Foundations of Operations Research

Master of Science in Computer Engineering

Roberto Cordone
roberto.cordone@unimi.it

Tuesday 13.15 - 15.15

Thursday 10.15 - 13.15

http://homes.di.unimi.it/~cordone/courses/2014-for/2014-for.html

Computational complexity

The theory of computational complexity aims to answer these questions:

- what are the resources required by an algorithm to solve a problem? (*this question focuses on a specific algorithm*)
- what are the resources intrinsically required to solve a problem? (*this question concern all possible algorithm, even still unknown*)

By resource we mean the time required by a computing machine (we neglect here the so called *space complexity*, concerning memory)

These questions are by no means easy: even giving them a clear and unambiguous form is a serious task

- what is a problem?
- what is a computing machine, and an algorithm?
- how is time measured?

By investigating a problem independently from the known algorithms, one can understand

- whether better algorithms could exist or not
- the kind of algorithm more appropriate for the problem

Problems and instances

A problem is a general question on a property of a family of mathematical objects

- *Is a given number prime?*
- *What is the length of the shortest path between two given nodes in a given arc-weighted directed graph?*

An instance is a specific question on a property of a single mathematical object

- *Is* 29 *a prime number?*
- *What is the length of the shortest path between nodes* 3 *and* 11 *in graph* $G = (N, A)$ *with* $N = \dots$ *and* $A = \dots$*, with respect to weight function* c*. where* $c_{12} = \dots$*?*

An instance can be coded as a sequence of numbers in binary format: formally speaking, an instance is a binary string
The size of the instance is the length of the string (number of bits)

A problem can be seen as the (usually infinite) set of instances concerning similar objects: formally speaking, a problem is a set of binary strings

Machines and algorithms

A machine is a formal mechanism to manipulate binary strings

- applying deterministic rules
- to perform elementary modifications on a current binary string, which is initialized with an instance of a problem
- terminating when some predefined condition is satisfied
- producing an output binary string

If the machine terminates, the solution of the given instance is formally represented by the output binary string

An elementary operation is each step of the rule application process

An algorithm is a machine which, when fed with any instance of a given problem, terminates and provides the corresponding solution

Why so much abstraction?

Because, technology advances and we want general theoretical results, independent from technological progress

Computational models

A computational model is a family of formal mechanism to manipulate binary strings having a similar structure

- they apply different rules
- but they apply the rules in the same way

The most famous computational models are

- the Turing machine, which works at a very low level moving bit-by-bit on the input string and possibly applying to it single-bit modifications
- the *RAM* machine, which works at a higher level directly accessing data coded into integer numbers and applying to them arithmetic operations (*it is similar to a simple standard computer*)

In general, given two computational models

- the machines they include are able to solve different sets of problems
- if they both include a machine to solve a given problem, the two machines require a different number of steps

However, all known computational models which have a physical counterpart

- solve exactly the same problems
- require a number of steps bounded by a polynomial in the number of steps required by the machine of the other model

Time complexity

Some reminders from lesson n. 3

- Time complexity is measured as the number of elementary operations (it depends on the computational model, but not on the single computer)
- It is measured in a worst-case perspective, that is considering the instance which requires the maximum number of operations for each size n
- The function $T(n)$ is simplified considering only its asymptotic behaviour, expressed through the "big-O" notation

We must anyway make clear

1. the computational model used (usually, the *RAM* machine)
2. the definition of instance size: either the length of the binary string or some higher-level parameter (e. g., number of nodes, or arcs)

These choices imply different complexity evaluations

However, they have no impact on the distinction between

- polynomial algorithms: $\exists d \in \mathbb{R}^+ : T \in O\left(n^d\right)$ (denoted as "efficient")
- exponential algorithms: $\exists d \in \mathbb{R}^+ : T \in O\left(d^n\right)$

Intrinsic complexity of a problem

The intrinsic complexity of a problem is defined as the smallest complexity of any possible algorithm for that problem (*in other words, the complexity of the best possible algorithm*)

This definition refers also to algorithm not yet known (*how can it be?*)

In general, the intrinsic complexity is unknown, but we know

- upper bounds: the complexity of any known algorithm is certainly not smaller
- lower bounds: some refined deduction on the fact that certain elementary operations are absolutely required (*trivial example: reading all of the data*)

Polynomial problems

A polynomial problem is a problem whose intrinsic complexity is polynomial (*note: a polynomial upper bound is enough; the exact complexity is unnecessary*

Even if all known algorithms for a problem are exponential, the problem could be easy

- the first polynomial algorithm for linear programming was proposed in 1979
- the first polynomial algorithm to test whether a number is prime or not was proposed in 2002

There is a huge number of problems with

- an exponential upper bound (inefficient algorithms)
- a polynomial lower bound (a small number of unavoidable operations)

Example

The most famous problem with an unsettled complexity is perhaps the Travelling Salesman Problem; given

- a directed graph $G = (N, A)$
- a cost function defined on the arcs $c : A \to \mathbb{R}$

find a circuit $C^* \subseteq A$

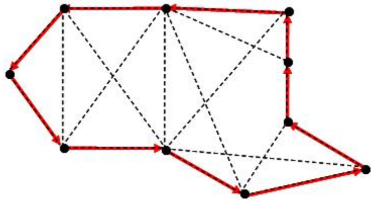
1. visiting all the nodes exactly once

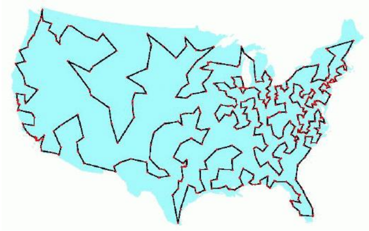
$$\left| C^* \cap \Delta_i^+ \right| = 1 \text{ for all } i \in N$$

2. with the minimum total cost

$$\sum_{(i,j) \in C^*} c_{ij} \leq \sum_{(i,j) \in C} c_{ij}$$

for all circuits C satisfying condition 1





Transformations

A nice and powerful approach to solving problems is to transform them into other problems

- the project planning problem into a longest path problem
- the minimum cut problem into a maximum flow problem
- the maximum flow problem into a linear programming problem

Polynomial transformation $P \preceq Q$: given two problems P and Q, suppose that

- it is possible to transform in polynomial time
  any instance of P into an instance of Q
- an algorithm to solve problem Q is known
- it is possible to transform in polynomial time
  the solution of the instance of Q into that of the original instance of P

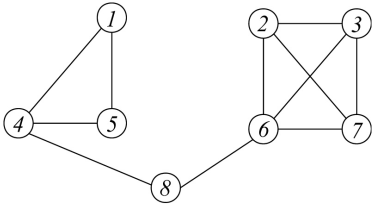
Such a transformation provides an algorithm to solve problem P, and in particular if the algorithm for Q is polynomial, so is also the algorithm for P

This is useful even if no (polynomial) algorithm for Q is currently known:
it suggests to focus on Q, as any positive result will immediately extend to P

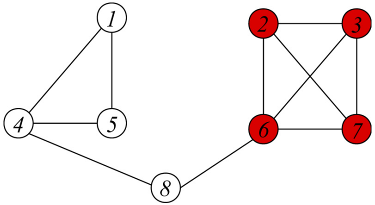
Example

k-clique problem (k-CP): given an undirected graph $G = (V, E)$, does it contain a subset of k pairwise adjacent vertices?

An instance of the k-CP



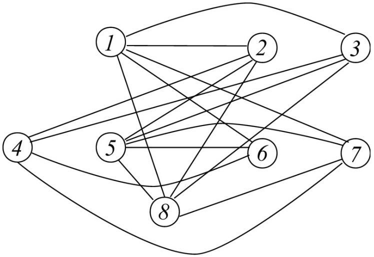
A k-clique with $k = 4$



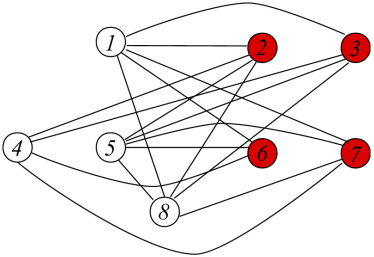
Example

k-independent set problem (*k-ISP*): given an undirected graph $G = (V, E)$, does it contain a subset of k pairwise nonadjacent vertices?

An instance of the *k-ISP*



A *k*-independent set with $k = 4$

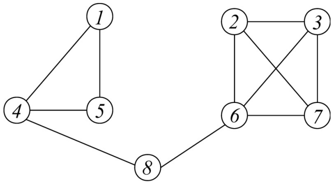


Example

$k\text{-}CP \preceq k\text{-}ISP$ $\qquad$ $(k\text{-}ISP \preceq k\text{-}CP)$

1. Start from the k-CP (k-ISP) instance $G = (V, E)$
2. Build the complementary graph $\bar{G} = (V, \bar{E})$: same vertex set, an edge for each unordered pair of vertices $\notin E$
3. Use $\bar{G}$ as an instance of the k-ISP (k-CP) and apply an algorithm to solve it
4. Identify the vertices in the k-independent set (clique) of graph $\bar{G}$: the corresponding vertices in graph G form a k-clique (independent set)

Corresponding instances of the two problems have the same solution (*true* or *false*)

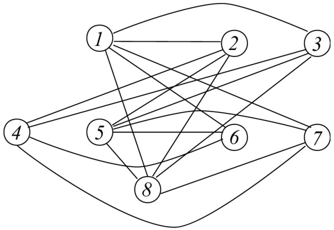


Example

$k\text{-}CP \preceq k\text{-}ISP$ $\quad\quad$ $(k\text{-}ISP \preceq k\text{-}CP)$

1. Start from the $k\text{-}CP$ ($k\text{-}ISP$) instance $G = (V, E)$
2. Build the complementary graph $\bar{G} = (V, \bar{E})$: same vertex set, an edge for each unordered pair of vertices $\notin E$
3. Use $\bar{G}$ as an instance of the $k\text{-}ISP$ ($k\text{-}CP$) and apply an algorithm to solve it
4. Identify the vertices in the k-independent set (clique) of graph $\bar{G}$: the corresponding vertices in graph G form a k-clique (independent set)

Corresponding instances of the two problems have the same solution (*true* or *false*)

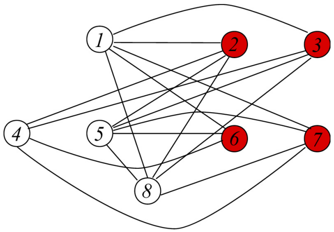


Example

$k\text{-}CP \preceq k\text{-}ISP$ $\quad$ $(k\text{-}ISP \preceq k\text{-}CP)$

1. Start from the $k\text{-}CP$ ($k\text{-}ISP$) instance $G = (V, E)$
2. Build the complementary graph $\bar{G} = (V, \bar{E})$: same vertex set, an edge for each unordered pair of vertices $\notin E$
3. Use $\bar{G}$ as an instance of the $k\text{-}ISP$ ($k\text{-}CP$) and apply an algorithm to solve it
4. Identify the vertices in the k-independent set (clique) of graph $\bar{G}$: the corresponding vertices in graph G form a k-clique (independent set)

Corresponding instances of the two problems have the same solution (*true* or *false*)

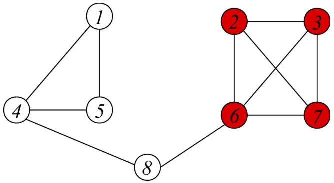


Example

$k\text{-}CP \preceq k\text{-}ISP$ $\quad$ $(k\text{-}ISP \preceq k\text{-}CP)$

1. Start from the k-CP (k-ISP) instance $G = (V, E)$
2. Build the complementary graph $\bar{G} = (V, \bar{E})$: same vertex set, an edge for each unordered pair of vertices $\notin E$
3. Use $\bar{G}$ as an instance of the k-ISP (k-CP) and apply an algorithm to solve it
4. Identify the vertices in the k-independent set (clique) of graph $\bar{G}$: the corresponding vertices in graph G form a k-clique (independent set)

Corresponding instances of the two problems have the same solution (*true* or *false*)



Relative complexity

Another useful effect is that polynomial transformations allow to compare the intrinsic complexity of different problems

Focus on the distinction between polynomial and exponential problems
The algorithm for P designed by transformation uses as a subroutine an algorithm for Q; it requires more time, but

- it requires only a polynomially larger time
- P might admit better algorithms

Consequently

- if Q is polynomial, P is also polynomial
- if Q is exponential, P could still be polynomial

Therefore, if $P \preceq Q$, then P is intrinsically not more complex than Q

The study of polynomial transformations aims to

- extend old algorithms to new problems
- identify the hardest problems, on which research should focus

Recognition problems

A recognition problem is a problem whose solution can be only *true* or *false*:

- positive instances have solution *true*
- negative instances have solution *false*

Example: “Is n a prime number?” or “Does vector V contain number x?”

Any optimization problem can be reformulated as a recognition problem by introducing an auxiliary threshold:

- optimization version: given a feasible region X and an objective function f, what is the minimum of f in X?

$$\min_{x \in X} f = f^* = ?$$

- recognition version: given a feasible region X, an objective function f and a threshold k, is there a solution in X with a value of f not larger than k?

$$\exists x \in X : f(x) \leq k?$$

The two versions are polynomially equivalent:

- given the recognition version, solve the optimization one and compare f^* to the threshold k: if $f^* \leq k$, the solution is *true*, otherwise it is *false*
- given the optimization version, introduce an auxiliary threshold k and solve the recognition version: if the solution is *true*, decrease k, otherwise, increase k; find f^* by dychotomic search

Truth certificates

Given a positive instance I of a recognition problem P, a truth certificate for I is a binary string D which allows to check in polynomial time that the solution of I is *true*; in general, the certificate D is strictly related to the instance I

Example: *the incidence vector of a* 4*-clique in a specific graph* G *allows to check that* G *includes a clique of at least* 4 *vertices*

For several problems

- no polynomial algorithm is known to obtain the solution
- a polynomial algorithm is known to check if a given binary string is a truth certificate for a given instance
    - if it is, the solution of the instance is *true*
    - if it is not, we do not know whether the solution of the instance is *false* or the certificate is simply not valid for that instance

Factorization problem: *Does* n *admit any factor* $\leq k$*?*
Given instance (n, k) with $n = 77$, $k = 10$

- $(7, 11)$ is a truth certificate: $7 \cdot 11 = n$ and $7 \leq k \Rightarrow (77, 10)$ is positive
- $(5, 6)$ is not a truth certificate: $5 \cdot 6 \neq n \Rightarrow$ we know nothing
  ($(77, 10)$ is actually positive, but $(5, 6)$ is nonvalid)

Given instance (n, k) with $n = 143$, $k = 10$

- $(2, 12)$ is not a truth certificate: $2 \cdot 12 \neq n \Rightarrow$ we know nothing
  ($(143, 10)$ is actually negative, because $143 = 11 \cdot 13$, both $\geq k$)

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

$\mathcal{P}$ is the class of all polynomial problems, also denoted as "easy" or "well-solved" problems

$\mathcal{NP}$ is the class of all problems which admit a truth certificate for each positive instance and a polynomial algorithm to check its validity

If a problem is in $\mathcal{P}$, any string is a valid truth certificate for any instance, because the solution can be computed directly (D can be unrelated to I)

Therefore

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

but it is not known whether $\mathcal{P} \subset \mathcal{NP}$ or $\mathcal{P} = \mathcal{NP}$

$\mathcal{P}$ versus $\mathcal{NP}$

The question $\mathcal{P} \subset \mathcal{NP}$ vs. $\mathcal{P} = \mathcal{NP}$ is one of the seven *Millennium Problems*
The Clay Mathematical Institute will pay a 1 000 000 dollar prize for its solution

The practical reason is that a huge number of practical problems of large economic relevance are in $\mathcal{NP}$, but it is not known whether they are also in $\mathcal{P}$

- in most cases, it would be desirable to prove that they are in $\mathcal{P}$, finding an efficient algorithm to solve them (*time is money*)
- in some cases, it would be desirable to prove that they are not in $\mathcal{P}$, because their hardness is used as an advantage (*public key cryptography is based on the hardness of finding the prime factors of a number*)

Most people conjecture that $\mathcal{P} \subset \mathcal{NP}$

Some problems in $\mathcal{NP}$ are particularly relevant, because all other problems in $\mathcal{NP}$ can be transformed into them:

- they are the hardest problem in $\mathcal{NP}$
- if a polynomial algorithm could be found for one of them,
  it could be extended to all problems $\mathcal{NP}$
- on a much more limited scale, if a reasonably good algorithm can be found for one of them, it can be used also for the other ones

The satisfiability problem (*SAT*)

- a Boolean variable x_j is a variable assuming only values *true* and *false*
- a literal is a simple function consisting of a Boolean variable x_j or its negation $\bar{x}_j$
- a Boolean formula is a function obtained combining literals through logical sums (*OR*) or products (*AND*)

Example: $f = (x_1\bar{x}_2 + x_1x_2\bar{x}_3)(\bar{x}_1 + x_3)$

Conjunctive normal form (*CNF*) is a Boolean formula composed by a product of sums of literals

Example: $f' = x_1(\bar{x}_1 + \bar{x}_2)(\bar{x}_1 + x_3)$

Satisfiability problem: Given a *CNF* $f(x_1, \dots, x_n)$, is there an assignment of values to variables x_j such that f is *true*?

x_1	0	0	0	0	1	1	1	1
x_2	0	0	1	1	0	0	1	1
x_3	0	1	0	1	0	1	0	1
f	0	0	0	0	0	1	0	0

$f = x_1(\bar{x}_1 + \bar{x}_2)(\bar{x}_1 + x_3)$

SAT and $\mathcal{NP}$-completeness

SAT is clearly in $\mathcal{NP}$: a truth certificate is given by the n binary values of the truth assignment and it is easy to check its validity

But no known algorithm is polynomial (e. g., evaluating all possible assignments takes exponential time, since they are 2^n)

In 1971, Cook proved that all problems in $\mathcal{NP}$ can be transformed into SAT, which means that if one solves SAT, one solves the whole of $\mathcal{NP}$

An $\mathcal{NP}$-complete problem is a problem

1. belonging to $\mathcal{NP}$
2. such that all problems belonging to $\mathcal{NP}$ can be reduced to it

SAT is $\mathcal{NP}$-complete
If SAT can be transformed into problem P, then P is also $\mathcal{NP}$-complete
(*just reduce all other problems to SAT and then SAT to P*)

Several problems have been proved to be $\mathcal{NP}$-complete
Each of them can be used to solve the other problems in $\mathcal{NP}$, and any progress in its solution immediately reflects on the other problems

Besides SAT, the most relevant $\mathcal{NP}$-complete is *Integer Linear Programming*