Introduction to Hard Computational Problems

Marcin Sydow

Beyond Polynomial-time algorithms

Most of the computational problems discussed previously are solvable by polynomial-time algorithms.

Not all important problems are such. There exist many important problems for which no algorithm faster than exponential is known (e.g. TSP, set cover, etc.)

There are even problems for which there are no algorithms at all! (famous "Turing's halting problem")

Hard and easy problems

- Euler cycle vs hamiltonian cycle in a graph
- CNF vs DNF satisfiability
- shortest vs longest path in a graph

In each pair above, one problem is computationally easy (polynomial algorithm exists) and another hard (no polynomial algorithm is known), despite seemingly similar formulations.

Polynomial Algorithms

Loosely speaking, a problem is "**polynomial**" if there exists an algoritm that solves it, whose time complexity is upper-bounded by a polynomial function of data size

Examples: sorting (O(n logn)), minimum spanning tree, huffman encoding, etc.

Polynomial problems are widely regarded as being computationally tractable ("easy").

Problem:
Honestly, a polynomial algorithm of $\Theta(n^{100})$ (for example) complexity is practically useless (and a $\Theta(2^n)$ – exponential – algorithm would be faster even for medium values of n)
$n^{100} = 2^n, 100 lgn = n, n/lgn = 100, n \approx 1000$

Why polynomial algorithms are regarded as "easy"

The choice of polynomial-time-solvable problems as "easy" is appealing for several reasons:

1. in practice, the exponents are not very high and usually get lower over time (as faster versions of algorithms are invented)
2. the family of polynomial algorithms is closed under summation, multiplication, composition
3. multiplicative factor hardware acceleration can be translated into multiplicative factor computation time acceleration (not true with exponential complexity - an additive acceleration in this case)
4. equivalence in various computational models (e.g. Turing machine, parallel machines, etc.)

P, NP, NP-complete...

The class of polynomially solvable problems is denoted as P.

NP is a class of problems such that their (given) solution can be verified in a polynomial time.

$NP - complete$ is a sub-class of NP representing the hardest problems in NP.

There is no known polynomial algorithm to solve any NP-complete problem

However, nobody could prove (for almost 40 years!) that $P \neq NP$

(one of the most famous open problems in computer science)

NP-complete: “hard” problems

Thus, once a computational problem has been proved to be NP-complete it is regarded as being computationally hard (as no algorithm faster than exponential is known for it nor is likely to be invented soon)

In such case, the following alternatives (to look for a polynomial algorithm) can be considered:

- exponential algorithms
- special cases
- approximation algorithms
- probabilistic algorithms
- heuristics

Optimisation Problem

An **instance** of an optimisation problem: a pair (F, c), where F is interpreted as a set of *feasible* solutions and c is the *cost* function: $c : F \to Q$.

The task in *minimisation* is to find $f \in F$ such that $c(f)$ is minimum possible in F. f is called *global optimal solution* to the instance F[1]

An **optimisation problem** is a set I of instances of an optimisation problem.

[1]Another variant is a *maximisation* problem, when we look for a feasible solution f with maximum possible value of $c(f)$ – in this case we can call c a *profit* function

Example of an optimisation problem: SHORTEST-PATH

Instance: a directed graph G=(V,A), two vertices $u, v \in V$ (source and target)

Feasible set: a set of all paths in G that start in u and end in v

Cost function: for a feasible solution f (a path from u to v) it is its length

SHORTEST-PATH **optimisation problem**: the set of all possible graphs and pairs of its vertices

This is a minimisation problem.

(is this problem hard or easy?)

Decision Problems

The solution is in the binary form: “yes” or “no”

Example: is the given boolean formula satisfiable?

An optimisation problem can be usually transformed into its **decision version** that is not harder.

For example, an instance of the SHORTEST-PATH optimisation problem with additional parameter $k \in N$ can be viewed as a decision problem: “is there a path from u to v in G of the length not exceeding k?”

(the problem is not harder since solution to the optimisation problem automatically solves the decision version, but not the opposite, in general)

Encoding of Problems

To be solved by a computer, an abstract problem should be first **encoded** into binary form.

Let Q be an abstract problem represented by the set of instances I

encoding: $e : I \rightarrow \{0,1\}^*$
(* - “Kleene star”)

Encoding transforms an abstract problem into a **concrete problem**, denoted as $e(Q)$

We say that an algorithm solves a concrete problem in time $O(T(n))$ iff for each instance i of size $|i| = n$ (in bits) it finds a solution in time $O(T(n))$.

Complexity class P

Definition

A complexity class P is the set of concrete *decision* problems for which there exist algorithms that solve them with complexity upper-bounded by a polynomial of n (the length of the concrete problem), i.e. the complexity is $O(n^k)$, for $n = |i|$ problem length, k - a positive constant (for each problem).

Notice that a decision problem solvable, for example, with a $\Theta(nlogn)$ complexity (n - size of encoded, concrete problem instance) is in P (even if "$n\ logn$" is *not* a polynomial of n itself, but is upper-bounded by such)

Remark on encoding's "compactness"

One can observe that the fact whether a concrete version of an abstract problem is in P class depends on the "compactness" of encoding.

It is assumed that encoding is "reasonably" compact. In particular, it is assumed that binary encoding is used for numbers (and that all the numbers are rational) which results in $\lceil log_2 n \rceil$ number of bits for a value n. [2]

Notice that unary encoding (which is definitely not a compact one) can make some complex problems look as polynomially solvable due to "expensive" encoding.

[2] It does not matter whether a binary positional system is used or another, since all the logarithms are asymptotically equivalent

Polynomial Relatedness of Encodings

Definition

A function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is **polynomial-time computable** iff there exists an algorithm that for any input $x \in \{0,1\}^*$ produces output $f(x)$ in the polynomially bounded time complexity $O(p(|x|))$ ($p()$ is a polynomial, $|x|$ is the size of concrete instance of the problem)

Given a set I of problem instances, two encodings e_1, e_2 are **polynomially related** iff there exist two functions f_{12}, f_{21} that are polynomial-time computable such that $\forall i \in I\ f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i))$ (and strings not being instances are mapped to such in both directions)

Lemma

Lemma:
Assume e_1, e_2 are two polynomially related encodings of an abstract problem Q. Then, $e_1(Q) \in P \Leftrightarrow e_2(Q) \in P$
(i.e. membership in P class is independent on polynomially related encodings)

Proof:
($\Leftarrow$) Assume that $e_1(Q)$ can be solved with $O(n^k)$ time for a constant k and that the encoding $e_1(i)$ can be computed from $e_2(i)$ with $O(n^c)$ time for a constant c, where $n = |e_2(i)|$. To solve encoded instance $e_2(i)$ of the problem $e_2(Q)$ it suffices to first compute encoding $e_1(i)$ (that takes time $O(n^c)$) and then compute the solution on the output. Its size $|e_1(i)| = O(n^c)$ since output cannot be (asymptotically) bigger than running time. Thus the total complexity will be $O(|e_1(i)|^k) = O(n^{ck})$ a polynomial of n (q.e.d.)

The proof of the other direction is symmetric.

"Standard" Encoding

A bit informally, we will assume some "standard" encoding of basic objects such as rational numbers[3], sets, graphs, etc. that are "reasonable" (e.g. unary encoding of numbers is not reasonable in this sense)

More precisely, in the context of the lemma, we will assume that encoding for all the numbers will be polynomially related to binary encoding (notice: decimal encoding is such), for a set – to comma-separated list of elements, etc.

Such "standard" encoding will be denoted by $\langle . \rangle$ symbols (e.g. $\langle G \rangle$ for a graph G).

Now, we can talk *directly* about abstract problems without reference to any particular encoding.

[3]We assume all the numbers used are rational, i.e. of finite precision

Basic Concepts of Formal Languages

P class and other important concepts can be defined with help of *formal languages*.

Σ: a finite **alphabet** of symbols

A **language** L over Σ is any arbitrary subset of strings of symbols from alphabet.

example: $\Sigma = \{0, 1\}$, $L = \{10, 11, 101, 111, 1011, ...\}$ (prime numbers in binary)

empty string ϵ, Σ^* all finite strings over alphabet

Operations on languages: union,intersection,complement ($\overline{L} = \Sigma^* \setminus L$), concatenation $L_1 L_2$ (or power: L^k), closure (Kleene star): $L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup ...$

Problems as languages

Each abstract decision problem is characterised by all the “yes” instances.

Definition

A (concrete) decision problem Q, is represented by language L over alphabet $\Sigma = \{0, 1\}$, such that:

$L = \{x \in \Sigma^* : \text{x is encoding of a “yes” instance of Q}\}$

Example: (PRIME) $L = \{\langle p \rangle : \text{p is a prime number}\}$

Decision problems and algorithms

We say that algorithm A **accepts** a string $x \in \{0,1\}^*$ if its output $A(x)$ is 1 ("yes").

The language L **accepted** by an algorithm A is the set of all strings:
$L = \{x \in \{0,1\}^* : A(x) = 1\}$

Similarly, x is **rejected** by algorithm A iff $A(x) = 0$.

Even if a language L is accepted be an algorithm, it may not stop (for example) for some $x \notin L$.

If an algorithm A accepts each $x \in L$ and rejects each $x \notin L$ we say L is **decided** by A

Complexity class

A **complexity class** can be viewed as a set of languages decided by a class of algorithms with specified **complexity measure** (a formal definition of a complexity class is a bit more complicated)

For example (the definition of P class):

$P = \{L \subseteq \{0,1\}^* : \text{L is decided by some polynomial algorithm}\}$

Theorem

$P = \{L : \text{L is } \textbf{accepted} \text{ by a polynomial-time algorithm}\}$ (notice that definition of P used the word "decided" instead)

Draft of Proof (after Cormen et al. "Introduction to algorithms"):(We should show that if L is accepted by a polynomial-time algorithm then it is also decided by some polynomial-time algorithm. We will use so-called "simulation" approach. Notice that the proof is **non-constructive** – it only proves the existence of some object (algorithm) however does not show how to find (construct) it).

Assume that language L is accepted by a polynomial time algorithm. Thus, there exist positive integer constants c and k such that A accepts L after at most cn^k steps. Now, imagine an algorithm A' that, for any input x, simulates cn^k steps of A and then checks whether A has accepted L. If yes, it also accepts x, if not yet, it rejects x. The simulation overhead can be implemented so that A' is still polynomial. We explained an existence of a polynomial-time algorithm A' that decides L.

Example: Hamiltonian Cycle Problem

We call a graph G=(V,E) **hamiltonian** iff there exists a *simple* (no vertex can be used more than once) cycle that uses all vertices in V (a hamiltonian cycle).

Now, we can consider a decision problem:
$HAM - CYCLE = \{\langle G \rangle : \text{graph G is hamiltonian}\}$

There is no polynomial-time algorithm known to solve this problem in general. (i.e. all the known algorithms have exponential or higher complexity wrt input size, e.g. checking all n! potentially possible permutations of vertices)

The idea of **certificate**

However, if there is provided a “solution” (a path in G) to an instance of HAM-CYCLE the time to **verify** whether it is a correct hamiltonian cycle is polynomial-time solvable.

In general, verification of a provided solution to a problem can be much faster that finding it.

Such a provided correct “solution” is called a **certificate** (it is quick to verify, but may be computationally hard to find it)

Verification Algorithms

A **verification algorithm** A takes two arguments: an input string x (e.g. binary encoding of a problem instance) and a **certificate** y

A **verifies** x if, there exists certificate y such that $A(x, y) = 1$

A **language verified** by a verification algorithm A:

$$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* A(x, y) = 1\}$$

Definition of NP complexity class

Definition

The *NP* **complexity class** is the class of languages (decision problems) that can be verified by a polynomial-time algorithms.

Equivalently, language (decision problem) L is in NP iff there exists a two-input polynomial-time verification algorithm A and a positive constant c such that:
$L = \{x \in \{0,1\}^* : \exists y(\text{certificate})|y| = O(|x|^c) \wedge A(x,y) = 1\}$

example: HAM-CYCLE is in NP

Remark: *NP* stands for "non-deterministic polynomial", which comes from another, definition of *NP* via so-called *non-deterministic Turing machines*. The above definition is equivalent, and perhaps easier to understand.

Observation: $P \subseteq NP$

If a decision problem (language) is in P it is also in NP.

When a decision problem is in P class its solution (“yes”/”no”) can be found by a polynomial time algorithm A. Why is it in NP too? If there is provided a certificate y, it can be simply ignored and the language is accepted only if A accepts it (in polynomial time). Thus we constructed a polynomial-time verification algorithm for it.

Thus, $P \subseteq NP$

The most Famous Problem in Computer Science

It is an open problem whether $P \neq NP$ since 1971.

Intuitively, P is a class of “easily solvable” decision problems and NP is a class of “easily verifiable” problems.

Most people believe $P \neq NP$ but nobody could prove (or disprove) it. There is a quite strong evidence to support this hypothesis (the existence of “NP-complete class” - next slides)

But, there is a $1.000.000 prize awaiting for the first person providing the proof (founded by the “Clay Mathematics Institute” (Cambridge, US), even if some notable researchers claim that, this problem perhaps lies out of reach for currently known tools in mathematics.

co-NP class

co-NP class is the class of languages (decision problems) L with the property $\bar{L} \in NP$ (that is its complement belongs to NP).

In other words L is in co-NP if it is easy to verify a **negative certificate**.

E.g. PRIME is in co-NP (why?)
(a factorisation of a number is a polynomially verifiable certificate that a number is **not a prime**)

It is unknown whether $NP \neq co - NP$

Of course $P \subseteq co - NP$ (for reasons analogous for the NP case)

Reductions

Intuitively, a problem Q can be **reduced** to another problem Q' if any instance of Q can be "translated" to an instance of Q' so that the solution of the latter provides a solution to the former one.

In other words, Q is "not harder" than Q'.

More formally, language L_1 is **polynomial-time reducible** to a language L_2, denoted as $L_1 \leq_P L_2$, if there exists polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that:
$\forall x \in \{0,1\}^* \, x \in L_1 \Leftrightarrow f(x) \in L_2$

such f is called a **reduction function** and a polynomial-time algorithm F that computes it a **reduction algorithm**.

Lemma

If $L_1 \leq_P L_2$ then $L_2 \in P$ implies $L_1 \in P$

Polynomial reductions provide a formal tool for showing that one problem is “at least as hard as” another problem within a polynomial-time factor.

(for short proof see e.g. Cormen p.1068, 3rd edition)

NP-complete class (NPC)

Definition

A language $L \subseteq \{0,1\}^*$ is **NP-complete** iff:

1. $L \in NP$ and
2. $L' \leq_P L$ for any language $L' \in NP$

Thus, $NP-complete$ is the class of the "hardest problems in NP" (because all problems in NP can be translated to them)

If a language satisfies the property 2. (but not necessarily 1.) it is called **NP-hard**

Theorem

If any NP-complete problem is polynomial-time solvable then $P = NP$. Equivalently, if any NP-complete problem is not polynomial-time solvable then no NP-complete problem is polynomial-time solvable

Proof: Assume $L \in P$ and $L \in NPC$. Thus, for any $L' \in NP$ it holds that $L' \leq_P L$ (NPC definition, property 2) but also, by the last lemma, $L' \in P$ what proves the first part. The other part, by contraposition.(quod erat demonstrandum)

The above theorem is a strong evidence for $P \neq NP$ hypothesis.

Boolean Satisfiability (SAT) - the first known NP-complete problem in history

The first problem proved to be in NP-complete class was "Boolean Satisfiability" (SAT) by Stephen Cook, who introduced the concept ("The complexity of theorem-proving procedures", Proceedings of 3rd ACM STOC, pp. 151–158, 1971)

SAT: is a given boolean formula *satisfiable*?

(does there exist an assignment of boolean values to the variables such that the value of the whole is "true"? (truth assignment))

Historical importance of SAT

More precisely, Cook showed that any (decision) problem in NP class can be reduced in polynomial time to SAT by, so called, *deterministic Turing Machine* [4]

A bit simpler prove can be found at: Garey, Michael R.; David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman. ISBN 0716710455.

As a consequence, if there exists any polynomial algorithm for SAT there exist polynomial algorithms for all problems in NP.

[4] Turing Machine is an important theoretical computation model, that will be not discussed here, and which is, informally speaking, polynomially equivalent to current computers

NP-completeness of CIRCUIT-SAT

CIRCUIT-SAT: given a boolean circuit answer if there exists a truth assignment for it.

In their handbook “Introduction to Algorithms”, Cormen et al. present a draft of a *direct* proof of NP-completeness of CIRCUIT-SAT, without refering to the concept of Turing Machines.

Proving NP-completeness, by definition

By definition, to prove NP-completeness of some problem L, we need to show that **every** problem in NP can be reduced (in polynomial time) to L (and, that L is in NP).

In particular, this was necessary for the (historically) first NP-complete problem: SAT.

However, if we already have at least one NP-complete problem at hand, the technique is simpler

Technique for proving NP-completeness

Lemma: If $L' \in NPC$ and L is a language such that $L' \leq_P L$ than L is NP-hard. In addition, if $L \in NP$ than $L \in NPC$.

Thus, it suffices to show reduction of a problem L to a known NP-complete problem to show it is NP-hard.

Proof: Because $L' \in NPC$, we have $L'' \leq_P L'$ for all $L'' \in NP$. Thus, $L' \leq_P L$ and, by transitivity of $\leq_P$ (exercise) we obtain $L'' \leq_P L$, i.e. L is NP-hard. If, additionally, $L \in NP$, by definition $L \in NPC$.

Notice, that this technique could not be used for proving the NP-completeness of the “first” known NP-complete problem. This makes the Cook’s theorem additionally important.

The method to prove NP-completeness

To prove that L is NP-complete:

1. prove that $L \in NP$
2. choose a known NP-complete problem L'
3. describe polynomial-time algorithm that maps every instance $x \in \{0,1\}^*$ of L' to an instance of L
4. prove that instance $x \in L'$ is “positive” if and only if its mapping in L is positive

This is simpler than proving NP-completeness directly from definition.

NPC class is growing

Following the Cook's innvetion (that SAT$\in NPC$) in 1971, next year in 1972 Richard Karp published his famous list of **21 NP-complete combinatorial problems**. (Both Cook and Karp subsequently received Turing Award for their achievements)

Currently, there are **many thousand** problems proved to be NP-complete.

Example: reduction from CIRCUIT-SAT to SAT

After proving that CIRCUIT-SAT $\in NPC$ (see Cormen et al. for a draft), we can illustrate the reduction technique on SAT.

SAT is in NP: enough to explain that verification algorithm works in time that is polynomial function of the problem size

CIRCUIT-SAT can be encoded as a graph G: nodes – logical gates, arcs – wires connecting the gates, and then the graph can be encoded in its standard encoding $< G >$ with adjacency lists. It is a standard encoding of CIRCUIT-SAT.

CIRCUIT-SAT$\in NPC$, cont.

The main idea is to repesent each logical gate in the circuit as a small logical clause of as follows:

Each wire is represented by a logical variable.

NOT gate, with an input (wire) x and output (wire) y is represented as the following logical clause: $y \leftrightarrow \neg x$

similarly, AND gate, with two inputs x_1, x_2 and output y: as $y \leftrightarrow x_1 \wedge x_2$,

analogously, OR gate: $y \leftrightarrow x_1 \vee x_2$.

Finally, the whole circuit, as the conjuction of the output wire variable and all the “gate” clauses.

CNF - Conjuctive Normal Form

Consider a boolean formula containing some (boolean) variables

literal: a variable or its negation

OR-clause: any number of literals joined by the OR operator
example of OR-clause: $x_1 \lor \neg x_2 \lor x_4 \lor x_6$

CNF: any number of OR-clauses joined by the AND operator
example of CNF: $(x_1 \lor \neg x_2 \lor x_4) \land (x_3 \lor \neg x_6)$

3-CNF: CNF such that each OR-clause consists of *exactly 3 distinct literals*
example of 3-CNF: $(x_1 \lor \neg x_2 \lor x_4) \land (x_3 \lor \neg x_6 \lor x_1)$

3-CNF-SAT is NP-complete (reduction from SAT)

3-CNF-SAT problem: is a given 3-CNF formula satisfiable?

It is NP-complete, can be reduced from SAT(i.e. given arbitrary boolean formula ϕ transform it to 3-CNF that is satisfiable only if the original is satisfiable)

1. using parentheses transform ϕ to its binary parse tree and treat it as a circuit (leaves are inputs, the root is output) and transform to a conjuntion of clauses ϕ' as in reduction from CIRCUIT-SAT to SAT
2. translate ϕ' to CNF ϕ'' by negating DNF obtained from ϕ' by using the disjunction of all false cases. Notice: each clause of CNF ϕ'' has at most 3 literals (due to binarity of the parse tree)
3. translate CNF ϕ'' to 3-CNF ϕ''' by substituting each clause with exactly 2 distinct literals $(l_1 \vee l_2)$ with equivalent $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ and each clause with exactly 1 distinct literal l with $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ (for any assignment of p, q one of the 4 clauses is equivalent to l and other evaluate to 1)

DNF

DNF: disjunctive normal form (many AND-clauses joined with OR)
example: $(x_1 \wedge \neg x_2 \wedge x_4) \vee (x_3 \wedge \neg x_6)$

DNF-SAT problem: is a given DNF formula satisfiable?

This problem is easily solvable in polynomial time

Explanation: a DNF formula is not satisfiable only if all its AND-clauses are not satisfiable, which can happen only if all AND clauses contain conjuction of a variable and its negation - easy to check in polynomial (even linear) time.

Also, 2-CNF-SAT is polynomial-time solvable (hint: $x \vee y$ is equivalent to $\neg x \vee y$ and this can be efficiently solved)

More examples of NP-complete problems

- CLIQUE
- VERTEX COVER
- INDEPENDENT-SET
- SET-COVER
- SET-PACKING
- HAM-CYCLE
- TSP
- SUBSET-SUM

CLIQUE problem

CLIQUE optimisation problem: given an undirected graph $G = (V, E)$ find the maximum *clique* in G. (a maximisation problem)
(A *clique* in G is a full subgraph of G)

The *decision-version* of the CLIQUE problem: CLIQUE: given an undirected graph $G = (V, E)$ and $k \in N \setminus \{0\}$ is there a clique of size k in G?

If k is a constant, the brute-force algorithm (checking all the possible k-element subsets of V whether they form a k-clique) formally has the polynomial complexity $\Theta(k^2(n!/k!(n-k)!)$, however if k is close to $|V|/2$ it becomes exponential.

CLIQUE (decision version) is NP-complete

(Reduction from 3-CNF-SAT)

Obviously, CLIQUE$\in$ *NP* (verifying if a given subset of a graph is a clique can be done in polynomial time)

Reduction: given a 3-CNF-SAT instance ϕ consisting of k OR-clauses construct a $3k$-vertex graph G that has a clique iff the corresponding 3-CNF formula is satisfiable.

Idea: For each OR-clause of ϕ create a triple of 3 vertices representing its literals. The edges are added only between any vertex and all consistent[5] vertices in other triples. Now, the literals that evaluate to 1 in the truth-assignment of ϕ constitute a clique of size k in G. On the other hand, any $k - clique$ in G corresponds to a truth-assignment of ϕ. The construction can be done in polynomial time.

[5] two literals are consistent except the situations one is a negation of another

VERTEX-COVER problem

VERTEX-COVER optimisation problem: given a graph $G = (V, E)$ find a minimum *vertex cover* in G. (a minimisation problem)

A vertex cover is such a subset of vertices $V' \subseteq V$ that each edge in E is adjacent with some vertex from V' (is “covered”).

VERTEX-COVER (decision version): given a graph $G = (V, E)$ and a natural positive number k answer if there is a vertex cover in G of size k.

VERTEX-COVER (decision version) is NP-complete

(reduction from CLIQUE)

Whether a given set of vertices ("certificate") is a vertex cover can be easily verified in polynomial time, so VERTEX-COVER$\in NP$

Let's explain that it is also NP-hard:
Given an instance C of k-clique (decision) problem, construct a graph that has a vertex cover if and only if C has a k-element clique.

Idea: An undirected graph $G = (V, E)$ has a k-clique if and only if a complement graph $\bar{G} = (V, \bar{E})$ has a vertex cover of size $|V| - k$

(A *complement* of an undirected graph $G = (V, E)$ has the same set o vertices V and has an edge (u, v), $u, v \in V$ only if $(u, v) \notin E$.)

INDEPENDENT-SET problem

Given a graph $G = (V, E)$, find a maximum subset $V' \subseteq V$ such that no pair of vertices from V' are adjacent in G. (maximisation problem)

INDEPENDENT-SET decision version: Given a graph $G = (V, E)$ and positive natural k answer whether there exists an independent set of size at least k

INDEPENDENT-SET (decision version) is NP-complete

Idea (reduction from VERTEX-COVER): a set of k vertices $V' \subseteq V$ is independent in G if and only if $V \setminus V'$ is a $((|V| - k)$-element) VERTEX-COVER in G.

SET-COVER (decision) problem

Given a set U of n elements and a family $S = \{S_1, ..., S_m\}$ of subsets of U and positive natural k, answer whether there exists a subfamily of S of at most k subsets such that their union is equal to ("covers") U

Possible interpretation: select the minimum set of people that have all desired skills in total, etc.

SET-COVER is NP-complete

Idea (reduction from VERTEX-COVER): given an instance $(G = (V, E), k)$ of VERTEX-COVER we translate it to a SET-COVER instance as follows. Each vertex $v \in V$ corresponds to the set of edges from E that are incident to v, and U is the set of all edges E.

SET-PACKING (decision) problem

Given a set U of n elements and a family $S = \{S_1, ..., S_m\}$ of subsets of U and positive natural k, answer whether there exists a subfamily of S of at least k subsets such that no pair of subsets intersect

Possible interpretation: subsets may correspond to processes that need non-sharable resources from a set U of all resources of the system; can we run at least k processes in parallel?

SET-PACKING is NP-complete

Idea: reduction from INDEPENDENT-SET, analogously as SET-COVER can be reduced from VERTEX-COVER.

HAM-CYCLE problem

A *hamiltonian cycle* in a graph is a cycle that uses each vertex exactly once[6]

HAM-CYCLE (decision) problem: given a graph $G = (V, E)$ answer whether there exists a hamiltonian cycle in it

HAM-CYCLE is NP-complete (it can be reduced from VERTEX-COVER, for example, see Cormen et al. for a proof)

[6] A cycle with no repeated vertices is called a *simple* cycle

Travelling Salesman Problem (TSP)

TSP optimisation problem: Given a graph $G = (V, E)$ with non-negative weights on edges find a hamiltonian cycle $C \subseteq E$ in G with minimum weight (defined as the sum of weights of the edges in C). (minimisation problem)

TSP decision version:
Given a graph $G = (V, E)$ with weights on edges and $k \in N$ answer whether there exists a hamiltonian cycle in G with weight of at most k.

TSP (decision) problem is NP-complete.

Idea (reduction from HAM-CYCLE): given an instance $G = (V, E)$ of HAM-CYCLE, construct an instance of TSP as $G' = (V, E')$ such that $E' = \{(i, j) : i, j \in V \wedge i \neq j\}$ and weight for each edge (i, j) defined as $w(i, j) = \lfloor (i, j) \notin E \rceil$ (assume self loops exist in G' and have weights of 1). The corresponding instance of TSP is then whether there exists a hamiltonian cycle of weight (at most) 0 in G'.

Reduction types

- Karp Reduction: single reference to the reduction target
- Cook Reduction: multiple reference, more general (also called Turing reduction, i.e. a reduction that is polynomial iff the target sub-routine is polynomial)

Karp reduction (showing $Y \leq_P X$):

- prove $X \in NP$
- choose Y that is NPC
- take an arbitrary instance s_Y of Y and show how to construct in polynomial time an instance s_X of X so that:
  - if s_Y is "yes" instance of Y then s_X is "yes" instance of X
  - if s_X is "yes" instance of X then s_Y is "yes" instance of Y

Yet another distinction among reduction types

(after Garey and Johnson's classic textbook: "Computers and Intractability", 1979)

- Restriction (the simplest, similar to the "Karp" type), e.g.:
    - VERTEX-COVER $\leq_P$ HITTING-SET (read as "VERTEX-COVER is a restriction of HITTING-SET")
    - EXACT-3-COVER $\leq_P$ MIN-COVER
- Local Replacement ("medium complexity") e.g.: SAT $\leq_P$ 3-SAT
- Component Design (the most complex) e.g.:
    - 3-SAT $\leq_P$ HAMILTON-CYCLE
    - 3-SAT $\leq_P$ 3-DIM-MATCHING
    - 3-SAT $\leq_P$ 3-COLORING
    - 3-DIM-MATCHING $\leq_P$ SUBSET-SUM

Graph k-Coloring

Assign each vertex a color (1 of k) so that neighbours have different colors
(resource allocation, map coloring, etc.)

is 2-coloring NP-complete? (it is equivalent to checking whether a graph is bi-partite what can be done in polynomial time. How? With BFS in $O(|V| + |E|)$ time

3-coloring is NP-complete (reduction from 3-SAT, for example)

for $k >= 3$ k-coloring is NP-complete (reduction from 3-coloring: original graph + $k - 3$-element clique connected to all original nodes)

Why NP-completeness is useful for practical algorithm-designers?

Assume, we want to solve a new problem. Now assume that somebody proved it to be an NP-complete problem.

Does it make sense to look for a fast algorithm solving it? Not really, because unless $P = NP$ there is no such algorithm.

In this situation it is possibly much better to invest the effort in a different way than looking for a fast exact solution.

Alternatives for dealing with NP-complete problems

For a problem proved to be NP-complete there are the following alternatives for proceeding (since finding an exact fast algorithm is unlikely):

- trying to design exponential-time algorithm that is as fast as possible
- focus on special cases, and find fast algorithms for them
- work towards a fast *approximation* algorithm, that does not solve the problem exactly, but you can prove some bounds on the solution quality
- (if you cannot do the above) use one of the fast *heuristics* to approximately solve the problem, without guarantees on the quality (this is possibly the least "ambitiuous" approach, but sometimes necessary)
- device a *randomised* algorithm, that is *expected* to be fast

Example

Lets consider another decision problem:
PARTITION: A - finite set of n items, each item $a \in A$ has weight $w(a) \in Z^+$.
OUTPUT: Can A be partitioned into 2 parts of identical weights? (i.e. does there exist $A' \subseteq A$, such that $\sum_{a \in A'} w(a) = B/2$, where $B = \sum_{a \in A} w(a)$)

On one hand: PARTITION is NP-complete (3SAT $\leq_P$ 3-DIM-MATCHING $\leq_P$ PARTITION)

On the other hand: There exists an algorithm that solves PARTITION in time $\Theta(nB)$

Does this mean **we have just found the first polynomial-time algorithm for an NP-complete problem!?**

Dynamic-programming algorithm for PARTITION

If B is odd answer no.

Otherwise, for integers $0 < i \leq n$, $0 \leq j \leq B/2$ let $t(i,j)$ be "true"/"false" according to the following statement: "there is a subset of $\{a_1, a_2, ..., a_i\}$ for which the sum of weights is exactly j".

The $t(i,j)$ table is filled row by row, starting with $t(1,j) = \mathit{true}$ iff $j = 0$ or $w(a_1) = j$. For $i > 1$ we set $t(i,j) = \lfloor t(i-1,j) = \mathit{true} \vee (w(a_i) \leq j \wedge t(i-1, j - w(a_i))\rceil$.

The answer is "yes" iff $t(n, B/2) = \mathit{true}$.

Time complexity of this (correct!) algorithm is: $\Theta(nB)$.

Is it a polynomial of the data size?
Not really.

Pseudo-polynomial algorithms

Since the numbers in the task are represented in binary form (“reasonable encoding” assumption), $\Theta(nB)$ can be actually **exponential** function of the data size.

The algorithm is polynomial only if the numerical values in the input are small enough (i.e. polynomial) in the data size

Such algorithms are called **pseudo-polynomial** algorithms.

Additional definitions

Let's consider an instance i of a given problem.

Let $length(i)$ and $max(i)$ specify integer functions (interpreted as "data size" and "maximum numeric value" in the input)

(Two pairs of functions (length,max) and (length',max') are *polynomially related* iff $length(i) \leq p'(length'(i))$, $length'(i) \leq p(length(i))$ and $max(i) \leq q'(max'(i), length'(i))$, $max'(i) \leq q(max(i), length(i))$ for all instances i and some polynomials p, p', q, q'.)

An algorithm is *pseudo-polynomial* iff it is bounded by a polynomial of length() and max().

A problem is called *number problem* if there exists no polynomial p such that $max(i) \leq p(length(i))$ for all instances i.

Strong NP-completeness

We call the problem **strongly NP-complete** if it contains a subproblem that is NP-complete and satisfies polynomial bound on *Max*.

PARTITION is not strongly NP-complete (as there exists a pseudo-polynomial algorithm for it)

Observations:

- If problem is NP-complete and is not a number problem then it cannot be solved by a pseudo-polynomial algorithms *unless* $P \neq NP$
- If a problem is strongly NP-complete, then it cannot be solved by a pseudo-polynomial algorithms *unless* $P \neq NP$

Example: Vertex Cover

Vertex Cover, decision version (VC):

Given a graph $G = (V, E)$, where $|V| = n$ and $k \in N$, is there a vertex cover of size at most k?
(i.e. such a set of vertices $S \subseteq V$, $|S| \leq k$ that each edge $e \in E$ hast at least one end in S).

VC is NP-complete (SAT -> 3-SAT -> IS (independent set) -> VC), so that no polynomial algorithm is likely to exist.

VC, cont

However, if k is fixed and small (e.g. $k = 3$), the method of checking *all*[7] possible subsets of size k has complexity $O(kn \cdot n^k)$ that is *polynomial* (of n).

Notice: even for relatively small values of n, k this polynomial algorithm is impractical: e.g. $n = 1000, k = 10$ would take more than the age of the Universe on a PC.

Interestingly, there exists an *exponential* algorithm that would be faster for small values of n, k !

Observations:
if G has at most k-element vertex cover, then:

- $|E| \leq kd$, where d is maximum degree of a node
- $|E| \leq k(n-1)$

[7] In general, checking all potential solutions is called *brute force* method

Exponential Algorithm for VC

Assume $e = (u, v) \in E$. G has at most k-element vertex cover iff at least one of the graphs $G \setminus \{u\}$ or $G \setminus \{v\}$ [8] has a vertex cover of size at most $k - 1$

```
If |E|=0 then answer ‘‘yes’’, if |E|>kn then answer ‘‘no’’
Else, take any edge e=(u,v)
  recursively check if either G\{u} or G\{v}
     has vertex cover T of size k-1
  if neither has, then answer ‘‘no’’
  else T+{u} or T+{v} is k-element vertex cover of G
```

Time complexity of the above algorithm is $O(2^k kn)$
Thus, for our previous example ($n = 1000, k = 10$) the algorithm will find the solution *very quickly*

[8] all the edges incident to a node are also removed

Time complexity analysis of the algorithm

Explanation: The recursion tree has height of k, thus the number of recursive calls is bounded by 2^{k+1}. Each recursive call (except the leaves) takes at most $O(kn)$ time.

Proof (by induction on k): $T(n, k) = O(2^k kn)$. Assume $c \in N$ is a constant:
$T(n, 1) \leq cn$
$T(n, k) \leq 2T(n, k-1) + ckn$

Assume the thesis is true for $k - 1$, then:

$T(n, k) \leq 2T(n-1, k-1) + ckn \leq 2c \cdot 2^{k-1}(k-1)n + ckn = c2^k kn - c2^k n + ckn \leq c \cdot 2^k kn$

Of course, this algorithm is not practical for higher values of k (as it is exponential in k)

NP-optmisation Problem

NP-optimisation problem Π consists of:

- set of *valid instances*, D_Π, recognisable in polynomial time (assume: all the numbers are rational, and encoded in binary, $|I|$ denotes the size of encoded instance I, in bits).
- each instance $I \in D_\Pi$ has a set of *feasible solutions*, $S_\Pi(I) \neq \emptyset$. Each feasible solution $s \in S_\Pi(I)$ is of length bounded by polynomial of $|I|$. Moreover, there is a polynomial algorithm that given a pair (I, s) decides whether $s \in S_\Pi(I)$
- there is a polynomially computable *objective function* obj_Π which assigns a nonnegative rational number to each pair (I, s) (an instance and its feasible solution).
- Π is specified to be either *minimisation* of *maximisation* problem

NP-optimisation Problems, cont.

Optimal solution of an instance of a minimisation (maximisation) problem is a feasible solution which achieves the minimum (maximum) possible value of the objective function (called also “cost” for minimisation or “profit” for maximisation).

$OPT_{\Pi}(I)$ denotes optimum objective function value for an instance I

Decision version of an NP-optimisation problem I: a pair (I, B), where $B \in Q$ and the decision problem is stated as “does there exist a feasible solution to I of cost $\leq B$, for minimisation problem I” (or, analogously “of profit $\geq B$”, for a maximisation problem)

Extending the definition of NP-hardness for optimisation problems

Decision version can be "reduced" to optimisation version. (i.e. polynomial algorithm for optimisation version can solve the decision version)

NP-optimisation problem can be called NP-hard if its decision version is NP-hard.

Approximation Algorithm

Let Π be a minimisation (maximisation) problem, $\delta : Z^+ \to Q^+$ a function that has values ≥ 1 (≤ 1).

Definition

An algorithm A is a *factor δ approximation algorithm* for Π if, for each instance I, A finds a feasible solution s for I such that: $obj_{\Pi}(I, s) \leq \delta(|I|) \cdot OPT(I)$
(for maximisation:$obj_{\Pi}(I, s) \geq \delta(|I|) \cdot OPT(I)$)

Observation: The closer δ to the value of 1, the better approximation.
Remark: δ can be also a function of some other parameter than length of input instance (|I|).

Example: Approximation of Vertex Cover

Let's consider an optimisation version of the Vertex Cover:

Given a graph $G = (V, E)$ find a subset $V' \subseteq V$ so that any edge $e \in E$ has at least one edge in V' (is "covered") and V' has *minimum possible size*.

This problem is an NP-optimisation problem and it is NP-hard because its decision version is NP-hard (and decision version is "not harder"[9])

Thus, no polynomial-time algorithm that finds optimum is known for this problem.

We will present:
a polynomial time 2-approximation algorithm for Vertex Cover.

[9] i.e. solution to an optimisation version automatically gives a solution for the decision version

2-approximation Algorithm for Vertex Cover

```
while(there are edges in E)
 take any edge e in E
 add both ends of e to the cover C
 remove e and all incident edges from E
```

Number of iterations bounded by $|E|$

Optimal cover must include at least one of the two ends of each selected edge, thus: $|C| \leq 2 \cdot OPT$

Reductions that preserve approximation factor

Usually, polynomial-time reductions map optimal solutions to optimal solutions.

However, it is not necessary, that near-optimal solutions are mapped to such.

All NP-complete problems are “equally hard” (in terms of polynomial reduction).

However, NP-complete problems may differ greatly in terms of “easyness of approximability”

Let’s introduce a formal notion of reductions that preserve approximation factor.

Example

Consider a 100-vertex graph with minimum vertex cover of size 49.

Assume the 2-approximation algorithm for VC finds a 98-node solution.

Since a complement of any vertex cover is an independent set, the algorithm found also an independent set.

What is its approximation ratio?

$(100 - 49)/(100 - 98) = \mathbf{25.5}$

Thus, the straightforward “reduction” from VC to IS does **not** preserve approximation factor.

Factor-preserving reductions

Definition

Let Π_1 and Π_2 be two minimisation problems (if the problems are of maximisation type, the definition is analogous).
An *approximation factor preserving reduction* from Π_1 to Π_2 consists of two polynomial algorithms, f and g such that:

- for any instance I_1 of the problem Π_1, $I_2 = f(I_1)$ is an instance of Π_2 such that $OPT_{\Pi_2}(I_2) \leq OPT_{\Pi_1}(I_1)$
- for any solution t of I_2, $s = g(I_1, t)$ is a solution of I_1 such that $obj_{\Pi_1}(I_1, s) \leq obj_{\Pi_2}(I_2, t)$

Observation: the definition is designed so that an α-approximation algorithm for Π_2 gives an α-approximation algorithm for Π_1

Factor-preserving reductions

Factor-preserving reduction indeed preserves the approximation factor
(i.e. if we have an α-approximation for a problem Π_2 we will have α-approximation for Π_1:

Proof:
Take instance i_1 of problem Π_1, compute instance $i_2 = f(i_1)$ of Π_2 such that $OPT_{\Pi_2}(i_2) \leq OPT_{\Pi_1}(i_1)$.

Take α-approximation solution t of i_2 compute $g(i_1, t) = s$ such that: $obj_{\Pi_1}(i_1, s) \leq obj_{\Pi_2}(i_2, t) \leq \alpha OPT_{\Pi_2}(i_2) \leq \alpha OPT_{\Pi_1}(i_1)$

(the middle inequality is due to the property of being α-approximation, the other two due to the definition of factor-preserving reduction)

Example: TSP with Triangle Inequality

Given full graph $G = (V, E)$ with “distances” on edges, such that: $d(u, v) \leq d(u, w) + d(w, v)$ for any u, v, w. (TSP with triangle inequality is still NP-complete)

- compute T: MST of G
- make a tour T’ around T (each edge twice)
- remove duplicate vertices by shortcuts (call it T'')

MST can be found in polynomial time
$|T| \leq OPT$ (any TSP-tour minus 1 edge is a ST)
$|T'| = 2|T|$ (each edge twice)
$|T''| \leq |T'|$ (triangle inequality)

Thus: $|T''| \leq 2 \cdot OPT$

Better algorithms exist (e.g. Christofides Algorithm: 3/2-approximation)

Constant factor approximation for general TSP?

Triangle inequality assumption allows for a constant factor approximation for TSP. Without this assumption the TSP problem is much "harder" in the following way:

Theorem

For a general TSP optimisation problem (in particular, when triangle inequality does not hold) there is no polynomial-time α-approximation algorithm for any $\alpha \geq 1$, unless $P \neq NP$

Proof

Proof (by contradiction): Assume, there is an α-approximation algorithm for TSP that has polynomial time complexity. We will show that this algorithm can be used to solve Hamiltonian Cycle (HAM-CYCLE) problem in polynomial time. Since HAM-CYCLE is NP-complete, this would lead to a contradiction.

Let $G = (V, E)$ be the graph corresponding to an instance of the HAM-CYCLE problem. Let's consider a *full* graph $G' = (V, E')$ (thus, $E \subseteq E'$) and define weights on its edges as follows: $w(e) = 1$ when $e \in E$, and $w(e) = \alpha \cdot |V| + 1$ otherwise.

Now, notice that the α-approximation algorithm *must* find a tour in G' that corresponds to a hamiltonian cycle in G if such exists, in polynomial time. Otherwise, the cost of found solution would be at least $(\alpha \cdot |V| + 1) + (|V| - 1) = (\alpha + 1)|V|$, while an optimum solution, corresponding to a hamiltonian cycle, would have cost of $|V|$. Thus, the cost of solution found would be more than α times higher than OPT that would contradict the approximation guarantee of the algorithm.

Thus, the α-approximation algorithm for TSP would lead to a polynomial-time algorithm for HAM-CYCLE which would imply $P = NP$.

Literature

- Cormen et al. "Introduction to Algorithms", chapters 34,35 (3rd edition)
- Kleinberg, Tardos "Algorithm Design", chapters 8,10,11
- Garey, Johnson "Computers and Intractability" (1979, difficult to get nowadays)
- Papadimitriou "Computational Complexity" (first chapters) (more advanced textbook)

Thank you for attention