CS:4330 Theory of Computation

Spring 2018

Computability Theory

The class P

Haniel Barbosa

THE UNIVERSITY OF IOWA

Readings for this lecture

Chapter 7 of [Sipser 1996], 3rd edition. Section 7.2.

Polynomial time

- ▷ Polynomial differences in running time are considered to be small whereas exponential differences are considered to be large.

- ▷ Consider the difference between growth rate of polynomial (e.g. n^3) and exponential (e.g. 2^n) functions, for $n = 10^3$:
  - $n^3 = 10^9$ is large but manageable
  - 2^n is a number "larger than the number of atoms in the universe", i.e. unmanageable

- ▷ In general polynomial time algorithms are fast enough for most purposes, while exponential time algorithms are less frequently useful

Source of complexity

- ▷ *Brute-force search*: exponential time algorithms that solve problems by exhaustively searching through a space of solutions
- ▷ An example is factoring a number into its constituent primes by searching through all potential divisors
- ▷ Sometimes brute-force search may be avoided through a deeper understanding of the problem, which may reveal polynomial time algorithms

Time complexity theory

- The aim of time complexity theory is to present fundamental properties of computation rather than properties of Turing machines or any other particular model

- Therefore we focus on aspects of computing that are unaffected by polynomial differences in running time

- This allows us to develop a theory that does not depend on the selection of a particular model of computation.

The class P

Definition

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine, i.e.

$$\mathbf{P} = \bigcup_k \mathrm{TIME}(n^k)$$

The class **P** plays a central role in time complexity theory because:

1. **P** is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM, i.e. **P** is mathematically robust
2. **P** roughly corresponds to the class of problems that are realistically solvable by computers, i.e. **P** is relevant from a practical standpoint

Practical relevance

- ▷ When a problem is in **P** we have a method to solve it in time n^k for some constant k. Whether n^k is practical depends on k and on the application
- ▷ Calling polynomial time the “threshold of practical solvability” has proven to be useful
- ▷ Once a polynomial time algorithm has been found for a problem that required exponential time, some key insight has been gained and further reductions in time complexity usually follow

Notational conventions for examples of problems in P

- We describe algorithms using numbered stages, where a stage is analogous to a high-level step of a Turing machine, or a sequence of simple steps of a Turing machine

- When we analyze an algorithm we need to do two things:
  1. Give a polynomial upper bound (using e.g. big-O) on the number of stages the algorithm uses
  2. Examine the individual stages in the description of the algorithm to be sure that each runs in polynomial time on a deterministic model

- When both tasks have been completed we can conclude that the algorithm runs in polynomial time because composition of polynomials is a polynomial

Notational conventions for examples of problems in P

- ▷ We use the notation $\langle \cdot \rangle$ to indicate a reasonable encoding of one or more objects into a string, without specifying any particular encoding method

- ▷ A reasonable method is one that allows for polynomial time encoding and decoding of objects into internal representations or into other reasonable encodings; familiar encoding methods for graphs, automata, etc., are reasonable

- ▷ Unreasonable encoding methods are those that generate exponentially large representations, such as using unary strings to encode natural numbers; to encode numbers one should use base k notation, for $k \geq 2$

Graph encodings

- As lists of nodes and edges
- Or as an adjacent matrix M where $M(i,j) = 1$ if there is an edge from node i to node j and $M(i,j) = 0$ otherwise
- Running time of graph algorithms may be computed in number of nodes instead of size of graph representation since the size of the representation is a polynomial in the number of nodes

The *PATH* problem

Does a directed path exists from two given nodes in a graph? Let

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

Theorem

$PATH \in \mathbf{P}$

Proof idea

Construct a polynomial time algorithm that decides *PATH*. Note that the brute-force algorithm that examines all potential paths in G and determines whether any is a direct path from s to t would not suffice.

Brute-force algorithm for *PATH*

- ▷ A potential path is a sequence of nodes in G having length at most m, in which m is the number of nodes in G
- ▷ If a direct path exists from s to t, one having a length at most m exists because repeating a node is never necessary
- ▷ The number of potential paths is roughly $\mathcal{O}(m^m)$, which is exponential in the number of nodes
- ▷ Therefore a brute-force approach must be avoided for deciding *PATH* in polynomial time

A polynomial time algorithm for *PATH*

$M =$"On input string $\langle G, s, t \rangle$ where G is a direct graph containing nodes s and t:

1. Place a mark on node s
2. Repeat until no additional nodes are marked:
    3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b
4. If t is marked, *accept*. Otherwise *reject*."

A polynomial time algorithm for *PATH*

$M =$"On input string $\langle G, s, t \rangle$ where G is a direct graph containing nodes s and t:

1. Place a mark on node s
2. Repeat until no additional nodes are marked:
    3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b
4. If t is marked, *accept*. Otherwise *reject*."

▷ Stages 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound

A polynomial time algorithm for *PATH*

$M =$"On input string $\langle G, s, t \rangle$ where G is a direct graph containing nodes s and t:

1. Place a mark on node s
2. Repeat until no additional nodes are marked:
    3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b
4. If t is marked, *accept*. Otherwise *reject*."

▷ Stages 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound

▷ Stage 3 runs at most m times because each time except the last it marks an additional node of G. Each run scans all the edges, which can be done polynomially on the number of edges. Hence, stages 2 and 3 are also polynomially bound

A polynomial time algorithm for *PATH*

$M =$"On input string $\langle G, s, t \rangle$ where G is a direct graph containing nodes s and t:

1. Place a mark on node s
2. Repeat until no additional nodes are marked:
    3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b
4. If t is marked, *accept*. Otherwise *reject*."

▷ Stages 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound

▷ Stage 3 runs at most m times because each time except the last it marks an additional node of G. Each run scans all the edges, which can be done polynomially on the number of edges. Hence, stages 2 and 3 are also polynomially bound

▷ Therefore M is a polynomial time algorithm for *PATH*

The RELPRIME problem

For $x, y \in \mathbb{N}$, x and y are *relatively prime* if 1 is the largest integer that evenly divides them both. Can we determine in polynomial time whether two numbers are relatively prime? Let

$$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}\}$$

Theorem

$RELPRIME \in \mathbf{P}$

Proof idea

Again, build a polynomial time algorithm avoiding brute-force, i.e. searching through all possible divisors of x and y and accepting if none is greater than 1.

A polynomial time algorithm for *RELPRIME*

We can use Euclidian algorithm to find the greatest common divisor of two numbers, $gcd(x, y)$:

$E =$"On input string $\langle x, y \rangle$ where $x, y \in \mathbb{N}^+$:

1. Repeat until $y = 0$
    2. Let $x \leftarrow x \bmod y$
    3. Exchange x with y
4. Output x"

Using E as a subroutine it is straightforward to build a solution R for *RELPRIME*:

$R =$"On input string $\langle x, y \rangle$ where $x, y \in \mathbb{N}^+$:

1. Run E on $\langle x, y \rangle$
2. If E returns 1, *accept*. Otherwise, *reject*."

If E runs in polynomial time, so does R.

Analyzing E

- ▷ After stage 2 is executed, $x < y$, by definition of mod
- ▷ After stage 3, $x > y$, since the two have been exchanged
- ▷ If $x/2 \geq y$, then x mod $y < y \leq x/2$ and x, after stage 2, drops by at least half. If $x/2 < y$, then x mod $y = x - y < x/2$ and x drops by at least half
- ▷ Therefore every execution of stage 2 (except possibly the first) cuts the value of x at least by half
- ▷ Since stage 3 exchanges x and y, each of their original values is reduced by half every other time through the loop
- ▷ Then the maximum number of times stages 2 and 3 are repeated is the lesser of $2\log_2 x$ and $2\log_2 y$. These logarithms are proportional to the length n of the representation of x and y
- ▷ Thus the number of stage executions is polynomially bounded and E is polynomial, which means that so is the solution to *RELPRIME*