Lecture 3: Computational complexity

MIEFis - Quantum Computation, L. S. Barbosa, 2018-19

Summary.
(1) Notion of growth rate of a function. Examples.

(2) Case study: closure operations.

(3) Computational complexity. The class P of polynomial decidable languages. $\mathsf{P} \neq \mathsf{NP}$?

---

1 Analysis of algorithms

Introduction.

The focus of this lecture is *computational efficiency*, i.e. the attempt to quantify the amount of computational resources required to solve a given problem. The efficiency of an algorithm will be discussed by studying how its number of basic operations scales as the size of the input increases.

A fundamental message is that the efficiency of an algorithm is to a considerable extent much more important than the technology used to execute it.

$\boxed{\text{Example}}$

Integer multiplication by the repeated addition algorithm (to compute $x \cdot y$, just add x to itself $y - 1$ times) or by the usual grade-school algorithm illustrated. For example, multiplying 577 by 423 using repeated addition requires 422 additions, whereas doing it with the grade-school algorithm takes 3 multiplications of a number by a single digit and 3 additions. Even for 11-digit numbers, a pocket calculator running the grade-school algorithm would beat the best current supercomputer running repeated addition.

Growth rate of a function.

$\boxed{\text{Example: transitive closure of } R \subseteq A^2}$

from 'above' R^* is the smallest relations containing R that is transitive and reflexive.

from 'below'

$$R^* \;=\; \{(a, b) \mid a, b \in A \text{ there exists a path from } a \text{ to } b \text{ in } R\}$$

which suggests an algorithm:

Algorithm 1: TC1.

$R^* := \emptyset$;
for $i := 1..n$ **do**
  for *each* i-*tuple* $(b_1, \cdots, b_i) \in A^i$ **do**
    if *it is a path* **then**
      $R^* := R^* \cup \{b_1, b_i\}$
  end
end

One may inquiry this algorithm on

- *correctness* vs *termination*
- ... but when?

Growth rate for functions

$$\mathcal{O}(f) \ = \ \{g \in \mathbb{N}^{\mathbb{N}} \mid \exists_{c,d \in \mathbb{N}+}. \ \forall_n. \ g(n) \leq c.f(n) + d\}$$

Stating that

$$h \in \mathcal{O}(f)$$

means that h *is no faster than* f.

Example: $p(n) = 31n^2 + 17n + 3$

Clearly $p(n) \leq 48n^2 + 3$, because $n^2 \geq n$. Thus $f \in \mathcal{O}(-^2)$ with constants 48 and 3. However, $-^2 \in \mathcal{O}(p)$ with constants 1 and 0.

Exercise 1

Define $f \sim g$ iff $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f)$. Prove $\sim$ is an equivalence relation.

Theorem

For any polynomial $p(n) = c_k n^k + \cdots + c_1 n + c_0$, $p \in \mathcal{O}(-^k)$ with constants $\sum_{1 \leq i \leq k} c_i$ and c_0.

$\boxed{\text{Theorem}}$

Any two polynomials p and q with the same degree verify $p \sim q$.

$\boxed{\text{Theorem}}$

The growth rate of function 2^n is higher than the one of an arbitrary polynomial.

Proof.
We want to show that

$$n^i \in \mathcal{O}(2^n) \text{ i.e. } n^i \leq c2^n + d \tag{1}$$

Let $c = (2i)^i$ and $d = (i^2)^i$, and consider two cases:

- $n \leq i^2 \Rightarrow n^i \leq c2^n + d,$ because $n^i \leq d$
- $n > i^2 \Rightarrow n^i \leq c2^n + d,$ because $n^i \leq c2^n$

Note that $n^i \leq (iq+i)^i = i^i(q+1)^i$, for q the integer quotient of n by i (i.e. $iq \leq n \leq i(q+1)$). Now,

$$\begin{array}{ll} & i^i(q+1)^i \\ \leq & \qquad \{\, n \leq 2^n \} \\ & i^i(2^{q+1})^i \\ \leq & \qquad \{\text{ definition of } c\} \\ & c2^{qi} \\ \leq & \qquad \{\text{ definition of } q\} \\ & c2^n \end{array}$$

Observe now that if a polynomial had the same growth rate than $-^2$, then any polynomial of a higher degree would have the same rate (because we've just proved that no polynomial grows as fast as $-^2$). But this leads to a contradiction because, as shown above, polynomials of different degrees have different rates of growth.

$\square$

Clearly, 2^n has a higher rate of grow than any polynomial. Other exponential functions — for example, 27^n, n^n, $n!$, 2^{n^2} or 2^{2^n} — have even higher rates of growth.

2 Case study: Closure algorithms

Computing R^*.

Algorithm 2: TC2.

$R^* := \emptyset$;
for $i := 1..n$ **do**
  for *each* i*-tuple* $(b_1, \cdots, b_i) \in A^i$ **do**
    if *it is a path* **then**
      $R^* := R^* \cup \{b_1, b_i\}$
  end
end

The algorithm examines each sequence $(b_1, \cdots, b_i)$; if this is a path add to the solution. Thus, the total number of basic operations (test and add) is

$$n(1 + n + n^2 + \cdots + n^n)$$

i.e. in each of the n iterations look for paths of length up to n. Therefore, TC1 $\in \mathcal{O}(n^{n+1})$.

Algorithm 3: TC3.

$R^* := R \cup \{(a, a) \mid a \in A\}$;
while $\exists_{a_i, a_j, a_k \in A}.\ (a_i, a_j), (a_j, a_k) \in R^*,\ (a_i, a_k) \notin R^*$ **do**
  $R^* := R^* \cup \{(a_i, a_k)\}$
end

- In each iteration one pair (if any) is added. Thus, the maximum number of additions corresponds to the maximum number of pairs available, i.e. n^2.
- In each iteration the algorithm searches for n^3 triples.

Therefore, TC3 $\in \mathcal{O}(n^2 \times n^3) = \mathcal{O}(n^5)$.

Exercise 2

The algorithm TC3 repeatedly searches for violations of the transitivity property. However, each triple must be checked again and again since the introduction of a new pair may entail new violations in triples that have already been checked. A better algorithm of $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^3)$ can be obtained by imposing an order to the triples so that a new pair added does not violate the transitivity condition established for triples already considered. In the algorithm below, TC4, triples are ordered by the middle index (in increasing order). Explain why the algorithm works and its growth rate.

Algorithm 4: TC4.

$R^* := R \cup \{(a, a) \mid a \in A\}$;
for $j = 1, 2, \cdots, n$ **do**
  for $i = 1, 2, \cdots, n$ *and* $k = 1, 2, \cdots, n$ **do**
    if $(a_i, a_j), (a_j, a_k) \in R^*$ *but* $(a_i, a_k) \notin R^*$ **then**
      $R^* := R^* \cup \{(a_i, a_k)\}$
  end
end

Closure problems.

A subset $C \subseteq A$ is *closed* for a relation $R \subseteq A^{n+1}$ if

$$b_{n+1} \in C \quad \Leftarrow \quad b_1, \cdots, b_n \in C \ \wedge \ (b_1, \cdots, b_n, b_{n+1}) \in R$$

e.g.

- $\mathbb{N}$ is closed for $+$
- the set of ancestors is closed for the relation *parent-of*
- any set is closed for $\subseteq$

Closure property: *The set* C *is closed under relations* $R_1, \cdots, R_m$

cf, the usual construction *the smallest set that contains* A *and has property* ϕ. But note that not all properties guarantees the existence of a smallest set satisfying ϕ. However, Theorem

If ϕ is a closure property defined by relations $R_1, \cdots, R_m$ on a set A and $B \subseteq A$, then there exists the smallest set C st $B \subseteq C$ and C has property ϕ.

Proof.
Let ϕ be defined by a relations $R_1, \cdots, R_m$ and S denote the set of subsets of A containing B and closed for each R_i. Clearly $S \neq \emptyset$ (why?). Then, define $C = \bigcap S$ (which is well defined because S is non empty). Then,

- $B \subseteq C$, by construction.
- C is closed under all relations $R_1, \cdots, R_m$. To see this, suppose $a_1, \cdots, a_{n-1} \in C$ and $(a_1, \cdots, a_{n-1}, a_n) \in R$. All sets in S contain $a_1, \cdots, a_{n-1}$ and because all of them are closed, all have a_n. Thus, $a_n \in C$.
- C is minimal: no strict subset C' of C exists (otherwise $C' \in S$ and $C \subseteq C'$).

□

Exercise 3

Set C in the theorem above is the *closure* of B under relations $R_1, \cdots, R_m$. Determine the closure of the singleton set containing yourself under the relation *parent of*. Similarly, determine the closure of set $\{0, 1\}$ under addition and the closure of the set of natural numbers under subtraction. Note that R^*, for a relation R is the closure of R under transitivity and reflexivity.

Theorem

Any closure property over a finite set can be computed in polynomial time.

Proof.

Algorithm 5: Computing a generic closure.

$C^\circ := C$;
while $\exists_{1 \leq i \leq k \textit{ and } r_i \textit{ elements } a_{j1} \cdots a_{j_{r_{i-1}}} \in C^\circ \textit{ and } a_{j_{r_i}} \in D \setminus C^\circ}$. $(a_{j1} \cdots a_{j_{r_i}}) \in R_i$ **do**
  $C^\circ := C^\circ \cup \{a_{j_{r_i}}\}$
end

Thus, the algorithm is $\mathcal{O}(n^{r+1})$ where n is the cardinal of D and r is the greatest arity of all relations considered.

□

Theorem

Any algorithm in polynomial time can be rendered as the computation of a closure over a set for a set of relations.

3 Computability *vs* complexity

The Travelling Salesman Problem.

Given a map with n cities and distances in Km, produce an itinerary that minimizes the total distance travelled.

- Clearly solvable (e.g. systematic examination of all itineraries)
- but unsolvable in any practical sense by current computers: too many itineraries ($(n-1)!$) to be explored. Notice that a $(n-1)!$ algorithm goes faster that 2^n. For 40 cities the number of itineraries is enormous: $39!$. Even if 10^{15} of them could be inspected per second (a value our of reach of current supercomputers) the required time for completing the calculation would be several billion lifetimes of the universe.

What is a *practically feasible algorithm*?

... should run for a number of steps bounded by a *polynomial* in the length of the input, i.e. have a polynomial rate of growth.

Polynomially decidable languages.

A language is *polynomially decidable* if there is a polynomially bounded Turing machine that decides it, i.e. a Turing machine which always halts after at most $p(n)$, where $p(n)$ is a polynomial and n is the length of the input.

The class P of such languages is the quantitative analog of the class of recursive languages. As the latter it is closed under complement, union, intersection, concatenations and Kleene star. But, on the other hand, not all recursive languages are polynomially decidable.

$\boxed{\text{Theorem}}$

$S \notin P$, where

$$S = \{\text{"}M\text{""}w\text{"} \mid M \text{ accepts input } w \text{ after at most } 2^{|w|} \text{ steps}\}$$

<u>Proof.</u>

If $S \in P$, language

$$S' = \{\text{"}M\text{"} \mid M \text{ accepts input "}M\text{" after at most } 2^{|\text{"}M\text{"}|} \text{ steps}\}$$

and its complement are also in P. This means that there exists a polynomially bounded Turing machine B which accepts all descriptions of Turing machines that fail to accept their own description in 2^n steps, where n is the length of the description, and halts in $p(n)$ steps for a polynomial $p(n)$.

Does B accept its own description "B"?

- If YES then B fails to accept "B" within $2^{|"B"|}$ steps. However, B halts in |"B"| steps (because, by assumption, the complement of S' is in P. This means that B halts much before $2^{|"B"|}$. Thus it should reject "B", which leads to a contradiction. Note that there is always an integer n_0 such that $p(n) \le 2^n$ for all $n \ge n_0$, and we may safely assume $|"B"| \ge n_0$.
- If NO a similar argument also leads to contradiction.

Problems.

Reachability. Given two nodes of a finite graph decide if there is a path connecting them.

Is a variant of the reflexive-transitive closure problem. Can be solved by computing this closure in time $\mathcal{O}(n^3)$ and inspect the result.

A *problem* is a set of inputs, typically infinite, with a Boolean question to be asked of each input. A problem needs to be encoded as a language problem so that its complexity can be analysed in a common setting. For example, the *Reachability* problem can be reduced to a decision problem for the language

$$R = \{K(G)s(i)s(j) \mid \text{there is a path in } G \text{ connecting nodes } n_i \text{ to } n_j\}$$

where K and s are suitable binary encoding functions for graphs and integers.

Other problems

Euler Cycle. Given a graph is there a closed path in it that uses each edge exactly once?

Note that the path can go many times through the same node (or even not at all if there are isolated nodes). It can be proved that the necessary condition on a graph to have such a path is that i) all nodes have equal numbers of incoming and outgoing edges, and ii) for each pair of nodes, neither of which isolated, there is a path connecting them. So, clearly the corresponding language

$$G = \{K(G) \mid G \text{ has an Euler cycle}\}$$

where $K(G)$ is some encoding of graphs as strings, is in n P.

Hamilton Cycle. Given a graph is there a cycle that passes through each node exactly once?

No polynomial algorithm is known. Of course the trivial one (generate all paths and choose) is not polynomial.

Equivalence of Finite Automata. Given two deterministic automata, determine whether they recognise the same language?

The problem is polynomial, as it is the variant in which only regular expressions are considered. However, one cannot conclude about the latter just by reducing to the former: actually, the generation of a finite automaton from a regular expression may increase exponentially the number of states.

Integer Partition. Given a set of n nonnegative integers represented in binary, is there a subset S of the original set such that $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?

The algorithm is $\mathcal{O}(nV)$ where V is the sum of all numbers in the original set divided by 2. In spite of its polynomial appearance, the problem is not polynomial in the length of the input. The reason is that the integers are encoded in binary: if all integers are about $\mathbf{2}^n$, then S is close to $\mathbf{2}^n \times \frac{n}{2}$.

Satisfiability. Is a Boolean formula in conjunctive normal form satisfiable?

No polynomial algorithm is known. However, if reduced to formulas with a maximum of two literals, it becomes polynomial.

The party problem. Given a list of acquaintances anda list of all pairs among them who do not get along, find the largest set of acquaintances you can invite to a dinner party such that every two invitees get along with one another

This is equivalent to the *independent set* problem mentioned below.

Optimisation problems

Require to find the best among many possible solutions, according to some cost function. The trick to transform optimisation into language problems is to fix each input with a *bound* on the cost function. For example, the *Traveling Salesman* problem can be rephrased as

Given an integer $n \geq 2$*, a* $n \times n$ *distance matrix*, and an integer $b \geq 0$, find a permutation of n such that its cost is less or equal to b (which, to build up intutition, may be regarded as a budget).

Independent Set. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \geq K$ such that for any two vertices in s there is no edge connecting them?

Clique. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \geq K$ such that for all vertices in s there is an edge connecting each pair?

Node Cover. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \leq K$ such that s covers all edges of the graph? (cf, minimising guards in a museum).

Note that a set of nodes covers an edge if it contains at least one endpoint of the edge.

No polynomial algorithms are known.

The class NP.

Most interesting problems mentioned above for which no polynomial algorithm exists — *Traveling Salesman, Satisfiability, Independent Set, Integer Partition*, etc., can be solved by *polynomially bounded nondeterministic Turing machines*. All computations of such machines do not continue for more than polynomially many steps.

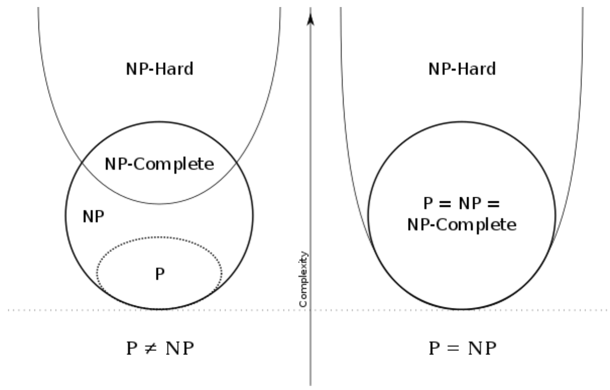

Figure 1: (from Wikipedia)

This defines the class NP (of *nondeterministic* polynomial languages), i.e. of languages that can be decided by a polynomially bounded nondeterministic Turing machine. Note the meaning of *decision* in this context. For a nondeterministic Turing machine deciding a language is required that all the computations of the machine must reject an input not in the language, whereas an input from the language must be accepted by at least one computation.

Although determinism and nondeterminism in the definition of Turing machines do not interfere on their expressiveness in what concerns decidability, separating determinism form nondeterminism at the polynomial level (the $\mathsf{P} \neq \mathsf{NP}$ conjecture), remains unsolved.

Another way to put it is to say that the complexity class NP that aims to capture the set of problems whose solutions can be efficiently *verified*. By contrast, the class P contains decision problems that can be efficiently *solved*. The P versu5 NP question asks whether or not the two classes are the same.

Actually NP does not refer to non-polynomial, but to nondeterminism.

The problems described above share an important completeness property: All problems in NP can be reduced to them (just as all recursively enumerable languages reducing to the halting problem). Such problems are called NP-complete, and they are the hardest of all NP problems. The number of real-life problems that are known to be NP-complete is enormous. Each of them has a polynomial algorithm iff $\mathsf{P} = \mathsf{NP}$. NP-complete problems are in NP, the set of all decision problems whose solutions can be verified in polynomial time; or, alternatively, solved in polynomial time on a non-deterministic Turing machine. A problem in NP is NP-complete if every other problem in NP can be transformed (or reduced) into p in polynomial time.

NP-hard problems live beyond NP, i.e. they are at least as hard as the hardest problems in NP. A problem is NP-hard if every problem in NP can be reduced in polynomial time to it. There are problem which are NP-hard but not NP-complete. For example the *halting problem* which fails to be decidable.

References.
My preference on complexity theory is Papadimitriou's wonderful book [3]; reference [2] provides an interesting alternative. S. Arora and B. Barak book [1] is a more recent textbook covering recent achievements in complexity theory (including challenges from quantum computation) and putting them in the context of the classical results. Worth reading.

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] D. Z. Du and K. I. Ko. *Theory of Computational Complexity*. Addison-Wesley, 2000.

[3] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.