CIS
RESEARCH GROUP

Heuristics & Metaheuristics for Optimization & Learning

Mario Pavone

mpavone@dmi.unict.it

http://www.dmi.unict.it/mpavone/

AA. 2024/2025 - LM 18

TEORIA DELLA COMPLESSITÀ

Problem

- A **problem is a general question to be answered, usually** possessing several **parameters, or free variables, whose** values are left unspecified.
- A problem is described by giving:
  - a general description of all its parameters, and
  - a statement of what properties the **answer, or solution, is** required to satisfy
- An **instance of a problem is obtained by specifying particular** values for all the problem parameters

A Classical Example: TSP problem

- The parameters of this problem consist of a finite set

$$C = c_1, c_2, \ldots, c_m$$

of *"cities"* and, for each pair of cities c_i, c_j in C, the *"distance"* $d(c_i, c_j)$ between them

- A ***solution is an ordering***

$$< c_{\Pi(1)}, c_{\Pi(2)}, \ldots, c_{\Pi(m)} >$$

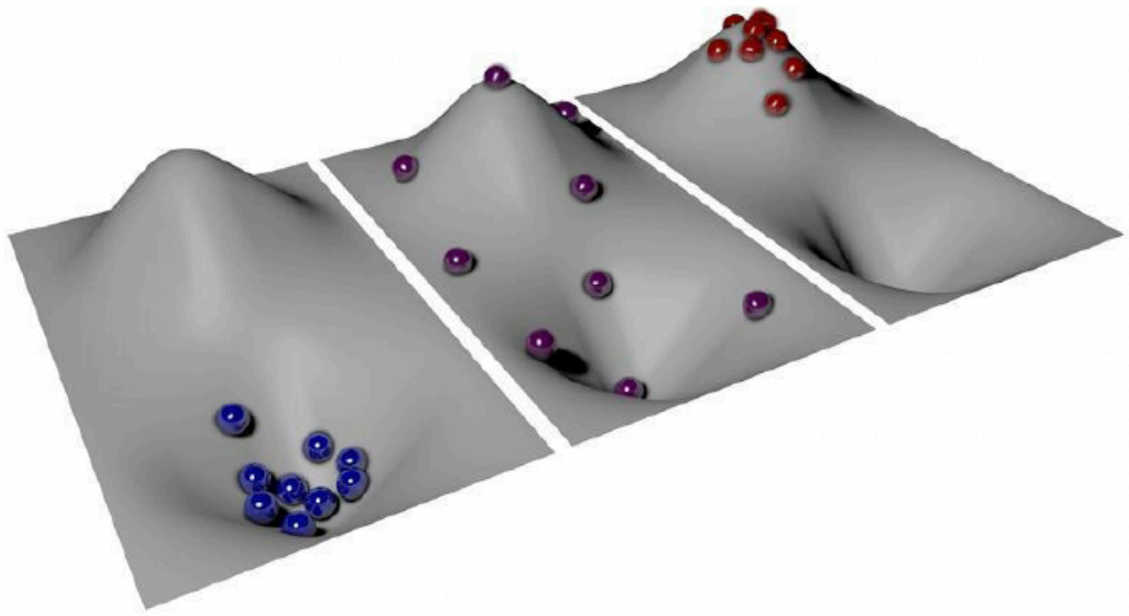
- of the given cities that minimizes

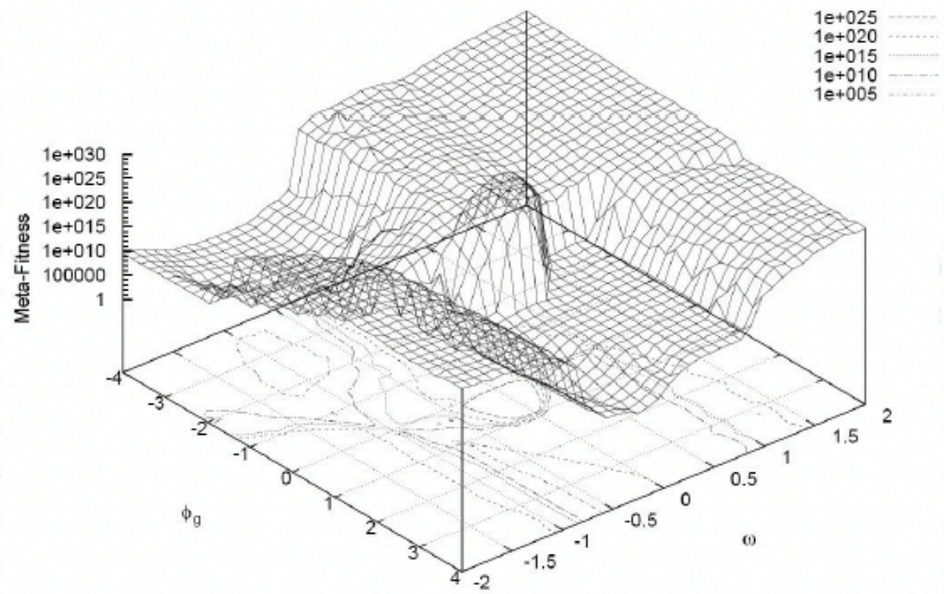
$$\left(\sum_{i=1}^{m-1} d\left(c_{\pi(i)}, c_{\pi(i+1)}\right) \right) + d\left(c_{\pi(m)}, c_{\pi(1)}\right)$$

this expression gives the length of the ***"tour"*** that starts at $c_{\Pi(1)}$, visits each city in sequence, and then returns directly to $c_{\Pi(1)}$ from the last city $c_{\Pi(m)}$

Combinatorial Landscapes

- The notion of landscape is among the rare existing concepts which help to understand **the behaviour of search algorithms** and heuristics and **to characterize the difficulty** of a combinatorial problem.





Example and Relevance of Landscape

- The search Landscape for the *K-SAT problem* is a **N dimensional hypercube** with

  N = number of variables = |V|

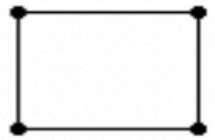
- Combinatorial optimization problems are often **hard to solve** since such problems may have **huge and complex search landscape**

Example: HYPERCUBES

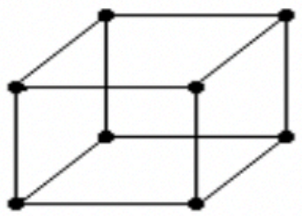
Ipercubo di dimensione 0

Ipercubo di dimensione 1

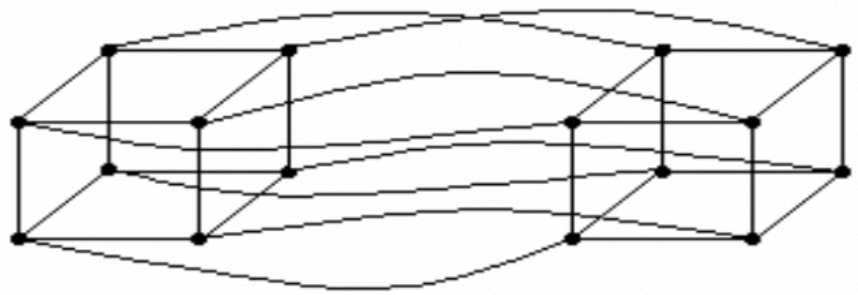
Ipercubo di dimensione 2



Ipercubo di dimensione 3



Ipercubo di dimensione 4



ALGORITHMS

- **Algorithms** are general, step-by-step procedures for solving problems
- For concreteness, we can think of them simply as being **computer programs**, written in some precise computer language.
- An algorithm is said to **solve** a problem ***P*** if that algorithm can be applied to **any** instance ***I*** of ***P*** and is **guaranteed always** to produce a solution for that instance ***I***
- For example:
  - An algorithm does not "solve" the TSP unless it always constructs an ordering that gives a minimum length tour

Why understand the theory of NP-Completeness?

- If we can establish a problem as NP-Complete, we can provide good evidence for its intractability
- We would do better spending our time developing an approximation algorithm rather than searching for a fast algorithm that solves the problem exactly.

Problem: the start point

- a **(abstract) PROBLEM** is a binary relation on a set **I** of problem instances, and a set **S** of problem solutions: $Q: I \rightarrow S$
- Es: Shortest-Path => instance is a triple (G, v_i, v_j), whilst the solution is a sequence of vertices in the graph (path);
- a given problem instance may have more than one solution

- **Instance** of a problem is a list of arguments, one argument for each parameter of the problem

Problem: the start point

- The theory of NP-completeness restrincts attention to **DECISION PROBLEMS**: those having a yes/no solution.
- **Problem**: to refer to a question such as: "Is a given graph G=(V,E) K-colorable?"
- **decision problems**: $Q: I \rightarrow \{0, 1\}$
- in general, if we can solve an optimization problem quickly, we can also solve its related decision problem quickly
- If an optimization problem is easy, therefore its related decision problems is easy as well.

Problem: the start point

- We define a problem Q to be a binary relation on a set I of problem instances and a set S of problem solutions:

$$Q: I \rightarrow S$$

- Shortest-path problem: *finding a shortest path between two given vertices in an unweigthed , undirected graph G=(V, E)*
- instance: is a triple consisting of a graph and two vertices
- solution: a sequence of vertices in the graph
- Shortest-paths are not necessarily unique: a given problem instance may have more than one solution
- **INSTANCE** of a problem is a list of arguments, one argument for each parameter of the problem

Problem: the start point

- The theory of NP-completeness restricts attention to **decision problems**: those having yes/no solution.
- **Decision problem**: to refer to a question such as: "Is a given graph G=(V,E) K-colorable?"
- **decision problems**: ***Q: I → {0, 1}***
- Example: if *i=<G,k>* is an instance of the graph coloring problem then *COL(i)=1 (yes)* if there exist a k-coloring, and *COL(i)=0 (no)* otherwise.
- **Optimization Problems**: those where some value must be minimized or maximized.
- in general, if we can solve an optimization problem quickly, we can also solve its related decision problem quickly

Problem: the start point

- a problem is ***UNDECIDABLE*** if there is no algorithm that takes as input an instance of the problem and determines whether the answer to that instance is "*yes*" or "*no*"
- problems that are solvable by polinomial-time algorithms as being **TRACTABLE**; ones that require superpolynomial time as being **INTRACTABLE**

Problem: the start point

- **NP-complete problems are intractable**
  - If any single NP-complete problem can be solved in polynomial time then every NP-complete problem has a polynomial-time algorithm
- If you can establish a problem as NP-complete you provide good evidence for **its intractability**
- You would then do better spending your time **developing an approximation algorithm** rather than searching for a fast algorithm that solves the problem exactly
- Polynomial-time solvable problems are generally regarded as tractable

Complexity Class P

- An algorithm solves a problem in time $O(T(n))$ if, when it is provided a **problem instance i of length $n=|i|$**, the algorithm can produce the solution in at most $O(T(n))$ time
- A problem is polynomial-time solvable if there exists an algorithm to solve it in time $O(n^k)$ for some constant k
- **COMPLEXITY CLASS P**: the set of decision problems that are solvable in polynomial time
- A function f is polynomial-time computable if there exists a polynomial-time algorithm A that, given any input $x \in I$, produces as output $f(x)$

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: every algorithm we've studied (Algorithms 1) provides polynomial-time solution to some problem
  - We define **P** to be the class of problems solvable in polynomial time
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

Complexity Class NP

- A *decision problem* (yes/no question) is in the *class NP* if it has a nondeterministic polynomial time algorithm
- Informally, such an algorithm:
  1. Guesses a solution (nondeterministically).
  2. Checks deterministically in polynomial time that the answer is correct.

Nondeterminism

- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct
  - If a solution exists, computer always guesses it
  - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds it has a working solution

Complexity Class NP

- A *decision problem* (yes/no question) is in the *class NP* if it has a nondeterministic polynomial time algorithm
- Informally, such an algorithm:
  1. Guesses a solution (nondeterministically).
  2. Checks deterministically in polynomial time that the answer is correct.
- Or equivalently, when the answer is "yes", there is a certificate (a solution meeting the criteria) that can be verified in polynomial time (deterministically)
- The **Complexity Class NP** is the class of problems that can be verified by a polynomial-time algorithm

Complexity Class NP

- A *decision problem* (yes/no question) is in the *class NP* if it has a nondeterministic polynomial time algorithm
- Informally, such an algorithm:

1. Guesses a solution (nondeterministically).
2. Checks deterministically in polynomial time that the answer is correct.

- Or equivalently, when the answer is "yes", there is a certificate (a solution meeting the criteria) that can be verified in polynomial time (deterministically)
- The **Complexity Class NP** is the class of problems that can be verified by a polynomial-time algorithm
- If $Q \in P$ then $Q \in NP$ => $\boldsymbol{P \subseteq NP}$
- Open question is: $\boldsymbol{P = NP}$

Hamiltonian Cycle Problem

- A hamiltonian cycle of an undirected graph G=(V,E) is a simple cycle that contains each vertex in V.
- A graph that contains a hamiltonian cycle is said to be hamiltonian
  - NOT ALL GRAPHS ARE HAMILTONIAN
- Hamiltonian-cycle problem: "Does a graph G have a hamiltonian cycle?"
  - HAM-CYCLE={<G>: G is a hamiltonian graph}

Hamiltonian Cycle Problem

- algorithm: lists all permutations of the vertices and check each permutation to see if it is a hamiltonian path
  - there are m! possible permutations of the vertices
- CHECK if the provided cycle is hamiltonian: if it is a permutation of the vertices of V and each consecutive edges along the cycle exists in the graph
- This verification can be implemented to run in $O(n^2)$ time => a hamiltonian cycle exists in a graph can be verified in polynomial time
  - N is the lenght of the econding of G

Hamiltonian Cycle Problem

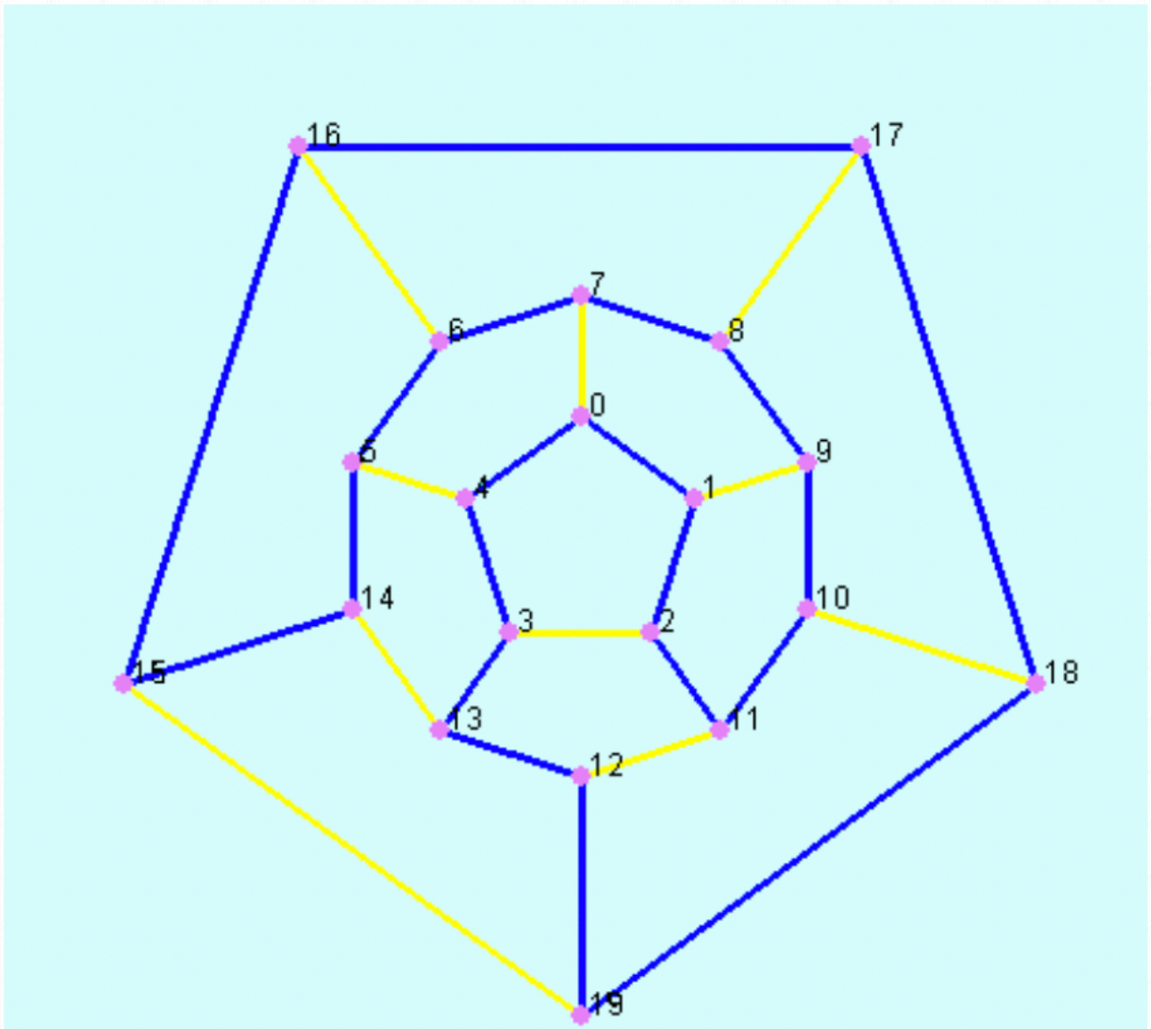
Hamiltonian Cycle Problem is in complexity class NP

hamiltonian cycle exists in a graph can be verified in polynomial time

- N is the lenght of the econding of G

Hamiltonian Cycle Problem

- *Hamiltonian Cycle* (input: a graph *G*)
- *Does G have a Hamiltonian cycle?*



Solution:

0, 1, 2, 11, 10, 9, 8, 7, 6, 5, 14, 15, 6, 17, 18, 19, 12, 13, 3, 4

NP Complete Problems

- The class of *problems in NP which are the "hardest"* are called the ***NP-complete* problems**

- A problem Q in NP is **NP-complete** if the existence of a polynomial time algorithm for Q implies the *existence of a polynomial time algorithm for all problems in* NP

- Steve Cook in 1971 proved that SAT is NP-complete (*Teorema di Cook-Levin*).

Reduction

- The crux of NP-Completeness is *reducibility*
  - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P
    - This rephrasing is called *transformation*
- Intuitively: If P reduces to Q, P is "no harder to solve" than Q

Reducibility

- An example:
  - P: Given a set of Booleans, is at least one TRUE?
  - Q: Given a set of integers, is their sum positive?
  - Transformation: $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ where $y_i = 1$ if $x_i =$ TRUE, $y_i = 0$ if $x_i =$ FALSE

Using Reductions

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Complete:
  - If P is NP-Complete, $P \in \mathbf{NP}$ and all problems R are reducible to P
  - Formally: $R \leq_p P \; \forall \; R \in \mathbf{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP-Complete

NP-Hard and NP-Complete

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in \mathbf{NP}$ are reducible to P, then P is *NP-Hard*
  - We say P is *NP-Complete* if P is NP-Hard and $P \in \mathbf{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP- Complete

NP-Hard and NP-Complete

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in \mathbf{NP}$ are reducible to P, then P is *NP-Hard*
  - We say P is *NP-Complete* if P is NP-Hard and $P \in \mathbf{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP- Complete

NP-Hard and NP-Complete

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in \mathbf{NP}$ are reducible to P, then P is *NP-Hard*
  - We say P is *NP-Complete* if P is NP-Hard and $P \in \mathbf{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP- Complete

NP-Hard and NP-Complete

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in \mathbf{NP}$ are reducible to P, then P is *NP-Hard*
  - We say P is *NP-Complete* if P is NP-Hard and $P \in \mathbf{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP- Complete

Teorema

Definizione

Un problema B é detto **NP-completo** se:

1) $\mathbf{B \in NP}$
2) $\mathbf{\forall A \in NP}$ vale che $\mathbf{A \leq_P B}$

Teorema

Se B é un problema NP-completo tale che $B \leq_P C$, per qualche problema C, allora B é NP-hard. Se poi $B \in NP$, allora vale anche che B é NP-completo.

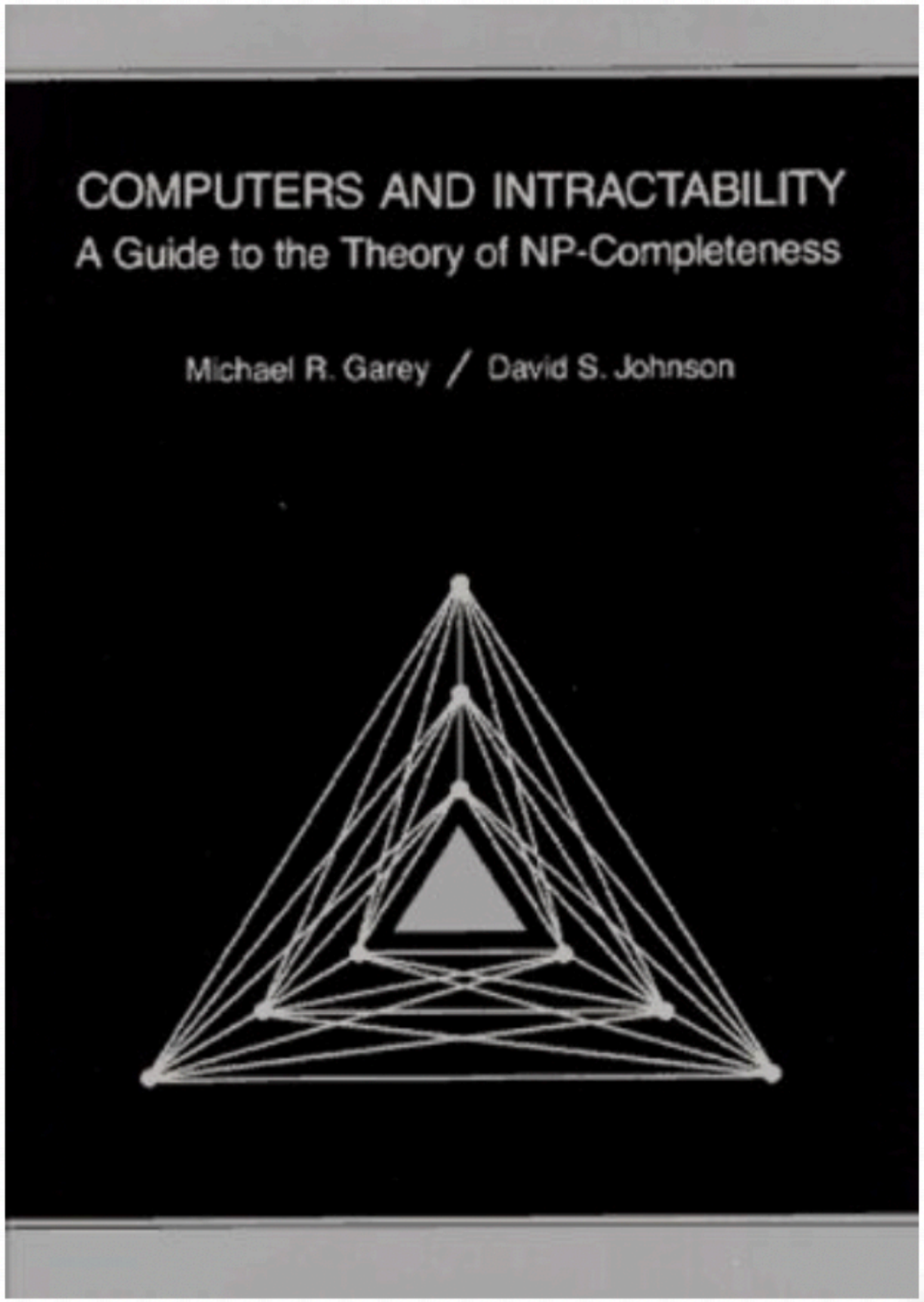
BIBLIOGRAPHY

M. R. Garey and D. S. Johnson, ***Computers and Intractability: A Guide to the Theory of NP-Completness***, W. H. Freeman, 1st ed. (1979).

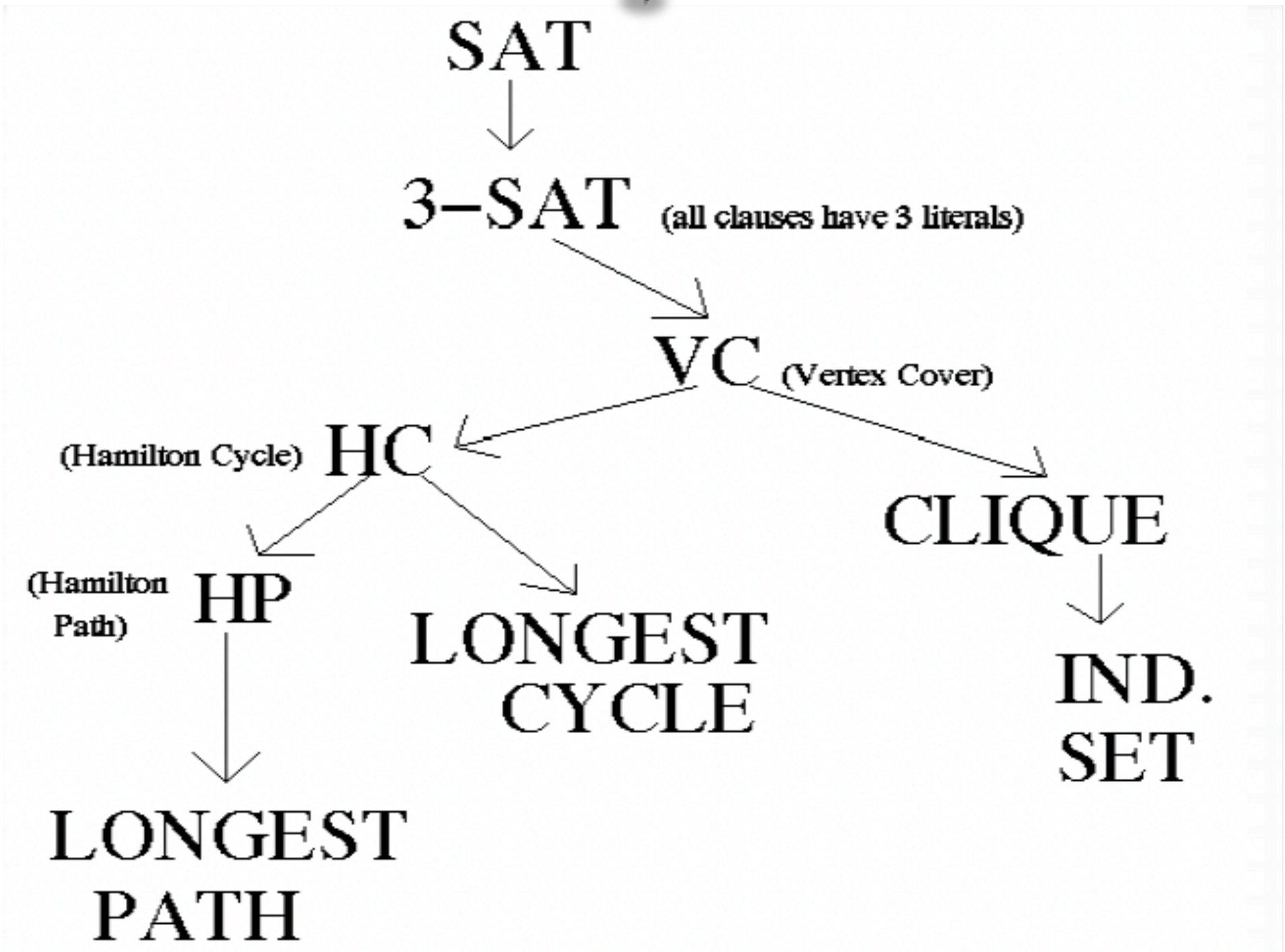
OR:

A compendium of NP optimization problems at the link:

http://www.csc.kth.se/~viggo/problemlist/



- Some problems in **NP** not known to be in **P**:
  - **Hamilton Path/Cycle**
  - **Independent Set**
  - **Satisfiability**
  - **Traveling Salesman Problem**
  - **Graph Coloring Problem**
  - **……**



History of NP–completeness Reductions

SAT (Satisfiability)

- **Variables:** $u_1, u_2, u_3, \ldots u_k$
- A **literal** is a variable u_i or the negation of a variable $\neg u_i$
- If u is set to *true* then $\neg u$ is *false* and if u is set to *false* then $\neg u$ is *true*
- A **clause** is a set of literals. A clause is *true* if at least one of the literals in the clause is *true*
- The **input to SAT** is a collection of clauses.

SAT (Satisfability)

- The output is the answer to: *"Is there an assignment of true/false to the variables so that every clause is satisfied (**satisfied** means the clause is true)?"*
- If the answer is yes, such an assignment of the variables is called a **truth assignment**.
- SAT is in NP: Certificate is true/false value for each variable in satisfying assignment.

ESEMPIO:

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$

ESEMPIO:

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$

- La formula é soddisfacibile, semplicemente ponendo $x_1=1$, $x_2=0$, e $x_3=0$.
- Infatti

$$(1 \vee 0 \vee 1) \wedge (0 \vee 1 \vee 1) \wedge (1 \vee 1) \wedge (0 \vee 1) = 1$$

- La seguente formula, invece, non é soddisfacibile

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

TEOREMA DI COOK-LEVIN

SAT è NP-Completo

Dimostrazione

In breve, il teorema di Cook-Levin fa vedere che:

– il problema SAT in NP può essere deciso da un opportuno modello di calcolo: la Macchina di Turing non Deterministica (NTM)

– data la descrizione di una NTM ***n*** e un input ***w*** per ***n***, si può costruire una espressione in forma normale congiuntiva ϕ_w che è soddisfacibile se e solo se l'output di ***n*** su input *w* è accettante

– la lunghezza della formula ϕ_w è polinomiale in $|n|$ e in $|w|$

Reduction

- **If SAT is solvable in polynomial time, then so is HAMILTON CYCLE**

- Graph **G** has n vertices *0*, *1*, *2*, …, *n-1*
- Variables: $x_{i,j}$ $(0 \le i, j \le n\text{-}1)$
- Meaning: node i is in position j in Hamilton cycle

Reduction

- Conditions to ensure a Hamilton cycle:
  1. Exactly one node appears in position *j*
- At least one node appears in position j
  - For each *j*, add a clause:

$$(x_{0,j} \text{ OR } x_{1,j} \text{ OR } x_{2,j} \text{ OR } \ldots \text{ OR } x_{n-1,j})$$

- At most one node appears in position j.
  - For each pair of vertices *i,k* add a clause

$$(\text{not } x_{i,j} \text{ OR not } x_{k,j})$$

Reduction

2. Vertex i occurs exactly once on the cycle

0 Vertex i occurs at least once

0 For each i, add a clause:

$$(x_{i,0} \text{ OR } x_{i,1} \text{ OR } x_{i,2} \text{ OR } \ldots \text{ OR } x_{i,n-1})$$

0 Vertex i occurs at most once

0 For each vertex i and pair of positions j,k add a clause

$$(\text{not } x_{i,j} \text{ OR not } x_{i,k})$$

Reduction

3. Consecutive vertices of the cycle are connected by an edge of the graph.

0 For each edge (i, k) which is missing from the graph and for each j add a clause

$$(\text{not } x_{i,j} \text{ OR not } x_{k,j+1 \bmod n})$$

Traveling Salesman Problem

- The well-known *traveling salesman problem*:
  - Optimization variant: a salesman must travel to n cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
    - Model as complete graph with cost $c(i,j)$ to go from city i to city j
- *How would we turn this into a decision problem?*
  - A: ask if $\exists$ a TSP with cost $< k$

Traveling Salesman Problem

- The steps to prove TSP is NP-Complete:
  - Prove that TSP $\in$ **NP**
  - Reduce the undirected hamiltonian cycle problem to the TSP
    - So if we had a TSP-solver, we could use it to solve the hamilitonian cycle problem in polynomial time
    - *How can we transform an instance of the hamiltonian cycle problem to an instance of the TSP?*
    - *Can we do this in polynomial time?*

ESEMPIO: CLIQUE

ISTANZA: Un grafo $G = (V, E)$ ed un intero k

DOMANDA: G ha una clique di taglia k?

(ovvero $\exists\, V' \subseteq V$, $|V'| = k$, tale che $\forall\, u, v \in V'$, $u \neq v$ vale che $(u, v) \in E$)

Teorema

CLIQUE é NP-completo

Dimostrazione

Proveremo che CLIQUE $\in$ NP, successivamente proveremo che 3SAT $\leq_P$ CLIQUE. Dal Teorema 1 discenderà che CLIQUE é NP-completo.

Un algoritmo di verifica per il problema CLIQUE é il seguente: avendo in input la coppia (G, k) e come certificato un sottoinsieme $X \subseteq V$, con $|X| = k$ (potenziale soluzione al problema CLIQUE), l'algoritmo controlla tutte le coppie di vertici (u, v), con $u, v \in X$, $u \neq v$, e produce in output SI *se e solo se ciascuna* di queste coppie é connessa da un arco di E. Il tempo di esecuzione dell'algoritmo é $O(n^2)$, se $n = |V|$.

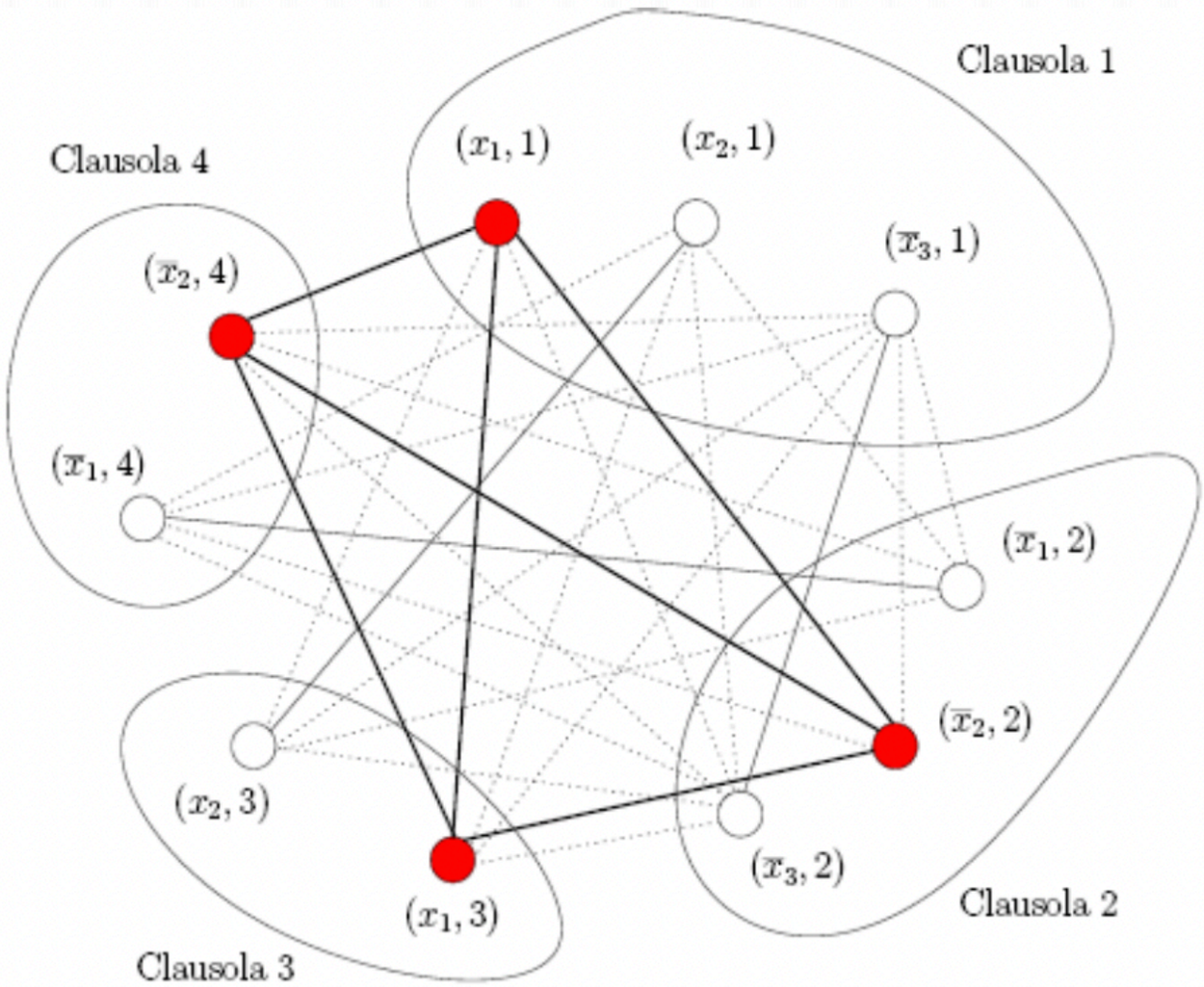
3SAT $\leq_P$ CLIQUE

- Sia $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ una istanza di 3SAT in una istanza (G_ϕ , k) di Clique come segue:

1. Per ogni letterale scriviamo un nodo
2. Colleghiamo tutti i nodi ad eccezione:
   - Nodi nella medesima clausola
   - Nodi che rappresentano letterali opposti

ESEMPIO

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$



ESEMPIO: VERTEX-COVER

ISTANZA: Un grafo $G = (V,E)$ ed un intero k

DOMANDA: esiste $V' \subseteq V$, $|V'| = k$, tale che $\forall (u, v) \in E$ o vale che $u \in V'$ oppure $v \in V'$?

Teorema

Vertex-Cover é NP-completo

Dimostrazione

Proveremo che Vertex-Cover $\in$ NP, successivamente proveremo che 3SAT $\leq_P$ Vertex-Cover. Dal Teorema 1 discenderà che Vertex-Cover é NP-completo.

Che il problema VERTEX COVER sia in NP é ovvio. Un algoritmo di verifica prende in input l'istanza di input (G, k) ed un certificato $V' \subseteq V$, $|V'| = k$, e controlla se

$\forall (u, v) \in E$ vale che $u \in V'$ oppure che $v \in V'$. Il controllo può essere fatto in tempo $O(n^2)$.