Chapter 21

Reductions and NP

CS 473: Fundamental Algorithms, Fall 2011
November 15, 2011

21.1 Reductions Continued

21.1.1 Polynomial Time Reduction

21.1.1.1 Karp reduction

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

(A) given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y

(B) $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of I_Y) is polynomial in $|I_X|$

(C) Answer to I_X YES *iff* answer to I_Y is YES.

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

Proposition 21.1.1 *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.

21.1.2 A More General Reduction

21.1.2.1 Turing Reduction

Definition 21.1.2 (Turing reduction.) *Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:*

(A) on any given instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"

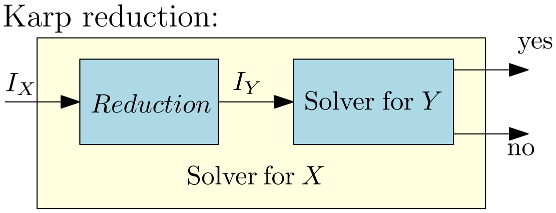
(B) a step is either a standard computation step, or
(C) a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

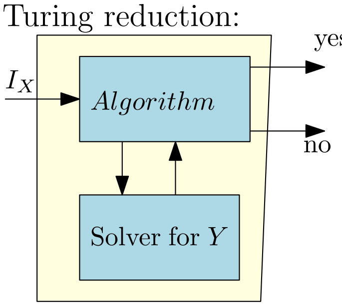
Note: In making sub-routine call to algorithm to solve Y, $\mathcal{A}$ can only ask questions of size polynomial in $|I_X|$. Why?

21.1.2.2 Comparing reductions

(A) Karp reduction:



(B) Turing reduction:



Turing reduction
(A) Algorithm to solve X can call solver for Y many times.
(B) Conceptually, every call to the solver of Y takes constant time.

21.1.2.3 Example of Turing Reduction

Input Collection of arcs on a circle.

Goal Compute the maximum number of non-overlapping arcs.

Reduced to the following problem:?

Input Collection of intervals on the line.

Goal Compute the maximum number of non-overlapping intervals.

How? Used algorithm for interval problem multiple times.

21.1.2.4 Turing vs Karp Reductions

(A) Turing reductions more general than Karp reductions.
(B) Turing reduction useful in obtaining algorithms via reductions.
(C) Karp reduction is simpler and easier to use to prove hardness of problems.
(D) Perhaps surprisingly, Karp reductions, although limited, suffice for most known NP-Completeness proofs.

21.1.3 The Satisfiability Problem (SAT)

21.1.3.1 Propositional Formulas

Definition 21.1.3 *Consider a set of boolean variables $x_1, x_2, \ldots x_n$.*

(A) A **literal** *is either a boolean variable x_i or its negation $\neg x_i$.*

(B) A **clause** *is a disjunction of literals.*
For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.

(C) A **formula in conjunctive normal form (CNF)** *is propositional formula which is a conjunction of clauses*
  (A) $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.

(D) A formula φ is a **3CNF** *:*
A CNF formula such that every clause has ***exactly*** *3 literals.*
  (A) $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_1)$ is a 3CNF formula, but $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is not.

21.1.3.2 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 3CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

21.1.3.3 Satisfiability

SAT

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

Example 21.1.4 $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ *is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true*

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

21.1.3.4 Importance of SAT and 3SAT

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

21.1.4 SAT and 3SAT

21.1.4.1 SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?
In **SAT** clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

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

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
(A) Pad short clauses so they have 3 literals.
(B) Break long clauses into shorter clauses.
(C) Repeat the above till we have a 3CNF.

21.1.4.2 3SAT $\leq_P$ SAT

(A) **3SAT** $\leq_P$ **SAT**.
(B) Because...
A **3SAT** instance is also an instance of **SAT**.

21.1.4.3 SAT $\leq_P$ 3SAT

Claim 21.1.5 **SAT** $\leq_P$ **3SAT**.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that
(A) φ is satisfiable iff φ' is satisfiable
(B) φ' can be constructed from φ in time polynomial in $|\varphi|$.
Idea: if a clause of φ is not of length 3, replace it with several clauses of length exactly 3

21.1.5 SAT $\leq_P$ 3SAT

21.1.5.1 A clause with a single literal

Reduction Ideas

Challenge: Some of the clauses in φ may have less or more than 3 literals. For each clause with < 3 or > 3 literals, we will construct a set of logically equivalent clauses.

(A) *Case clause with one literal:* Let c be a clause with a single literal (i.e., $c = \ell$). Let u, v be new variables. Consider

$$\begin{aligned} c' = \; & \Big(\ell \vee u \vee v\Big) \wedge \Big(\ell \vee u \vee \neg v\Big) \\ & \wedge \Big(\ell \vee \neg u \vee v\Big) \wedge \Big(\ell \vee \neg u \vee \neg v\Big). \end{aligned}$$

Observe that c' is satisfiable iff c is satisfiable

21.1.6 SAT $\leq_P$ 3SAT

21.1.6.1 A clause with two literals

Reduction Ideas: 2 and more literals

(A) *Case clause with 2 literals:* Let $c = \ell_1 \vee \ell_2$. Let u be a new variable. Consider

$$c' = \Big(\ell_1 \vee \ell_2 \vee u\Big) \wedge \Big(\ell_1 \vee \ell_2 \vee \neg u\Big).$$

Again c is satisfiable iff c' is satisfiable

21.1.6.2 Breaking a clause

Lemma 21.1.6 *For any boolean formulas X and Y and z a new boolean variable. Then*

$$X \vee Y \textit{ is satisfiable}$$

if and only if, z can be assigned a value such that

$$\Big(X \vee z\Big) \wedge \Big(Y \vee \neg z\Big) \textit{ is satisfiable}$$

(with the same assignment to the variables appearing in X and Y).

21.1.7 SAT $\leq_P$ 3SAT (contd)

21.1.7.1 Clauses with more than 3 literals

Let $c = \ell_1 \vee \cdots \vee \ell_k$. Let $u_1, \ldots u_{k-3}$ be new variables. Consider

$$\begin{aligned} c' = \; & \Big(\ell_1 \vee \ell_2 \vee u_1\Big) \wedge \Big(\ell_3 \vee \neg u_1 \vee u_2\Big) \\ & \wedge \Big(\ell_4 \vee \neg u_2 \vee u_3\Big) \wedge \\ & \cdots \wedge \Big(\ell_{k-2} \vee \neg u_{k-4} \vee u_{k-3}\Big) \wedge \Big(\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}\Big). \end{aligned}$$

Claim 21.1.7 *c is satisfiable iff c' is satisfiable.*

Another way to see it — reduce size of clause by one:

$$c' = \Big(\ell_1 \vee \ell_2 \ldots \vee \ell_{k-2} \vee u_{k-3}\Big) \wedge \Big(\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}\Big).$$

21.1.7.2 An Example

Example 21.1.8

$$\varphi = \Big(\neg x_1 \vee \neg x_4\Big) \wedge \Big(x_1 \vee \neg x_2 \vee \neg x_3\Big) \\ \wedge \Big(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\Big) \wedge \Big(x_1\Big).$$

Equivalent form:

$$\begin{aligned}\psi = {} & (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z) \\ & \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1) \\ & \wedge (x_1 \vee u \vee v) \wedge (x_1 \vee u \vee \neg v) \\ & \wedge (x_1 \vee \neg u \vee v) \wedge (x_1 \vee \neg u \vee \neg v).\end{aligned}$$

21.1.8 Overall Reduction Algorithm

21.1.8.1 Reduction from SAT to 3SAT

```
ReduceSATTo3SAT(φ):
    // φ: CNF formula.
    for each clause c of φ do
        if c does not have exactly 3 literals then
            construct c' as before
        else
            c' = c
    ψ is conjunction of all c' constructed in loop
    return Solver3SAT(ψ)
```

Correctness (informal)
φ is satisfiable iff ψ is satisfiable because for each clause c, the new 3CNF formula c' is logically equivalent to c.

21.1.8.2 What about 2SAT?

2SAT can be solved in polynomial time! (In fact, 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 2CNF clauses. Introduce a face variable α, and rewrite this as

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

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

21.1.8.3 What about 2SAT?

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

(Hint: Create a graph with two vertices for each variable (for a variable x there would be two vertices with labels $x = 0$ and $x = 1$). For ever 2CNF clause add two directed edges in the graph. The edges are implication edges: They state that if you decide to assign a certain value to a variable, then you must assign a certain value to some other variable.

Now compute the strong connected components in this graph, and continue from there...)

21.1.9 3SAT and Independent Set

21.1.9.1 Independent Set

Problem: Independent Set

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

21.1.9.2 3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a 3CNF 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 3CNF formulas – reduction would not work for other kinds of boolean formulas.

21.1.9.3 Interpreting 3SAT

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

(A) 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.

(B) 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.

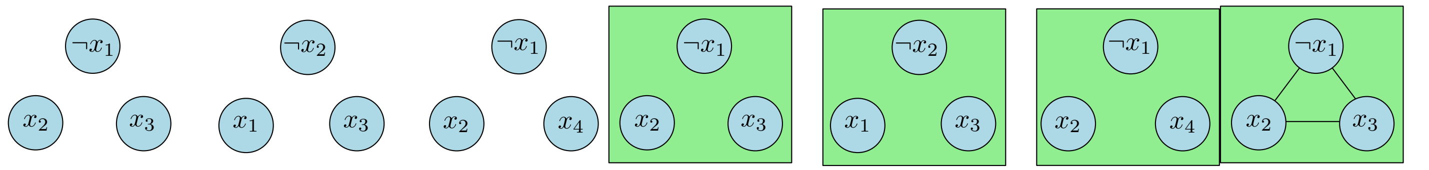


Figure 21.1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

21.1.9.4 The Reduction

(A) G_φ will have one vertex for each literal in a clause
(B) 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
(C) Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
(D) Take k to be the number of clauses

21.1.9.5 Correctness

Proposition 21.1.9 *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
 (A) Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size

■

21.1.9.6 Correctness (contd)

Proposition 21.1.10 *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
 (A) S must contain exactly one vertex from each clause
 (B) S cannot contain vertices labeled by conflicting clauses
 (C) 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

■

21.1.9.7 Transitivity of Reductions

Lemma 21.1.11 *$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.*

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

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

21.2 Definition of NP

21.2.0.8 Recap ...

Problems

(A) **Independent Set**
(B) **Vertex Cover**
(C) **Set Cover**
(D) **SAT**
(E) **3SAT**

Relationship

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

21.3 Preliminaries

21.3.1 Problems and Algorithms

21.3.1.1 Problems and Algorithms: Formal Approach

Decision Problems

(A) *Problem Instance:* Binary string s, with size $|s|$
(B) *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 21.3.1 *(A) A is an algorithm for problem X if $A(s) =$ "yes" iff $s \in X$*
(B) 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

21.3.1.2 Polynomial Time

Definition 21.3.2 *Polynomial time (denoted P) is the class of all (decision) problems that have an algorithm that solves it in polynomial time*

Example 21.3.3 *¡2-¿ Problems in P include*
(A) Is there a shortest path from s to t of length $\leq k$ in G?
(B) Is there a flow of value $\geq k$ in network G?
(C) Is there an assignment to variables to satisfy given linear constraints?

21.3.1.3 Efficiency Hypothesis

A problem X has an efficient algorithm iff $X \in P$, that is X has a polynomial time algorithm.

Justifications:

(A) Robustness of definition to variations in machines.
(B) A sound theoretical definition.
(C) Most known polynomial time algorithms for "natural" problems have small polynomial running times.

21.3.1.4 Problems with no known polynomial time algorithms

Problems

(A) **Independent Set**
(B) **Vertex Cover**
(C) **Set Cover**
(D) **SAT**
(E) **3SAT**

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

Question: What is common to above problems?

21.3.1.5 Efficient Checkability

Above problems share the following feature:

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:

(A) **SAT** formula φ: proof is a satisfying assignment
(B) **Independent Set** in graph G and k: a subset S of vertices

21.3.2 Certifiers/Verifiers

21.3.2.1 Certifiers

Definition 21.3.4 *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

Efficient Certifier
C is an *efficient certifier* for problem X if there is a polynomial $p(\cdot)$ such that for every string s, $s \in X$ iff there is a string t with $|t| \leq p(|s|)$, $C(s,t) =$ "yes" and C runs in polynomial time

21.3.2.2 Example: Independent Set

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

21.3.3 Examples

21.3.3.1 Example: Vertex Cover

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

21.3.3.2 Example: SAT

(A) *Problem:* Does formula φ have a satisfying truth assignment?
  (A) *Certificate:* Assignment a of 0/1 values to each variable
  (B) *Certifier:* Check each clause under a and say "yes" if all clauses are true

21.3.3.3 Example:Composites

(A) *Problem:* Is number s a composite?
  (A) *Certificate:* A factor $t \leq s$ such that $t \neq 1$ and $t \neq s$
  (B) *Certifier:* Check that t divides s (Euclid's algorithm)

21.4 NP

21.4.1 Definition

21.4.1.1 Nondeterministic Polynomial Time

Definition 21.4.1 *Nondeterministic Polynomial Time (denoted by NP) is the class of all problems that have efficient certifiers*

Example 21.4.2 *¡2-¿* **Independent Set**, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, *Composites are all examples of problems in NP*

21.4.1.2 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.

21.4.2 Intractability

21.4.2.1 P versus NP

Proposition 21.4.3 $P \subseteq NP$

For a problem in P no need for a certificate!

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

(A) Certifier C on input s, t, runs $A(s)$ and returns the answer
(B) C runs in polynomial time
(C) If $s \in X$ then for every t, $C(s,t) =$ "yes"
(D) If $s \notin X$ then for every t, $C(s,t) =$ "no"

■

21.4.2.2 Exponential Time

Definition 21.4.4 *Exponential Time (denoted EXP) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\text{poly}(|s|)})$*

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

21.4.2.3 NP versus EXP

Proposition 21.4.5 $NP \subseteq EXP$

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

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

■

21.4.2.4 Examples

(A) **SAT**: try all possible truth assignment to variables.
(B) **Independent Set**: try all possible subsets of vertices.
(C) **Vertex Cover**: try all possible subsets of vertices.

21.4.2.5 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 $P = NP$?

21.4.3 If $P = NP \ldots$

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

(A) Many important optimization problems can be solved efficiently.
(B) The RSA cryptosystem can be broken.
(C) No security on the web.
(D) No e-commerce ...
(E) Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

21.4.3.2 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!