Polynomial time solvable problems [1]

Camilo Rocha & Miguel Romero

October 3, 2019

Pontificia Universidad Javeriana de Cali

Análisis y Diseño de Algoritmos

Class P

The **class P** consists of those problems that are solvable in **polynomial time**. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem. Most of the problems examined in this course are in P.

Class P

Many problems of interest are **optimization problems**, in which each feasible (i.e., “legal”) solution has an associated value, and we wish to find a feasible solution with the best value. And some others are **decision problems**, in which the answer is simply “yes” or “no” (or, more formally, “1” or “0”).

Class P

We begin our study of NP-completeness by formalizing our notion of polynomial-time solvable problems.

Class P

We begin our study of NP-completeness by formalizing our notion of polynomial-time solvable problems.

To understand the class of polynomial-time solvable problems, we must first have a formal notion of what a "problem" is.

Abstract problems

We define an **abstract problem** Q to be a binary relation on a set I of problem **instances** and a set S of problem **solutions**.

Abstract problems

We define an **abstract problem** Q to be a binary relation on a set I of problem **instances** and a set S of problem **solutions**.

For example, an instance for `SHORTEST-PATH` is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.

Abstract problems

The theory of NP-completeness restricts attention to **decision problems**: those having a yes/no solution. In this case, we can view an **abstract decision problem** as a function that maps the instance set I to the solution set $\{0, 1\}$.

Encodings

In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the program understands.

Encodings

In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the program understands.

An **encoding** of a set S of abstract objects is a mapping e from S to the set of binary strings.

Encodings

A computer algorithm that “solves” some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a **concrete problem**.

Encodings

A computer algorithm that “solves” some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a **concrete problem**.

A concrete problem is **polynomial-time solvable**, if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.

Class P

We can now formally define the **complexity class P** as the set of concrete decision problems that are polynomial-time solvable.

Encodings

We can use encodings to map abstract problems to concrete problems. Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$, an encoding $e : I \to \{0, 1\}^*$ can induce a related concrete decision problem, which we denote by $e(Q)$.

If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$, then the solution to the concrete-problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$.

Encodings

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is polynomial-time computable if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$. For some set I of problem instances, we say that two encodings e_1 and e_2 are **polynomially related** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

Encodings

If two encodings e_1 and e_2 of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use.

Lemma 1
Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

A formal-language framework

An **alphabet** Σ is a finite set of symbols. A **language** L over Σ is any set of strings made up of symbols from Σ.

We denote the **empty string** by ϵ, the **empty language** by 0, and the language of all strings over Σ by Σ^*.

A formal-language framework

We can perform a variety of operations on languages. Set-theoretic operations, such as **union** and **intersection**, follow directly from the set-theoretic definitions.

We define the **complement** of L by $L = \Sigma^* - L$. The **concatenation** L_1L_2 of two languages L_1 and L_2 is the language

$$L = \{x_1x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

A formal-language framework

The **closure** or **Kleene star** of a language L is the language

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots ,$$

where L^k is the language obtained by concatenating L to itself k times.

Class P

The set of instances for any decision problem Q is simply the set Σ^*, where $\Sigma = \{0, 1\}$.

Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view Q as a language L over $\Sigma = \{0, 1\}$, where

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

Class P

We say that an algorithm A **accepts** a string $x \in \{0,1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1. The language **accepted** by an algorithm A is the set of strings

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

that is, the set of strings that the algorithm accepts.

An algorithm A **rejects** a string x if $A(x) = 0$.

Class P

A language L is **decided** by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.

Class P

A language L is **accepted in polynomial time** by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$.

A language L is **decided in polynomial time** by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.

Class P

We can informally define a **complexity class** as a set of languages, in which membership is determined by a **complexity measure**, such as running time, of an algorithm that determines whether a given string x belongs to language L.

Class P

$$\mathsf{P} = \{L \subseteq \{0,1\}^* : \text{there exists an algorithm } A \\ \text{that decides } L \text{ in polynomial time}\}.$$

In fact, P is also the class of languages that can be accepted in polynomial time.

Theorem 2
$\mathsf{P} = \{L : L$ *is accepted by a polynomial-time algorithm*$\}$.

Class P

Proof.
Because the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if L is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm.

Class P

Proof (Cont.)
Let L be the language accepted by some polynomial-time algorithm A. We shall use a classic “simulation” argument to construct another polynomial-time algorithm A' that decides L.

Because A accepts L in time $O(n^k)$ for some constant k, there also exists a constant c such that A accepts L in at most cn^k steps. For any input string x, the algorithm A' simulates cn^k steps of A.

Proof (Cont.)
After simulating cn^k steps, algorithm A' inspects the behavior of A. If A has accepted x, then A' accepts x by outputting a 1. If A has not accepted x, then A' rejects x by outputting a 0.

The overhead of A' simulating A does not increase the running time by more than a polynomial factor, and thus A' is a polynomial-time algorithm that decides L. $\square$

Questions?

References

T. Cormen, C. Leiserson, R. Rivest, and C. Stein.
Introduction to Algorithms.
Computer science. MIT Press, 2009.

Thanks!