NP-Complete Problems, Part I

Jim Royer

April 1, 2019

Uncredited diagrams are from DPV or homemade.

Efficient and Not-So-Efficient Algorithms

Problem spaces tend to be big:

- A graph on n vertices can have up to n^{n-2} spanning trees.
- A graph on n vertices can have $\Theta(2^n)$ many paths between verts s and t.
- *Etc.*

The Good News

In our previous work, out of problems with $\Theta(2^n)$ (or worse) many choices, we have found the right answer in time $O(n^k)$ for some k.

The Bad News

Not all problems are so nice.

Search Problems

Search problems are those of the form:

Given: ...
Find: ... *(usually within a large search space)*

Satisfiability (as a search problem)

Given: A boolean formula in conjunctive normal form (CNF).
Find: A satisfying assignment for it (if it has one).

We need to define some terms.

Propositional Logic

- The *formulas* of propositional logic are given by the grammar:

$$\begin{aligned} P &::= \mathit{Var} \mid \neg P \mid P \wedge P \mid P \vee P \\ \mathit{Var} &::= \text{the syntactic category of variables} \end{aligned}$$

- A *truth assignment* is a function $\mathcal{I}$: Variables $\to \{\,\mathsf{False}, \mathsf{True}\,\}$.
- $\mathcal{I}$, a truth assignment, determines the value of a formula by:

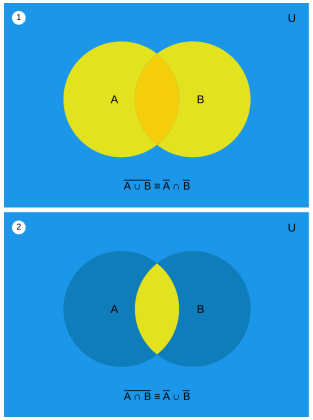
$$\begin{aligned} \mathcal{I}[\![x]\!] = \mathsf{True} &\ \textit{iff}\ \mathcal{I}(x) = \mathsf{True} \quad (x \text{ a variable}) \\ \mathcal{I}[\![\neg p]\!] = \mathsf{True} &\ \textit{iff}\ \mathcal{I}[\![p]\!] = \mathsf{False} \\ \mathcal{I}[\![p \wedge q]\!] = \mathsf{True} &\ \textit{iff}\ \mathcal{I}[\![p]\!] = \mathcal{I}[\![q]\!] = \mathsf{True}. \\ \mathcal{I}[\![p \vee q]\!] = \mathsf{True} &\ \textit{iff}\ \mathcal{I}[\![p]\!] = \mathsf{True} \quad \text{or} \quad \mathcal{I}[\![q]\!] = \mathsf{True}. \end{aligned}$$

- A *satisfying assignment* for a formula p is an $\mathcal{I}$ with $\mathcal{I}[\![p]\!] = \mathsf{True}$.
- The only known algorithms for SAT run in exponential time (in the worst case).

Digression: **DeMorgan's Laws**

$$\neg(P \vee Q) \iff (\neg P) \wedge (\neg Q)$$

$$\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$$

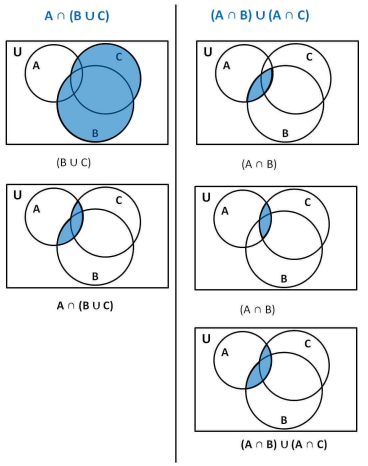


Digression: **Distributive Laws**

$$A \wedge (B \vee C) \iff (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) \iff (A \vee B) \wedge (A \vee C)^{\star}$$

* Exercise for the reader.



Conjunctive Normal Form

- Instead of writing $\neg x$ we write $\overline{x}$.
- A variable x and the negation of a variable $\overline{x}$ are called *literals*.
- A *clause* is a disjunction of literals.
  E.g.: $(x \vee \overline{y} \vee z)$.
- A *conjunctive normal form formula* is a conjunction of clauses.
  E.g.: $(x \vee y \vee z) \wedge (x \vee \overline{y}) \wedge (y \vee \overline{z}) \wedge (z \vee \overline{x}) \wedge (\overline{x} \vee \overline{y} \vee \overline{z})$

Satisfiability (as a search problem)

Given: A boolean formula in conjunctive normal form (CNF).
Find: A satisfying assignment for it (if it has one).

- Note the differences with the boolean circuit evaluation problem.
- If a CNF formula has n variables, there are 2^n possible assignments.

Digression: **Translating Boolean Formulas to CNF**

- A formula is in *negation normal form* (NNF) iff the only place a negation symbol appears in F is in front of a variable.

Step 1

Given a formula F translate it to an equivalent NNF formula using DeMorgan's Laws.

Step 2

Given a NNF formula F translate it to an equivalent CNF formula using the distributive law $A \vee (B \wedge C) \iff (A \vee B) \wedge (A \vee C)$.

Back to Search Problems, 1

Elements of a Search Problem

- I: an instance of the problem
- S: a possible solution for I
- C : (Instances)$\times$(Potential Solutions) $\to$ { *True*, *False* }
  $C(I, S)$ = the result of checking if S solves I
- An *efficient checking algorithm* for C is an algorithm for C that, on input (I, S) runs in $O(|I|^k)$-time for some k. *(Implies that $|S|$ cannot be too large.)*

For SAT:

An instance	:	a CNF formula	$(x \vee \bar{y}) \wedge (y \vee \bar{x})$
A potential solution	:	a truth assignment	$x \mapsto True, y \mapsto True$
efficient checker	:	boolean circuit evaluation	

Back to Search Problems, 2

Elements of a Search Problem

- I: an instance of the problem
- S: a possible solution for I
- C : (Instances)$\times$(Pot. Solutions) $\to$ { *True*, *False* }
  $C(I, S)$ = the result of checking if S solves I
- An *efficient checking algorithm* for C is an algorithm for C that, on input (I, S) runs in $O(|I|^k)$-time for some k. *(Implies that $|S|$ cannot be too large.)*

Variations of SAT

Horn Clause SAT	Easy (Chapter 5)
2SAT	Easy
3SAT	Hard, it seems

Definition

nSAT is the restricted version of SAT in which each clause as at most n literals.

Traveling Salesman

Traveling Salesman (TS) as a search problem

Given: n vertices and all $n \cdot (n-1)/2$-many distances between them, b a budget (number)
Find: An ordering of $1, \ldots, n$: $\pi(1), \pi(2), \ldots, \pi(n)$ (a tour) so that

$$d_{\pi(1),\pi(2)} + d_{\pi(2),\pi(3)} + \cdots + d_{\pi(n),\pi(1)} \leq b.$$

Traveling Salesman (TS) as an optimization problem

Given: n vertices and all $n \cdot (n-1)/2$-many distances between them.
Find: An ordering of $1, \ldots, n$: $\pi(1), \pi(2), \ldots, \pi(n)$ so that the tour's cost $d_{\pi(1),\pi(2)} + d_{\pi(2),\pi(3)} + \cdots + d_{\pi(n),\pi(1)}$ is minimal.

The two problems are equivalent:

- A solution to the optimization problem, solves the search problem. *(Why?)*
- Given a way to solve the search problem, you can construct a solution to the opt. problem via binary search. *(How?)*

- The only known algorithms for these problems are exponential time.
- TS is a restriction of the Minimal Spanning Tree problem in which the MST is allowed no branches.

Euler and Hamiltonian Paths, 1

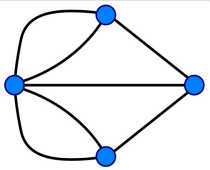
Definition

A path in an undirected graph is an *Euler path* when it uses each *edge* of the graph exactly once. *(The path may pass through a vertex many times.)*

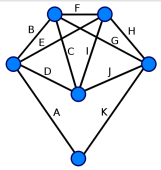
If the path is a cycle, then it is called an *Euler Tour* or an *Euler Circuit*.

Theorem (Euler)

G has an Euler path $\iff$ *G is connected and has at most two vertices of odd degree.*



No Euler Path



Euler Tour: $A\ B \ldots K$

Euler and Hamiltonian Paths, 2

Definition

A path in an undirected graph is a *Rudrata path* (or more usually a *Hamiltonian path*) when it uses each *vertex* of the graph exactly once.

If the path is a cycle, then it is called a *Rudrata Cycle* or *Hamiltonian Cycle*.

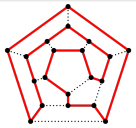


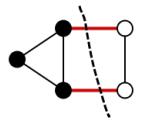
Image from: `http://en.wikipedia.org/wiki/Hamiltonian_path`

- There is a nice poly-time algorithm for the Euler Path Search problem. (See `http://en.wikipedia.org/wiki/Eulerian_path`.)
- All known algorithms for the Hamiltonian Path Search Problem are exponential-time.

Cuts and bisections

Definition

A *cut* in a graph is a set of edges which, if removed, disconnect the graph.
A *minimum cut* is a cut of smallest size.



Minimum Cut Problem

Given: An undirected graph G and a budget b (a number),
Find: A cut of G of at most b edges.

Balanced Cut Problem

Given: An undirected graph $G = (V, E)$ and a budget b (a number),
Find: A partition of V: S and T with $|S|, |T| \geq |V|/3$ such that the number of edges between S and T is at most b.

- You can use Ford-Fulkerson to solve Min-Cut in poly-time.
- The only known algorithms for Balanced-Cut are exponential time.
- Balanced-Cuts are important in clustering. (See DPV.)

Integer Linear Programming

Integer Linear Programming (ILP)

Given: constraints $A\vec{x} \leq \vec{b}$ and objective function $\vec{c}^T \cdot \vec{x}$ and goal: g
Find: A vector of integers $\vec{x}$ satisfying $A\vec{x} \leq \vec{b}$ and $\vec{c}^T \cdot \vec{x} \geq g$.

— or equivalently —

Given: constraints $A\vec{x} \leq \vec{b}$
Find: A vector of integers $\vec{x}$ satisfying $A'\vec{x} \leq \vec{b}'$.
($\vec{c}^T \cdot \vec{x} \geq g$ is incorporated into the constraints.)

Zero-One Equations (ZOE)

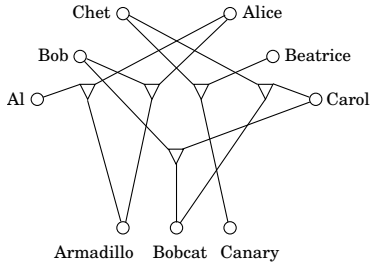
Given: constraints $A\vec{x} \leq \vec{b}$
Find: A vector of 0's and 1's $\vec{x}$ satisfying $A'\vec{x} \leq \vec{b}'$.

- ILP and ZOE show up in lots of optimization work.
- The only known algorithms for ILP and ZOE are exponential time.

Three-dimensional matching

3D Matching

Given: $R \subseteq A \times B \times C$ where $|A| = |B| = |C| = n$.
Find: A subset $M \subseteq R$ of n many triples such that
if (a, b, c) and (a', b', c') are distinct elements of M,
then $a \neq a'$, $b \neq b'$, and $c \neq c'$.



- 2D matching is poly-time (via Ford-Fulkerson).
- The only known algorithms for 3D Matching are exponential time.

Independent set, vertex cover, and clique, 1

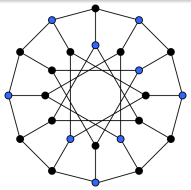
Definition

Suppose $G = (V, E)$ is an undirected graph and $U \subseteq V$.

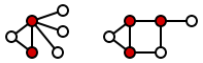
(a) U is *independent* when for each $u, v \in U$, $(u, v) \notin E$.

(b) U is a *vertex cover* when each edge of E has at least one endpoint in U.

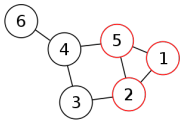
(c) U is a *clique* when for each distinct $u, v \in U$, $(u, v) \in E$.



The blue vertices are a max-sized independent set for the graph.



The red vertices are a min-sized vertex cover for the graph.



The red vertices are a max-sized clique for the graph.

Independent set, vertex cover, and clique, 2

Definition

Suppose $G = (V, E)$ is an undirected graph and $U \subseteq V$.

(a) U is *independent* when for each $u, v \in U$, $(u, v) \notin E$.

(b) U is a *vertex cover* when each edge of E has at least one endpoint in U.

(c) U is a *clique* when for each distinct $u, v \in U$, $(u, v) \in E$.

Independent Set Problem

Given: G and b.
Find: An independent set for G of size $\geq b$.

Vertex Cover Problem

Given: G and b.
Find: A vertex cover for G of size $\leq b$.

Clique Problem

Given: G and b.
Find: A clique in G of size $\geq b$.

- The only known algorithms for these problems are exponential time.

Longest Path, Knapsack, Subset Sum

The Longest Path Problem

Given: A undirected graph G.
Find: A longest simple path in G. *(Simple path $\equiv$ a path with no repeated vertices.)*

Knapsack

Given: Weights $w_1, \ldots, w_n$, values $v_1, \ldots, v_n$, Total Capacity: W, and Goal: G (all positive integers).
Find: A selection of items with total weight $\leq W$ and total value $\geq G$.

Subset Sum

Given: A multiset of integers M and goal G.
Find: An $\{x_1, \ldots, x_k\} \subseteq M$ such that $G = x_1 + \cdots + x_k$.

- The only known algorithms for these problems are exponential time.
  But didn't we have an LP solution to Knapsack? Yes, but ...

Problems: Hard and Easy

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

- All of the hard problems above are hard for the same reason.
- In fact, they are all "the same problem" in disguise.

P and NP, 1

Recall: Elements of a Search Problem

- I: an instance of the problem and S: a possible solution for I
- C : (Instances)$\times$(Pot. Solutions) $\to$ { *True*, *False* }; $C(I,S) = \begin{cases} True, & \text{if } S \text{ solves } I; \\ False, & \text{otherwise.} \end{cases}$

Definition (NP and P: As search problems)

(a) An *efficient checking algorithm* is an algorithm that computes C as above in $O(|I|^k)$-time for some k. *(This implies that $|S|$ cannot be too large.)*

(b) Each C : (Instances)$\times$(Pot. Solutions) $\to$ { *True*, *False* } that is computable in $O(|I|^{O(1)})$ time determines an (artificial) search problem: [S is a solution for I] $\iff$ [$C(I,S) = True$].

(c) *NP* = the class of all search problems that have eff. checking algorithms.

(d) *P* = the class of all search problems for which one can ***find*** solutions (or determine there are none) in polynomial time.

P and NP, 2

Definition

A *decision problem* is a problem with an Yes/No answer.

SAT as a decision problem: Does CNF formula F have satisfying assignment?
Hamiltonian Path as a decision problem: Does G have a Hamiltonian Path?

Definition (NP and P: As decision problems)

(a) *NP* = the class of all decision problems that have eff. checking algorithms.

(b) *P* = the class of all decision problems that have polytime algorithms.

In Algorithms: search problems are a bit more natural than decision problems.
In Computational Complexity Theory: the reverse
Which is better? The search & decision forms of NP problems are roughly of equivalent hardness.

The $1,000,000,000 Question: $P \stackrel{?}{=} NP$.
Collect your prize here: `http://www.claymath.org/millennium/`

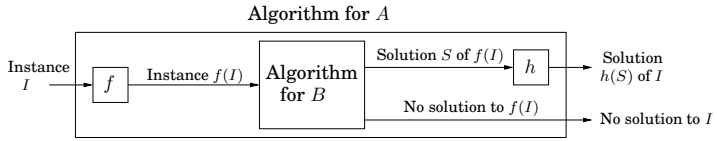
Formalizing Reductions, 1

Q: What is the basis for believing all those problems we just listed are hard?
Reducing problem A to problem B $\approx$ rephrasing A in B's language
E.g., The rephrasing the *Boolean Circuit Eval Problem* as an LP problem.

Definition

A problem A is *reducible* to problem B (written: $A \to B$ [DPV] or $A \trianglelefteq B$ [these slides]) when there are two polytime computable functions f and h:

- f transforms an A-instance I into a B-instance $f(I)$.
- h transforms a B-solution S of $f(I)$ for into an A-solution $h(S)$ of I.



Formalizing Reductions, 2

Definition

(a) A problem is *NP-hard* when ***all*** NP-problems reduce to it.

(b) A problem is *NP-complete* when it is in NP and NP-hard.

Suppose $A \trianglelefteq B$. Then:

- B is easy $\implies$ A is also easy. *(Why?)*
- A is hard $\implies$ B is also hard. *(Why?)*

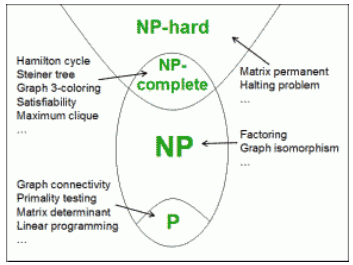
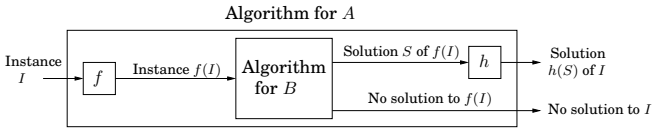


Image from `http://www.solipsys.co.uk/new/PVsNP.html`



Formalizing Reductions, 2

Definition

(a) A problem is *NP-hard* when ***all*** NP-problems reduce to it.

(b) A problem is *NP-complete* when it is in NP and NP-hard.

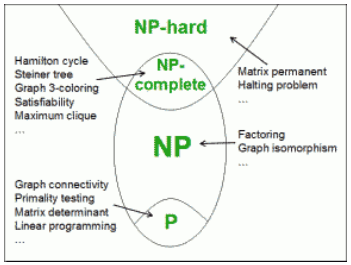
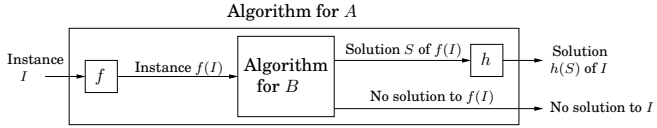


Image from `http://www.solipsys.co.uk/new/PVsNP.html`

Suppose $A \trianglelefteq B$. Then:

- B is easy $\implies$ A is also easy. *(Why?)*
- A is hard $\implies$ B is also hard. *(Why?)*

$(P \implies Q) \iff (\neg Q \implies \neg P)$

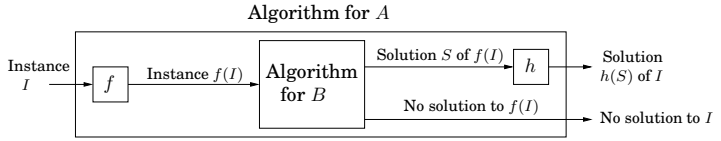


Formalizing Reductions, 3

- Reductions compose
  I.e., $A \trianglelefteq B$ and $B \trianglelefteq C$ implies that $A \trianglelefteq C$.

$\therefore$ If $A \trianglelefteq B$ and A is NP-hard, then so it B.

$\therefore$ If A is NP-complete and B is an NP-problem with $A \trianglelefteq B$, then B is also NP-complete. *(Why is this handy?)*



Formalizing Reductions, 4

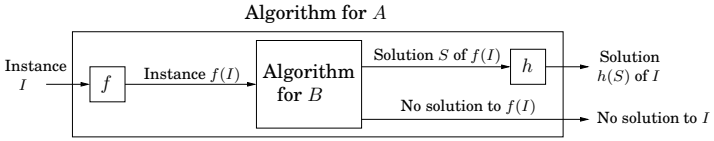
Steps in proving that a reduction works

Argue that:

1. f and h are polytime computable. *(Some hand waving on this is usually OK.)*
2. If $f(I)$ has solution S, then I has solution $h(S)$.
3. If $f(I)$ has no solution, then I has no solution.

3′. If I has a solution, then $f(I)$ also has solution.

- 3 and 3′ are equivalent: $(P \implies Q) \iff (\neg Q \implies \neg P)$.
- However, 3′ is usually easier to prove.



The Plan of §8.3