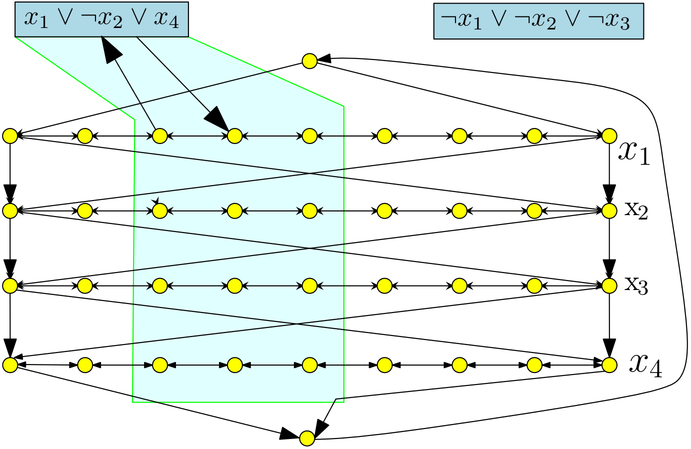
The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.



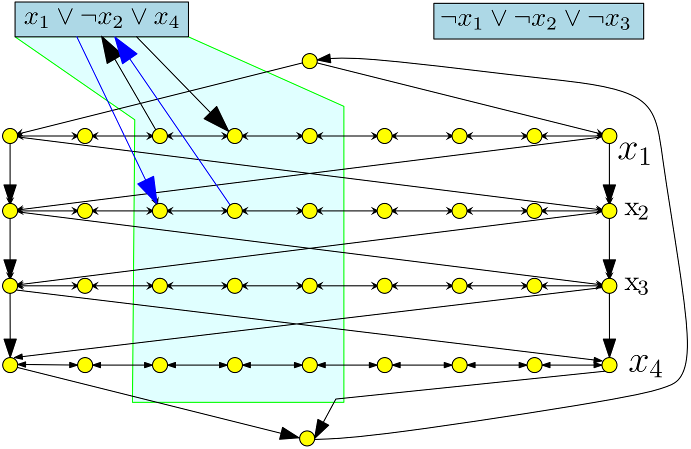
The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.



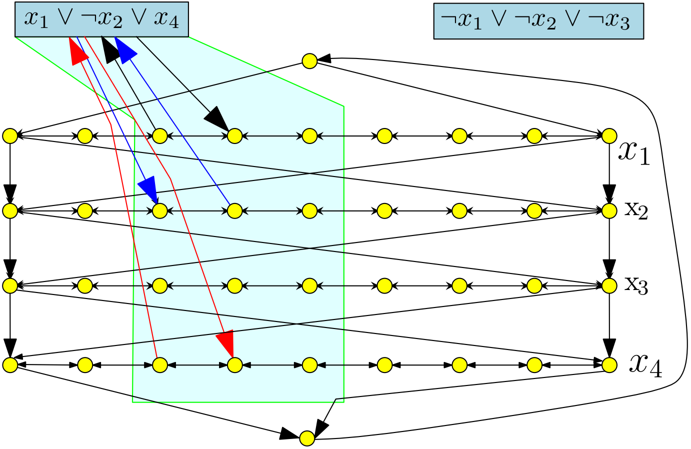
The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.

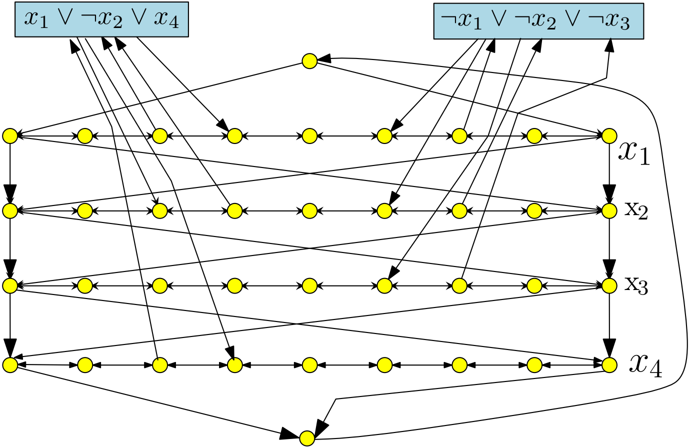


The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.



- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.



Correctness Proof

Proposition

φ has a satisfying assignment iff G_φ has a Hamiltonian cycle.

Proof.

$\Rightarrow$ Let a be the satisfying assignment for φ. Define Hamiltonian cycle as follows

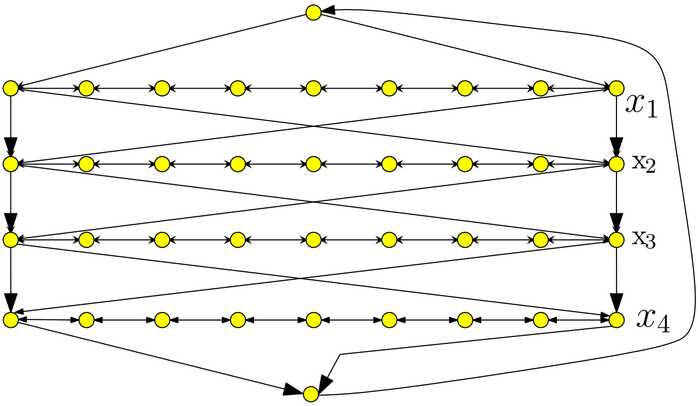
- If $a(x_i) = 1$ then traverse path i from left to right
- If $a(x_i) = 0$ then traverse path i from right to left
- For each clause, path of at least one variable is in the "right" direction to splice in the node corresponding to clause □

Hamiltonian Cycle $\Rightarrow$ Satisfying assignment

Suppose Π is a Hamiltonian cycle in G_φ

- If Π enters c_j (vertex for clause C_j) from vertex $3j$ on path i then it must leave the clause vertex on edge to $3j+1$ on the *same path* i
  - If not, then only unvisited neighbor of $3j+1$ on path i is $3j+2$
  - Thus, we don't have two unvisited neighbors (one to enter from, and the other to leave) to have a Hamiltonian Cycle
- Similarly, if Π enters c_j from vertex $3j+1$ on path i then it must leave the clause vertex c_j on edge to $3j$ on path i

Example



Hamiltonian Cycle $\implies$ Satisfying assignment (contd)

- Thus, vertices visited immediately before and after C_i are connected by an edge
- We can remove c_j from cycle, and get Hamiltonian cycle in $G - c_j$
- Consider Hamiltonian cycle in $G - \{c_1, \ldots c_m\}$; it traverses each path in only one direction, which determines the truth assignment

Hamiltonian Cycle

Problem

Input *Given* *undirected* *graph* $G = (V, E)$

Goal *Does* G *have a Hamiltonian cycle? That is, is there a cycle that visits every vertex exactly one (except start and end vertex)?*

NP-Completeness

Theorem

Hamiltonian cycle *problem for* ***undirected*** *graphs is* ***NP-Complete****.*

Proof.

- The problem is in **NP**; proof left as exercise.
- Hardness proved by reducing Directed Hamiltonian Cycle to this problem □

Reduction Sketch

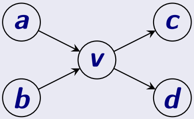
Goal: Given directed graph G, need to construct undirected graph G' such that G has Hamiltonian cycle iff G' has Hamiltonian cycle

Reduction Sketch

Goal: Given directed graph G, need to construct undirected graph G' such that G has Hamiltonian cycle iff G' has Hamiltonian cycle

Reduction

- Replace each vertex v by 3 vertices: v_{in}, v, and v_{out}

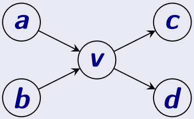


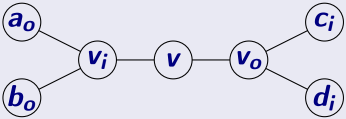
Reduction Sketch

Goal: Given directed graph G, need to construct undirected graph G' such that G has Hamiltonian cycle iff G' has Hamiltonian cycle

Reduction

- Replace each vertex v by 3 vertices: v_{in}, v, and v_{out}



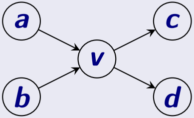


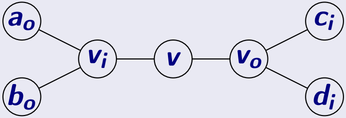
Reduction Sketch

Goal: Given directed graph G, need to construct undirected graph G' such that G has Hamiltonian cycle iff G' has Hamiltonian cycle

Reduction

- Replace each vertex v by 3 vertices: v_{in}, v, and v_{out}
- A directed edge (x, y) is replaced by edge (x_{out}, y_{in})





Reduction: Wrapup

- The reduction is polynomial time (exercise)
- The reduction is correct (exercise)

Part I

Reductions Continued

A **polynomial time reduction** from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ that has the following properties:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of I_Y) is polynomial in $|I_X|$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Notation: $X \leq_P Y$ if X reduces to Y

Proposition

If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.

Such a reduction is called a **Karp reduction**. Most reductions we will need are Karp reductions.

A More General Reduction

Turing Reduction

Definition (Turing reduction.)

Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:

1. on any given instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

A More General Reduction

Turing Reduction

Definition (Turing reduction.)

Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:

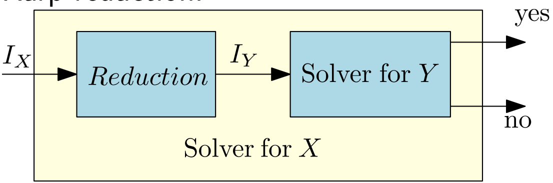
1. on any given instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

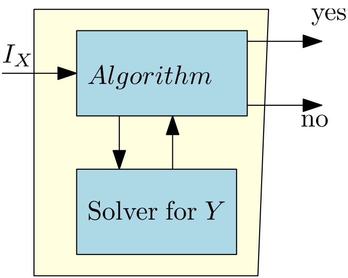
Note: In making sub-routine call to algorithm to solve Y, $\mathcal{A}$ can only ask questions of size polynomial in $|I_X|$. Why?

Comparing reductions

1. Karp reduction:



2. Turing reduction:



Turing reduction

1. Algorithm to solve X can call solver for Y many times.
2. Conceptually, every call to the solver of Y takes constant time.

Relation between reductions

Consider two problems X and Y. Which of the following statements is correct?

(A) If there is a Turing reduction from X to Y, then there is a Karp reduction from X to Y.

(B) If there is a Karp reduction from X to Y, then there is a Turing reduction from X to Y.

(C) If there is a Karp reduction from X to Y, then there is a Karp reduction from Y to X.

(D) If there is a Turing reduction from X to Y, then there is a Turing reduction from Y to X.

(E) All of the above.

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How? Used algorithm for interval problem multiple times.

Turing vs Karp Reductions

1. Turing reductions more general than Karp reductions.
2. Turing reduction useful in obtaining algorithms via reductions.
3. Karp reduction is simpler and easier to use to prove hardness of problems.
4. Perhaps surprisingly, Karp reductions, although limited, suffice for most known **NP-Complete**ness proofs.
5. Karp reductions allow us to distinguish between **NP** and **co-NP** (more on this later).

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \lor x_2 \lor \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is a CNF formula.

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \lor x_2 \lor \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is a CNF formula.
4. A formula φ is a $3\mathrm{CNF}$:
   A CNF formula such that every clause has **exactly** 3 literals.
   1. $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3 \lor x_1)$ is a $3\mathrm{CNF}$ formula, but $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is not.

Satisfiability

Problem: SAT

Instance: A CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Problem: 3SAT

Instance: A $3\mathrm{CNF}$ formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Satisfiability

SAT

Given a CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

Example

1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true
2. $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is not satisfiable.

3SAT

Given a 3CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

(More on **2SAT** in a bit...)

Importance of SAT and 3SAT

1. **SAT** and **3SAT** are basic constraint satisfaction problems.
2. Many different problems can reduced to them because of the simple yet powerful expressively of logical constraints.
3. Arise naturally in many applications involving hardware and software verification and correctness.
4. As we will see, it is a fundamental problem in theory of **NP-Complete**ness.

3SAT $\leq_P$ SAT

1. **3SAT** $\leq_P$ **SAT**.
2. Because...
   A **3SAT** instance is also an instance of **SAT**.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ **3SAT**.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

1. φ is satisfiable iff φ' is satisfiable.
2. φ' can be constructed from φ in time polynomial in $|\varphi|$.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ **3SAT**.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

1. φ is satisfiable iff φ' is satisfiable.
2. φ' can be constructed from φ in time polynomial in $|\varphi|$.

Idea: if a clause of φ is not of length $\mathbf{3}$, replace it with several clauses of length exactly $\mathbf{3}$.

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

In **SAT** clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

$$\left(x \vee y \vee z \vee w \vee u\right) \wedge \left(\neg x \vee \neg y \vee \neg z \vee w \vee u\right) \wedge \left(\neg x\right)$$

In **3SAT** every clause must have **exactly** **3** different literals.

To reduce from an instance of **SAT** to an instance of **3SAT**, we must make all clauses to have exactly **3** variables...

Basic idea

1. Pad short clauses so they have **3** literals.
2. Break long clauses into shorter clauses.
3. Repeat the above till we have a 3CNF.
   Note: Need to add new variables.

What about 2SAT?

2SAT can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from **SAT** (or **3SAT**) to **2SAT**. If there was, then **SAT** and **3SAT** would be solvable in polynomial time.

Why the reduction from 3SAT to 2SAT fails?

Consider a clause $(x \vee y \vee z)$. We need to reduce it to a collection of $2\mathrm{CNF}$ clauses. Introduce a face variable α, and rewrite this as

$$
\begin{array}{lll}
 & (x \vee y \vee \alpha) \wedge (\neg\alpha \vee z) & \text{(bad! clause with 3 vars)} \\
\text{or} & (x \vee \alpha) \wedge (\neg\alpha \vee y \vee z) & \text{(bad! clause with 3 vars).}
\end{array}
$$

(In animal farm language: **2SAT** good, **3SAT** bad.)

What about **2SAT**?

A challenging exercise: Given a **2SAT** formula show to compute its satisfying assignment...

Look in books etc.

Independent Set

Problem: Independent Set

Instance: A graph G, integer k.   **Question:** Is there an independent set in G of size k?

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ

Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

G_φ should be constructable in time polynomial in size of φ

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.
G_φ should be constructable in time polynomial in size of φ

Importance of reduction: Although **3SAT** is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.

Interpreting 3SAT

There are two ways to think about **3SAT**

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick x_i and $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

The Reduction

1. G_φ will have one vertex for each literal in a clause

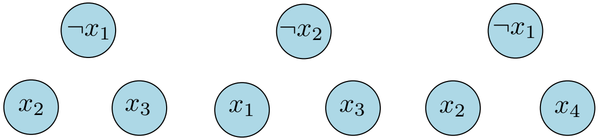


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

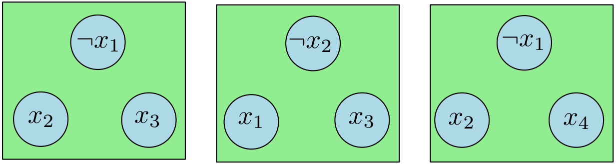


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

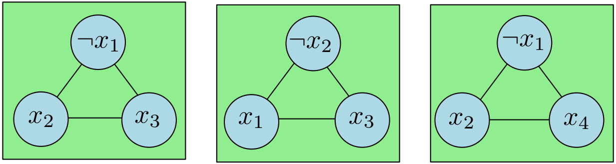


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict

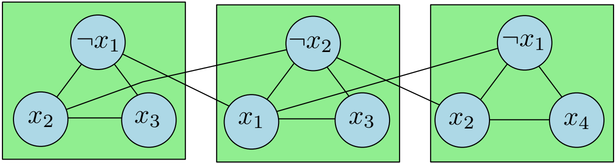


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
4. Take k to be the number of clauses

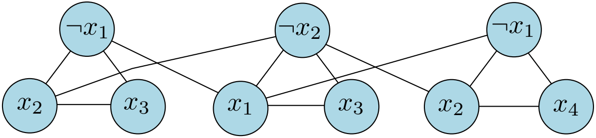


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

1. Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size $\square$

Correctness (contd)

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Leftarrow$ Let S be an independent set of size k

1. S must contain exactly one vertex from each clause
2. S cannot contain vertices labeled by conflicting clauses
3. Thus, it is possible to obtain a truth assignment that makes in the literals in S true; such an assignment satisfies one literal in every clause

$\square$

Transitivity of Reductions

Lemma

$X \le_P Y$ and $Y \le_P Z$ implies that $X \le_P Z$.

Note: $X \le_P Y$ does not imply that $Y \le_P X$ and hence it is very important to know the FROM and TO in a reduction.

To prove $X \le_P Y$ you need to show a reduction FROM X TO Y
In other words show that an algorithm for Y implies an algorithm for X.

Part II

Definition of NP

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

3SAT $\leq_P$ **Independent Set**

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

3SAT $\leq_P$ **Independent Set** $\overset{\leq_P}{\geq_P}$ **Vertex Cover**

Recap ...

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

3SAT $\leq_P$ **Independent Set** $\overset{\leq_P}{\geq_P}$ **Vertex Cover** $\leq_P$ **Set Cover**

Recap ...

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

3SAT $\leq_P$ **Independent Set** $\overset{\leq_P}{\geq_P}$ **Vertex Cover** $\leq_P$ **Set Cover**
3SAT $\leq_P$ **SAT** $\leq_P$ **3SAT**

Problems and Algorithms: Formal Approach

Decision Problems

1. Problem Instance: Binary string s, with size $|s|$
2. Problem: A set X of strings on which the answer should be "yes"; we call these YES instances of X. Strings not in X are NO instances of X.

Definition

1. A is an algorithm for problem X if $A(s) =$ "yes" iff $s \in X$.
2. A is said to have a polynomial running time if there is a polynomial $p(\cdot)$ such that for every string s, $A(s)$ terminates in at most $O(p(|s|))$ steps.

Polynomial Time

Definition

Polynomial time (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

Polynomial Time

Definition

Polynomial time (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

Example

Problems in **P** include

1. Is there a shortest path from s to t of length $\leq k$ in G?
2. Is there a flow of value $\geq k$ in network G?
3. Is there an assignment to variables to satisfy given linear constraints?

Efficiency Hypothesis

A problem $\boldsymbol{X}$ *has an efficient algorithm iff* $\boldsymbol{X} \in \mathbf{P}$, *that is* $\boldsymbol{X}$ *has a polynomial time algorithm*.

Justifications:

1. Robustness of definition to variations in machines.
2. A sound theoretical definition.
3. Most known polynomial time algorithms for “natural” problems have small polynomial running times.

Problems with no known polynomial time algorithms

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

Question: What is common to above problems?

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X *of* X *there is a proof/certificate/solution that is of length poly(*$|I_X|$*) such that given a proof one can efficiently check that* I_X *is indeed a YES instance.*

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k: a subset S of vertices.

Certifiers

Definition

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if for every $s \in X$ there is some string t such that $C(s, t) =$ "yes", and conversely, if for some s and t, $C(s, t) =$ "yes" then $s \in X$.
The string t is called a certificate or proof for s.

Certifiers

Definition

An algorithm $C(\cdot,\cdot)$ is a **certifier** for problem X if for every $s \in X$ there is some string t such that $C(s,t) =$ "yes", and conversely, if for some s and t, $C(s,t) =$ "yes" then $s \in X$.
The string t is called a certificate or proof for s.

Definition (Efficient Certifier.)

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that for every string s, we have that

- $s \in X$ if and only if
- there is a string t:
  1. $|t| \leq p(|s|)$,
  2. $C(s,t) =$ "yes",
  3. and C runs in polynomial time.

Example: Independent Set

1. Problem: Does $G = (V, E)$ have an independent set of size $\geq k$?
    1. Certificate: Set $S \subseteq V$.
    2. Certifier: Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.

Example: Vertex Cover

1. Problem: Does G have a vertex cover of size $\leq k$?
   1. Certificate: $S \subseteq V$.
   2. Certifier: Check $|S| \leq k$ and that for every edge at least one endpoint is in S.

Example: SAT

1. Problem: Does formula φ have a satisfying truth assignment?
   1. Certificate: Assignment $\boldsymbol{a}$ of $\mathbf{0/1}$ values to each variable.
   2. Certifier: Check each clause under $\boldsymbol{a}$ and say “yes” if all clauses are true.

Example: Composites

Problem: Composite

Instance: A number s.   **Question:** Is the number s a composite?

1. Problem: **Composite**.
    1. Certificate: A factor $t \leq s$ such that $t \neq 1$ and $t \neq s$.
    2. Certifier: Check that t divides s.

Not composite?

Problem: Not Composite

Instance: A number s.
Question: Is the number s not a composite?

The problem **Not Composite** is

- **(A)** Can be solved in linear time.
- **(B)** in **P**.
- **(C)** Can be solved in exponential time.
- **(D)** Does not have a certificate or an efficient certifier.
- **(E)** The status of this problem is still open.

Example: A String Problem

Problem: PCP

> **Instance:** Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
> **Question:** Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
   1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
   2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

Example: A String Problem

Problem: PCP

Instance: Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
Question: Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
   1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
   2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

PCP = Posts Correspondence Problem and it is undecidable!
Implies no finite bound on length of certificate!

Nondeterministic Polynomial Time

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Nondeterministic Polynomial Time

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Example

Independent Set, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, and **Composite** are all examples of problems in **NP**.

Why is it called...

Nondeterministic Polynomial Time

A certifier is an algorithm $C(I, c)$ with two inputs:

1. I: instance.
2. c: proof/certificate that the instance is indeed a YES instance of the given problem.

One can think about C as an algorithm for the original problem, if:

1. Given I, the algorithm guesses (non-deterministically, and who knows how) a certificate c.
2. The algorithm now verifies the certificate c for the instance I.

NP can be equivalently described using Turing machines.

Asymmetry in Definition of NP

Note that only YES instances have a short proof/certificate. NO instances need not have a short certificate.

Example

SAT formula φ. No easy way to prove that φ is NOT satisfiable!

More on this and **co-NP** later on.

$\mathbf{P}$ versus $\mathbf{NP}$

Proposition

$\mathbf{P} \subseteq \mathbf{NP}$.

P versus NP

Proposition

$\mathbf{P} \subseteq \mathbf{NP}$.

For a problem in **P** no need for a certificate!

Proof.

Consider problem $X \in \mathbf{P}$ with algorithm A. Need to demonstrate that X has an efficient certifier:

1. Certifier C on input s, t, runs $A(s)$ and returns the answer.
2. C runs in polynomial time.
3. If $s \in X$, then for every t, $C(s, t) =$ "yes".
4. If $s \notin X$, then for every t, $C(s, t) =$ "no". ∎

Exponential Time

Definition

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\mathrm{poly}(|s|)})$.

Exponential Time

Definition

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\mathrm{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O(2^{n^3})$, ...

NP versus **EXP**

Proposition

NP $\subseteq$ **EXP**.

Proof.

Let $X \in$ **NP** with certifier C. Need to design an exponential time algorithm for X.

1. For every t, with $|t| \leq p(|s|)$ run $C(s,t)$; answer "yes" if any one of these calls returns "yes".
2. The above algorithm correctly solves X (exercise).
3. Algorithm runs in $O(q(|s| + |p(s)|)2^{p(|s|)})$, where q is the running time of C. $\square$

Examples

1. **SAT**: try all possible truth assignment to variables.
2. **Independent Set**: try all possible subsets of vertices.
3. **Vertex Cover**: try all possible subsets of vertices.

Is **NP** efficiently solvable?

We know **P** $\subseteq$ **NP** $\subseteq$ **EXP**.

Is NP efficiently solvable?

We know **P** $\subseteq$ **NP** $\subseteq$ **EXP**.

Big Question

Is there are problem in **NP** that does not belong to **P**? Is **P** $=$ **NP**?

If $\mathsf{P} = \mathsf{NP}$ . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.

If P = NP . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.

If $\mathsf{P} = \mathsf{NP}\ldots$

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.

If $\mathsf{P} = \mathsf{NP}$ . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .

If $\mathsf{P} = \mathsf{NP}$ ...

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce ...
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If $\mathsf{P} = \mathsf{NP}$ this implies that...

(A) **Vertex Cover** can be solved in polynomial time.

(B) $\mathsf{P} = \mathsf{EXP}$.

(C) $\mathsf{EXP} \subseteq \mathsf{P}$.

(D) All of the above.

P versus **NP**

Status

Relationship between **P** and **NP** remains one of the most important open problems in mathematics/computer science.

Consensus: Most people feel/believe $P \neq NP$.

Resolving **P** versus **NP** is a Clay Millennium Prize Problem. You can win a million dollars in addition to a Turing award and major fame!

Linear Programming in **NP**

Is LP in NP? Recall LP in (one) standard form is $\max cx, Ax \leq b$.

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m\times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq K$? Input has $n + mn + m + 1$ numbers.

- What is the certificate?
- What is the certifier?

Linear Programming in NP

Is LP in NP? Recall LP in (one) standard form is $\max cx, Ax \leq b$.

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq K$? Input has $n + mn + m + 1$ numbers.

- What is the certificate?
- What is the certifier?

Certificate: A solution $y \in \mathbb{R}^n$ consisting of n numbers?
Certifier: Check that $Ay \leq b$ and that $cy \geq K$

Linear Programming in NP

Is LP in NP? Recall LP in (one) standard form is $\max cx, Ax \leq b$.

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq K$? Input has $n + mn + m + 1$ numbers.

- What is the certificate?
- What is the certifier?

Certificate: A solution $y \in \mathbb{R}^n$ consisting of n numbers?
Certifier: Check that $Ay \leq b$ and that $cy \geq K$

Caveat: What is the representation size of y? Are we even guaranteed rational numbers? How many bits do we need to represent y and is it polynomial in the input size?

Linear Programming in NP

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq B$?

Assume for simplicity that $Ax \leq b$ defines a bounded polytope

- there is an optimum solution x^* which is a vertex
- x^* is defined as the unique solution to $A'x = b'$ where A' is a full-rank sub-matrix of A and b' is the corresponding sub-vector of b
- thus $x^* = (A')^{-1}b' = \frac{1}{\det(A')}(\text{adjoint}(A'))^T b'$

Linear Programming in **NP**

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq B$?

Assume for simplicity that $Ax \leq b$ defines a bounded polytope

- there is an optimum solution x^* which is a vertex
- x^* is defined as the unique solution to $A'x = b'$ where A' is a full-rank sub-matrix of A and b' is the corresponding sub-vector of b
- thus $x^* = (A')^{-1}b' = \frac{1}{\det(A')}(\text{adjoint}(A'))^T b'$

Main question: How many bits does $\det(A)$ have as a function of numbers in A?

Linear Programming in NP

Main question: How many bits does $\det(A)$ have as a function of numbers in A?

One definition of determinant of a $n \times n$ matrix A is:

$$\det(A) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \prod_{i=1}^{n} A_{i\sigma(i)}$$

Here S_n is the set of all $n!$ permutations of $\{1, 2, \ldots, n\}$ and $\text{sign}(\sigma) \in \{-1, 1\}$ is the signature of σ depending on whether σ can be obtained by odd or even number of transpositions.

Therefore $|\det(A)| \leq n! \times (\max_{ij} |A_{ij}|)^n$ and hence

$$\log |\det(A)| \leq n \log n + n \log(\max_{ij} |A_{ij}|)$$

Integer Programming in NP

Is IP in NP? Recall IP in (one) standard form is
$\max cx, Ax \leq b, x \in \mathbb{Z}^n$.

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq K$? Input has $n + mn + m + 1$ numbers.
Certificate: A solution $y \in \mathbb{R}^n$ consisting of n numbers?
Certifier: Check that $Ay \leq b$ and that $cy \geq K$

Integer Programming in NP

Is IP in NP? Recall IP in (one) standard form is
$\max cx, Ax \leq b, x \in \mathbb{Z}^n$.

Given c, A, b where $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$ and integer K, is optimum value $\geq K$? Input has $n + mn + m + 1$ numbers.
Certificate: A solution $y \in \mathbb{R}^n$ consisting of n numbers?
Certifier: Check that $Ay \leq b$ and that $cy \geq K$

Caveat: What is the representation size of y? How many bits do we need to represent y and is it polynomial in the input size? Note that unlike LP y is not necessarily a vertex of the polytope defined by $Ax \leq b$. Can be in the interior.

Need some advanced tools to prove that there always exists a y with representation size polynomial in input size.

Part III

NP-Completeness and Cook-Levin Theorem

"Hardest" Problems

Question

What is the hardest problem in **NP**? How do we define it?

Towards a definition

1. Hardest problem must be in **NP**.
2. Hardest problem must be at least as "difficult" as every other problem in **NP**.

NP-Complete Problems

Definition

A problem X is said to be **NP-Complete** if

1. $X \in$ **NP**, and
2. (Hardness) For any $Y \in$ **NP**, $\mathbf{Y} \leq_P \mathbf{X}$.

Solving NP-Complete Problems

Proposition

Suppose X is **NP-Complete***. Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

NP-Hard Problems

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in \mathbf{NP}$, we have that $\mathbf{Y} \leq_P \mathbf{X}$.

An **NP-Hard** problem need not be in **NP**!

Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

Consequences of proving **NP-Complete**ness

If X is **NP-Complete**

1. Since we believe **P** $\neq$ **NP**,
2. and solving X implies **P** $=$ **NP**.

X is unlikely to be efficiently solvable.

Consequences of proving **NP-Complete**ness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.

Consequences of proving **NP-Complete**ness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.
(This is proof by mob opinion — take with a grain of salt.)

NP-Complete Problems

Question

Are there any problems that are **NP-Complete**?

Answer

Yes! Many, many problems are **NP-Complete**.

Cook-Levin Theorem

Theorem

SAT *is* **NP-Complete**.

Cook-Levin Theorem

Theorem

SAT *is* **NP-Complete***.*

Using reductions one can prove that many other problems are **NP-Complete**

Proving that a problem X is NP-Complete

To prove X is **NP-Complete**, show

1. Show X is in **NP**.
    1. certificate/proof of polynomial size in input
    2. polynomial time certifier $C(s, t)$
2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X

Proving that a problem **X** is **NP-Complete**

To prove X is **NP-Complete**, show

1. Show X is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $C(s, t)$
2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X

SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?

Proving that a problem **X** is **NP-Complete**

To prove X is **NP-Complete**, show

1. Show X is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $C(s, t)$
2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X

SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
Transitivity of reductions:

$Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

NP-Completeness via Reductions

1. **SAT** is **NP-Complete**.
2. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
3. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
4. **Clique** is **NP-Complete**
5. **Vertex Cover** is **NP-Complete**
6. **Set Cover** is **NP-Complete**
7. **Hamilton Cycle** is **NP-Complete**
8. **3-Color** is **NP-Complete**

NP-Completeness via Reductions

1. **SAT** is **NP-Complete**.
2. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
3. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
4. **Clique** is **NP-Complete**
5. **Vertex Cover** is **NP-Complete**
6. **Set Cover** is **NP-Complete**
7. **Hamilton Cycle** is **NP-Complete**
8. **3-Color** is **NP-Complete**

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be **NP-Complete**.

A surprisingly frequent phenomenon!