Chapter 3: Fundamentals of computational complexity

Goal: Evaluate the computational requirements (we focus on time) to solve computational problems.

Two major types of issues:

- Evaluate the complexity of a given algorithm A to solve a given problem P.
- Evaluate the inherent difficulty of a given problem P.

Focus here on discrete optimization problems.

3.1 Algorithm complexity (recap)

Goal: Estimate the performance of alternative algorithms for a given problem so as to select the most appropriate one for the instances of interest.

Definition: An *instance* I of a problem P is a special case of P.

Example

Problem P : order m integer numbers $c_1, \ldots, c_m$

Instance I : $m = 3, c_1 = 2, c_2 = 7, c_3 = 5$

The computing time of an algorithm is evaluated in terms of

the number of *elementary operations* (arithmetic operations, comparisons, memory accesses,...) needed to solve a given instance *I*.

Assumption: all elementary operations require one unit of time.

Clearly, the number of elementary operations depends on the size of the instance.

Size of an instance

Definition: The *size* of an instance I, denoted by $|I|$, is the number of bits needed to encode (describe) I.

Example

An instance is specified by values: m and $c_1, \ldots, c_m$

Since $\lceil \log_2(i) \rceil$ bits are needed to encode a positive integer i

$$|I| \leq \lceil \log_2 m \rceil + m \cdot \lceil \log_2 c_{\max} \rceil \qquad \text{where } c_{\max} = \max\{c_j : 1 \leq j \leq m\}$$

For the previous instance $m = 3,\ c_1 = 2,\ c_2 = 7,\ c_3 = 5$

$$\Rightarrow\ |I| \leq \lceil \log_2 3 \rceil + 3 \cdot \lceil \log_2 7 \rceil$$

Time complexity

We look for a function $f(n)$ such that, for every instance I of size at most n ($\forall\ I$ with $|I| \leq n$)

the number of elementary operations to solve instance $I \ \leq\ f(n)$.

Observations:

- Since $f(n)$ is an upper bound $\forall\ I$ with $|I| \leq n$, we consider the worst case behaviour.
- $f(n)$ is expressed in asymptotic terms – O(…) notation.

Example An $O(m \log m)$ algorithm is available to sort m integer numbers (e.g., quicksort).

Definition: An algorithm is *polynomial* if it requires, in the worst case, a number of elementary operations

$f(n) = O(n^d)$, where d is a constant and $n = |I|$ is the size of the instance.

We distinguish between algorithms whose order of complexity (in the worst case) is

$O(n^d)$ polynomial

$O(2^n)$ exponential

Polynomial algorithms with, for instance, d≥6 are not efficient in practice!

Examples

- Dijkstra's algorithm for shortest path problem

  Size of the instance: $|I| = O(m \log_2 n + m \log_2 c_{\max})$

  Time complexity: $O(n^2)$ where n is the number of nodes

  $\Rightarrow$ polynomial w.r.t $|I|$ $(|I| \geq m \geq n - 1)$.

- Basic version of Ford-Fulkerson's algorithm for maximum flow problem

  Size of the instance: $|I| = O(m \log_2 n + m \log_2 k_{\max})$

  Time complexity: $O(m^2 k_{\max})$ where m is the number of arcs

  $\Rightarrow$ not polynomial with respect to $|I|$.

3.2 Inherent problem complexity

Goal: Evaluate the inherent difficulty of a given computational problem so as to adopt an appropriate solution approach.

Intuitively, we look for the complexity of "the most efficient algorithm that could ever be designed" for that problem.

Definition: A problem P is *polynomially solvable* ("*easy*") if there is a polynomial-time algorithm providing an (optimal) solution for every instance.

Examples: min spanning trees, shortest paths, max flows,...

? Do “difficult” problems (which cannot be solved in polynomial time) actually exist?

For many (discrete) optimization problems, the best algorithm known today requires a number of elementary operations which grows, in the worst case, exponentially in the size of the instance.

Observation: This does not prove that they are “difficult”!

Example

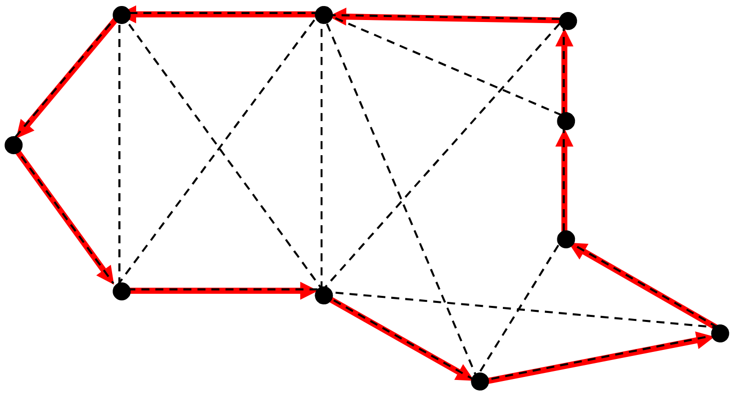
Given an integer number, determine whether it is prime.

Thought to be difficult for a long time, until Agrawal-Kayal-Saxena found a polynomial-time algorithm in 2002.

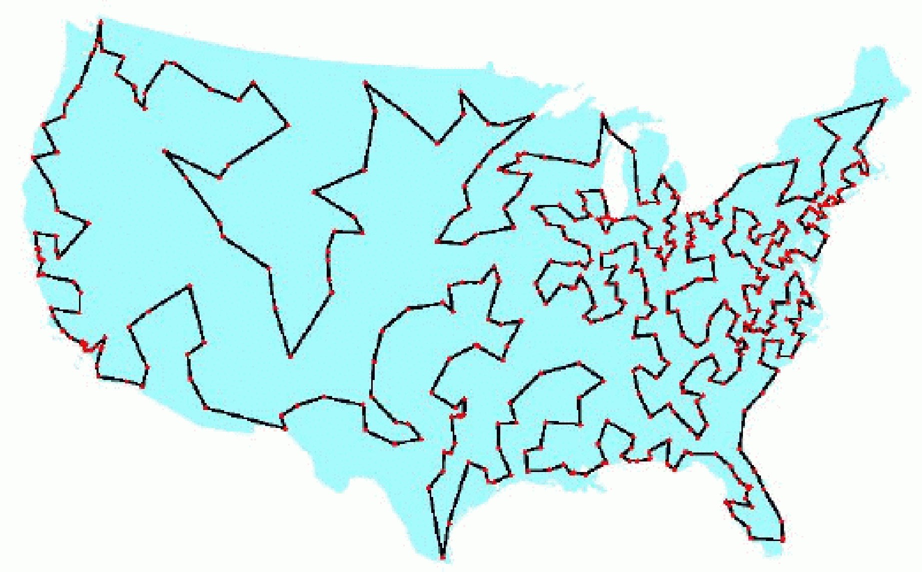
Traveling salesman problem (TSP)

Problem

Given a directed $G = (N, A)$ with a cost $c_{ij} \in \mathbf{Z}$ for each $(i, j) \in A$, determine a <u>circuit</u> of minimum total cost visiting each node <u>exactly once</u>.



arc cost (e.g., distance, travel time)



Definition: A *Hamiltonian circuit* C is a circuit that visits each node exactly once.

Denoting by H the set of all Hamiltonian circuits of G, the problem amounts to

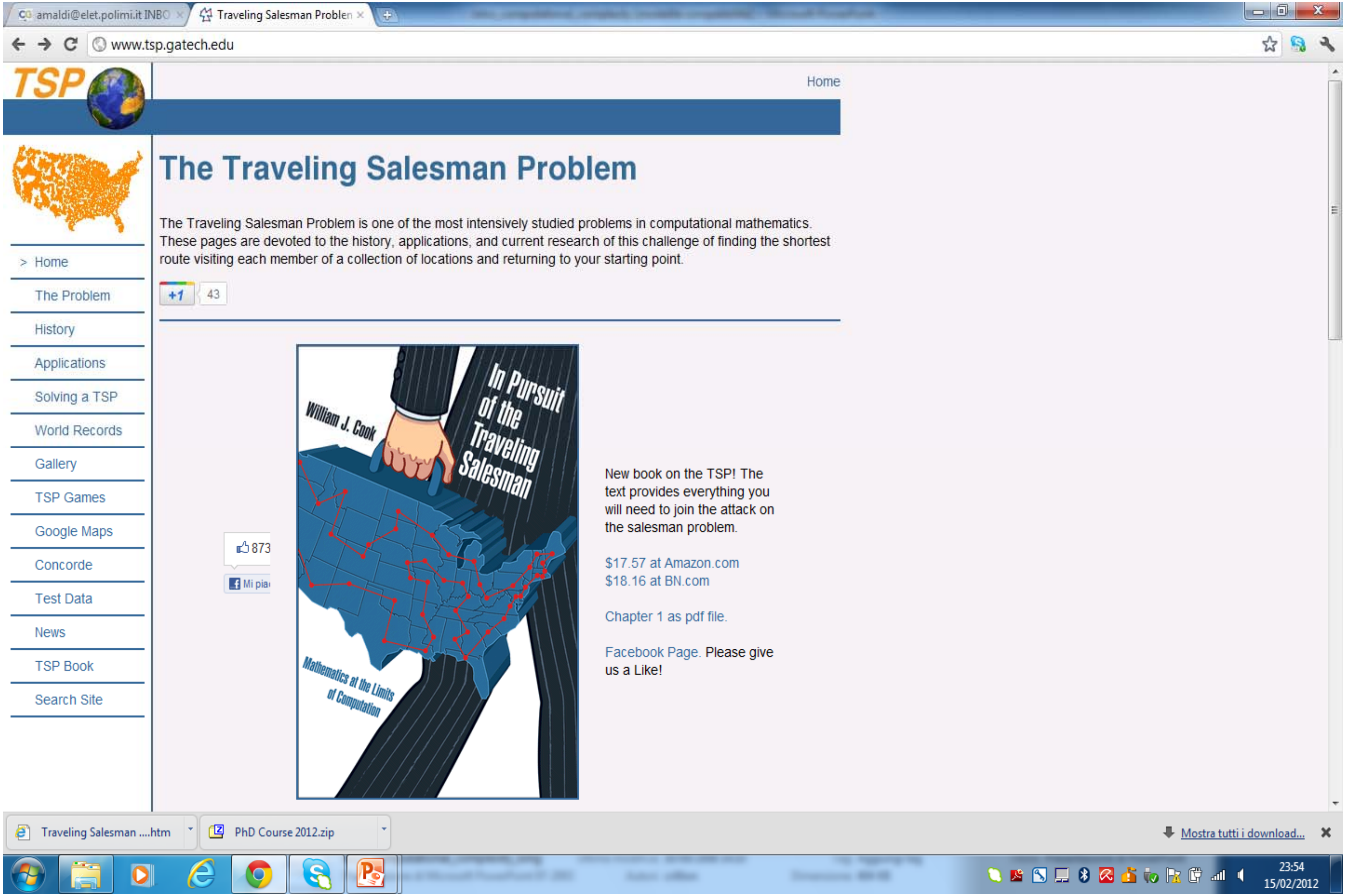
$$\min_{C \in H} \sum_{(i,j) \in C} c_{ij}$$

Observation: H contains a finite number of elements:

$$|H| \leq (n-1)!$$

Applications: logistics, scheduling, VLSI design,…

Many variants and extensions (Vehicle Routing Problem --VRP)



The Traveling Salesman Problem

The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics. These pages are devoted to the history, applications, and current research of this challenge of finding the shortest route visiting each member of a collection of locations and returning to your starting point.

New book on the TSP! The text provides everything you will need to join the attack on the salesman problem.

$17.57 at Amazon.com
$18.16 at BN.com

Chapter 1 as pdf file.

Facebook Page. Please give us a Like!

http://www.math.uwaterloo.ca/tsp/

3.3 Basics of $\mathcal{NP}$-completeness theory

We consider recognition problems rather than optimization problems.

Definition: A *recognition problem* is a problem whose solution is either "yes" or "no".

To each optimization problem we can associate a recognition version.

Example TSP-r

Given a directed $G = (N, A)$ with integer costs c_{ij} and an integer L, does there exist a Hamiltonian circuit of total cost $\leq L$?

Recognition problems

Any optimization problem is at least as difficult as (not easier than) the recognition version.

Example

If we knew how to solve TSP (determine a Hamiltonian circuit of minimum total cost), we could obviously solve TSP-r (decide whether ∃ a Hamiltonian circuit of total cost $\leq L$).

If the recognition version is "difficult",

then the optimization problem is also "difficult".

Complexity class $\mathcal{P}$

Definition: $\mathcal{P}$ denotes the class of all recognition problems that can be solved in polynomial time.

For each recognition problem in $\mathcal{P}$, there exists an algorithm providing, for every instance I, the answer "yes" or "no" in polynomial time in $|I|$.

Example: recognition versions of optimal spanning trees, shortest paths, maximum flows.

Observation: $\mathcal{P}$ can be formally defined in terms of polynomial time (deterministic) Turing machines.

A. Turing 1912-54

Complexity class $\mathcal{NP}$

Definition: $\mathcal{NP}$ denotes the class of all recognition problems such that, for each instance with "*yes*" answer, there exists a concise certificate (proof) which allows to verify in polynomial time that the answer is "*yes*".

Example

TSP-r $\in \mathcal{NP}$

Indeed, one can verify in polynomial time if a given sequence of nodes corresponds to a Hamiltonian circuit and if its total cost $\leq L$.

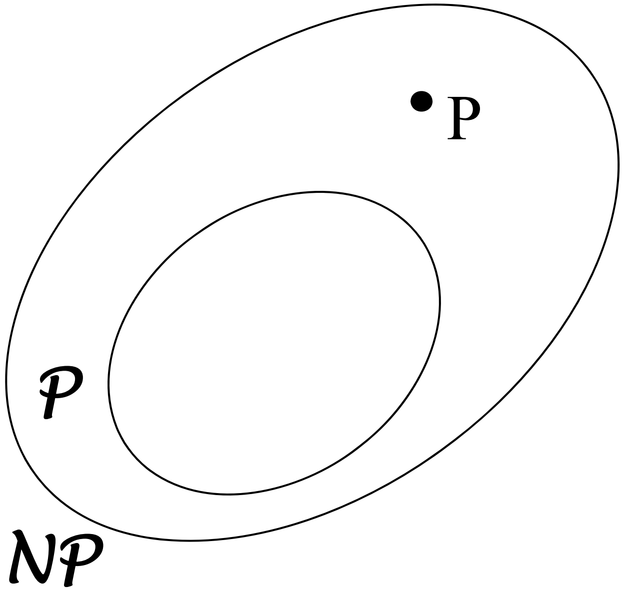
Formal definition:

> $\mathcal{NP}$ denotes the class of all recognition problems for which $\exists$ a polynomial $p(n)$ and a certificate-checking algorithm $\mathcal{A}_{cc}$ such that :
>
> I is a "yes"-instance $\Leftrightarrow$ $\exists$ a certificate $\gamma(I)$ of polynomial size ($|\gamma(I)| \leq p(|I|)$) and $\mathcal{A}_{cc}$ applied to the input "$I,\gamma(I)$" reaches the answer "yes" in at most $p(|I|)$ steps.

Observation: We do not consider how difficult it is to find the certificate (it could be provided by an "oracle")! It suffices that it exists and it allows to verify the "yes" answer in polynomial time.

Relationship between $\mathcal{P}$ and $\mathcal{NP}$

Clearly $\mathcal{P} \subseteq \mathcal{NP}$



Conjecture $\mathcal{P} \subset \mathcal{NP}$

One of the "Millennium Prize Problems" 2000!

$\mathcal{NP}$ does not stand for "Not Polynomial" algorithm but for "Non-deterministic Polynomial" Turing machines.

Polynomial time reductions

Concept needed to classify recognition problems according to their intrinsic complexity and to identify the most difficult ones in $\mathcal{NP}$.

Definition:

Let P_1 and $P_2 \in \mathcal{NP}$, then P_1 *reduces in polynomial time* to P_2

$(P_1 \propto P_2)$ if there exists an algorithm to solve P_1 which

- uses (once or several times) a hypothetical algorithm for P_2 as a subroutine,
- the algorithm for P_1 runs in polynomial time if we assume that the algorithm for P_2 runs in constant time (i.e. is $O(1)$).

Definition:

A reduction is a *polynomial time transformation* ($P_1 \propto_t P_2$) if the algorithm that solves P_2 is called only once.

Example

Undir-TSP-r: Given undirected graph $G = (N, E)$ with arc costs and an integer L, $\exists$ a Hamiltonian cycle of total cost $\leq L$?

TSP-r: Given a directed graph $G' = (N', A')$ with arc costs and an integer L', $\exists$ a Hamiltonian circuit of total cost $\leq L'$?

Undir-TSP-r $\propto_t$ TSP-r

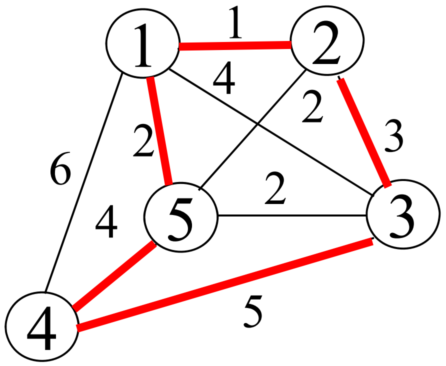
Show that Undir-TSP-r $\propto_t$ TSP-r :

in polynomial time

$\forall I_1 \in P_1$ it is easy to construct a particular $I_2 \in P_2$

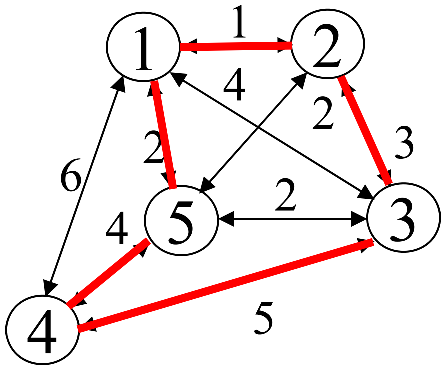
$G=(N,E)$

$L = 15$



$G'=(N',A')$

$L' = 15$



such that I_1 has a “yes” answer $\Leftrightarrow$ I_2 has a “yes” answer.

Consequence:

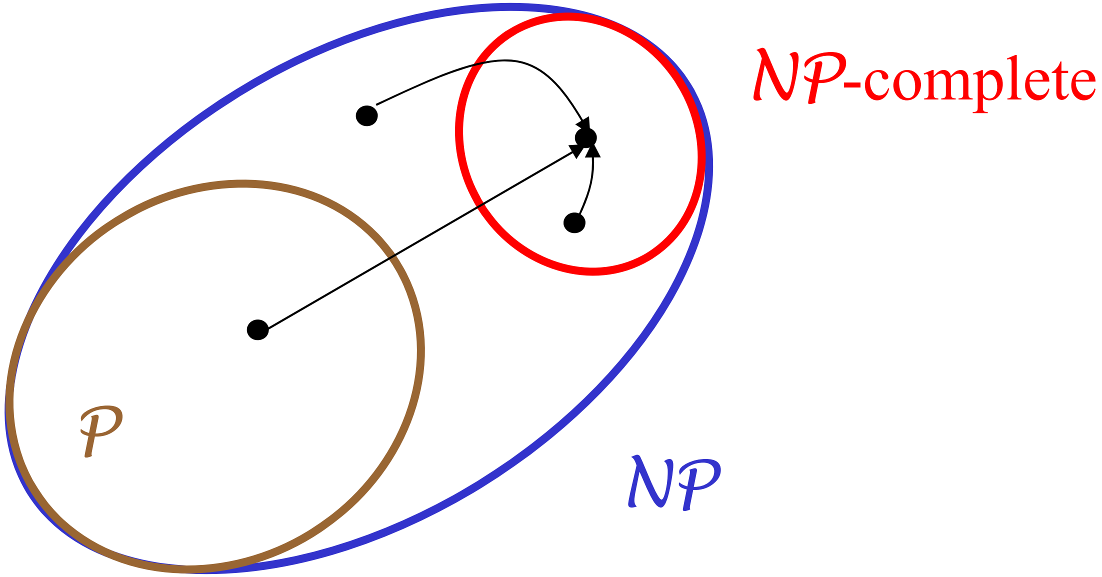
If $P_1 \propto P_2$ and P_2 admits a polynomial-time algorithm, then also P_1 can be solved in polynomial time.

$$P_2 \in \mathcal{P} \Rightarrow P_1 \in \mathcal{P}$$

$\mathcal{NP}$-complete problems

Definition: A problem P is *$\mathcal{NP}$-complete* if and only if

1) P belongs to $\mathcal{NP}$
2) every other problem $P' \in \mathcal{NP}$ can be reduced to P in polynomial time ($P' \propto P$).



Consequence: If there exists a polynomial-time algorithm for any $\mathcal{NP}$-complete problem ($\in \mathcal{P}$), then all problems in $\mathcal{NP}$ can be solved in polynomial time (we would have $\mathcal{P} = \mathcal{NP}$).

This is considered to be **extremely unlikely**

Therefore **$\mathcal{NP}$-completeness** provides **strong evidence** that a problem is **inherently difficult**.

cf. Long list of important recognition problems that are $\mathcal{NP}$-complete and for which no polynomial time algorithms are known.

Do $\mathcal{NP}$-complete problems exist?

Satisfiabilty problem (SAT)

Given m Boolean clauses $C_1, \ldots, C_m$ (disjunctions – OR – of Boolean variables y_j and their complements $\bar{y}_j$), does there exist a truth assignment (of values “true” or ”false” to the variables) satisfying all the clauses?

Example

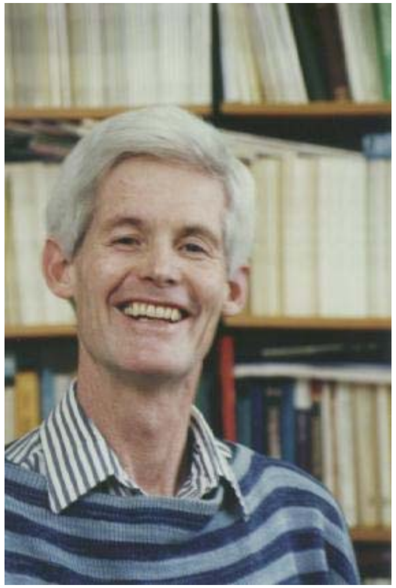
$$C_1 : \ (y_1 \vee y_2 \vee y_3)$$
$$C_2 : \ (\bar{y}_1 \vee \bar{y}_2)$$
$$C_3 : \ (y_2 \vee \bar{y}_3)$$

Truth assignment: y_1 = true, y_2 = false, y_3 = false

First problem proved to be $\mathcal{NP}$-complete:

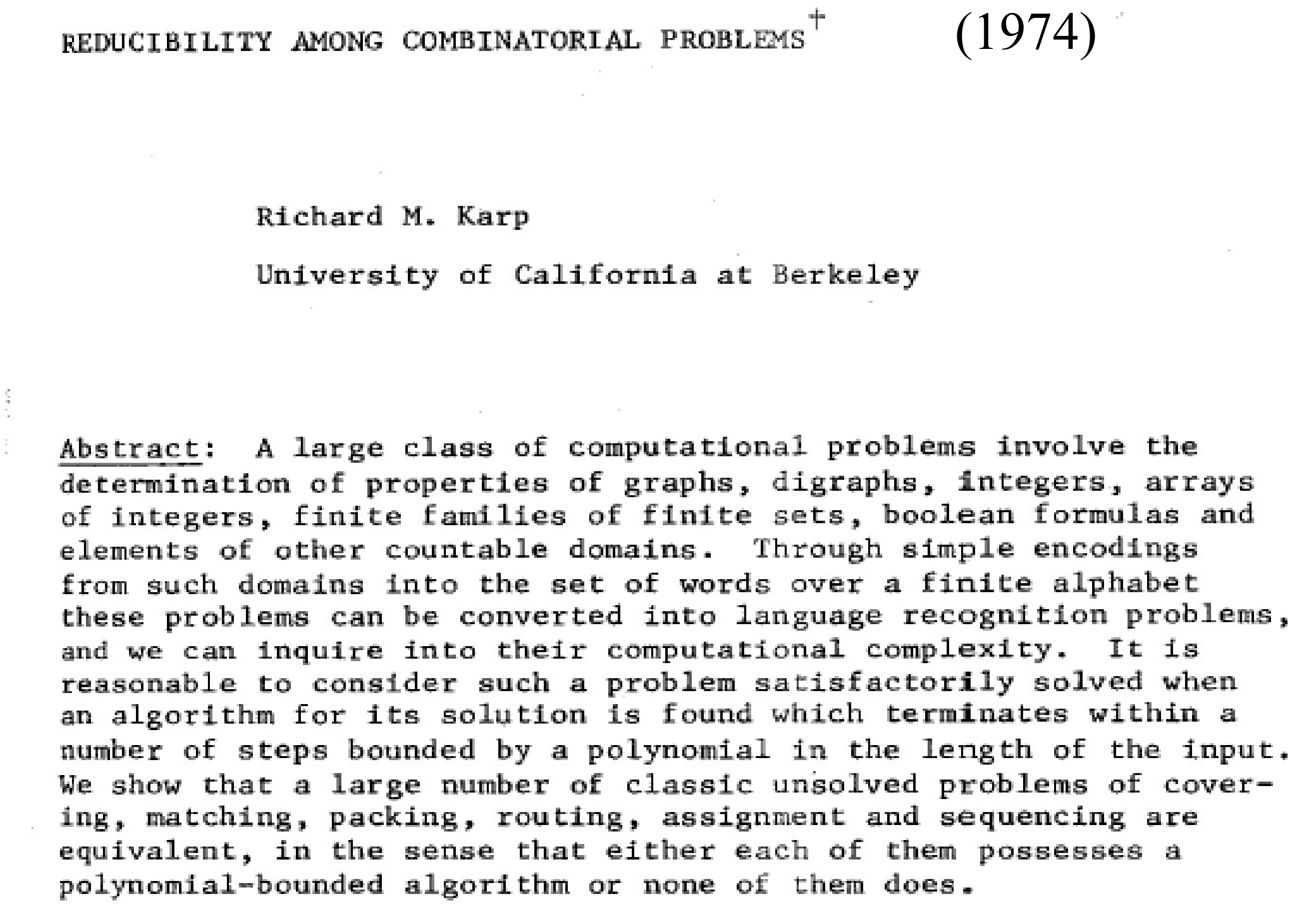
Theorem (Cook 1971)

SAT is $\mathcal{NP}$-complete.



Stephen A.Cook 1939-

Using the characterization of $\mathcal{NP}$ in terms of polynomial time non-deterministic Turing machine and the concept of polynomial time reduction.



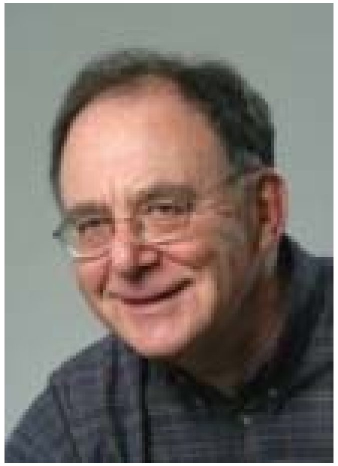
REDUCIBILITY AMONG COMBINATORIAL PROBLEMS+

(1974)

Richard M. Karp

University of California at Berkeley

Abstract: A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings from such domains into the set of words over a finite alphabet these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.



Richard M. Karp 1935-

Show that the recognition versions of 21 discrete optimization problems are $\mathcal{NP}$-complete.

How to show that a problem is $\mathcal{NP}$-complete

To show that $P_2 \in \mathcal{NP}$ is $\mathcal{NP}$-complete it "suffices" to show that an $\mathcal{NP}$-complete problem P_1 reduces in polynomial time to P_2 :

$P \propto P_1, \forall\, P \in \mathcal{NP}$, and $P_1 \propto P_2$ implies by transitivity that

$$P \propto P_2\,, \forall\, \mathrm{P} \in \mathcal{NP}.$$

Example

P_1: Given undirected G with arc costs and an integer L, $\exists$ a Hamiltonian cycle of total cost $\leq L$?

P_2: Given directed G' with arc costs and an integer L', $\exists$ a Hamiltonian circuit of total cost $\leq L'$?

$P_2 \in \mathcal{NP}$ and $P_1 \propto P_2$ with P_1 $\mathcal{NP}$-complete

Other examples of $\mathcal{NP}$-complete problems

- Given undirected $G = (N, E)$, does there exist a Hamiltonian cycle? (Karp 74)
- Given directed $G = (N, A)$, two nodes s and t, and an integer L, ∃ a simple path (with distinct intermediate nodes) from s to t containing a number of arcs $\geq L$? (exercise 3.4)
- Given directed $G = (N, A)$ with arc costs, two nodes s and t, and an integer L, ∃ a simple path from s to t of total cost $\leq L$? (exercise 3.4)
- Given a linear system $A\mathbf{x} \geq \mathbf{b}$ with integer coefficients and binary variables, ∃ a solution $\mathbf{x} \in \{0,1\}^n$? (pages 32-33)

$\mathcal{NP}$-hard problems

Definition: A problem is *$\mathcal{NP}$-hard* if every problem in $\mathcal{NP}$ can be reduced to it in polynomial time (even if it does not belong to $\mathcal{NP}$).

Example

TSP is $\mathcal{NP}$-hard.

Indeed, TSP-r (does there exist a Hamiltonian circuit of total cost $\leq L$?) is $\mathcal{NP}$-complete.

Observation: All optimization problems with an $\mathcal{NP}$-complete recognition version are $\mathcal{NP}$-hard.

<u>Integer Linear Programming</u> (ILP):

Given A $m \times n$, $\mathbf{b}$ $m \times 1$ and $\mathbf{c}$ $n \times 1$ with integer coefficients, find $\mathbf{x} \in \{0, 1\}^n$ that satisfies $A\mathbf{x} \geq \mathbf{b}$ and minimizes $\mathbf{c}^T\mathbf{x}$.

Proposition (Karp 74): ILP is $\mathcal{NP}$-hard. (also exercise 3.3)

Proof We show that ILP recognition version is $\mathcal{NP}$-complete.

<u>ILP-r</u>: Given $A\mathbf{x} \geq \mathbf{b}$ with integer coefficients, $\exists$ a solution $\mathbf{x} \in \{0, 1\}^n$?

1) ILP-r belongs to $\mathcal{NP}$ since: *i*) it is a recognition problem, *ii*) given a solution vector $\mathbf{x} \in \{0, 1\}^n$ we can verify in polynomial time that it satisfies all inequalities of $A\mathbf{x} \geq \mathbf{b}$.

2) Show that the $\mathcal{NP}$-complete problem SAT can be transformed in polynomial time to ILP-r.

For any instance I_1 of SAT we can construct in polynomial (linear) time a special instance I_2 of ILP-r as follows:

I_1 of SAT:	I_2 of ILP-r:
$(y_1 \vee y_2 \vee y_3)$	$x_1 + x_2 + x_3 \geq 1$
$(\overline{y}_1 \vee \overline{y}_2)$	$(1- x_1)+(1- x_2) \geq 1$
$(y_2 \vee \overline{y_3})$	$x_2 +(1- x_3) \geq 1$
$y_i \in \{\text{true, false}\} \quad \forall i \in \{1,2,3\}$	$x_i \in \{0, 1\} \quad \forall\ i \in \{1,2,3\}$

and, clearly, the answer to I_1 is “yes” if and only if the answer of I_2 is “yes”.

Other examples of $\mathcal{NP}$-hard problems

- Given directed $G = (N, A)$ with arc costs, two nodes s and t, find a simple path from s to t of maximum total cost.
  (exercise 3.4)

- Given directed $G = (N, A)$ with arc costs, two nodes s and t, find a simple path from s to t of minimum total cost.
  (exercise 3.4)

- ….