

Introduction

Theorems 7.8 and 7.11 illustrate important distinction among models of TM:

- There is at most a polynomial difference between the time complexity of problems measured on deterministic single-tape and multitape TM.
- There is at most an exponential difference between the time complexity of problems measured on deterministic and nondeterministic TM.

22c:131 Limits of Computation

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c131>

The University of Iowa
Department of Computer Science

The Class P

Source of complexity

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

Polynomial time

Note: Polynomial differences in running time are considered to be small whereas exponential differences are considered to be large.

Example: Consider the difference between growth rate of polynomial (typically n^3) and exponential (typically 2^n)

- For $n = 1000$, $n^3 = 1,000,000,000$ a large but manageable number
- For $n = 1000$, 2^n is a number larger than the number of atoms in the universe, i.e., an unmanageable number.

Conclusion: polynomial time algorithms are fast enough for many purposes; exponential time algorithms are rarely useful

Class P

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

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

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.

Time complexity theory

- The aim of a 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 computations that are unaffected by polynomial differences in running time which are considered insignificant and thus are ignored.
- This allows us to develop a theory that doesn't depend on the selection of a particular model of computation.

Notational conventions

- 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 \mathcal{O} notation) on the number of stages that the algorithm uses when it run on an input of length n
 2. examine the individual stages in the description of the algorithm to be sure that each can be implemented 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.

Note

- 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 application
- **Example:** running time n^{100} is unlikely to be of any practical use. But calling polynomial time the threshold of practical solvability has proven to be useful.
- Once a polynomial time has been found for a problem that required exponential time some key insight have been gained and further reduction in its time complexity usually follows.

Graph encodings

- Lists of nodes and edges
- 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 the size of graph representation because graph representation is polynomial in number of nodes

Notations

- We use the notation $\langle \bullet \rangle$ 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 of computation model; familiar encoding methods for graphs, automata, etc., are reasonable
- Unreasonable encoding method are those that generate exponential large representations, such as using unary strings $\dots 111\dots$ to encode natural numbers; instead, base k notation for $k \geq 2$ should be used

Brute-forth algorithm for *PATH*

1. A potential path is a sequence of nodes in G having a length at most m where m is the number of nodes in G .
2. If a direct path exist from s to t , one having a length at most m exists because repeating a node is never necessary.
3. The number of potential paths is $O(k^m)$, which is exponential in number of nodes, where k is the maximum number of successors.

Conclusion: to get a polynomial algorithm that decides *PATH* we need to avoid the brute-forth approach.

The *PATH* problem

$PATH = \{(G, s, t) \mid G \text{ is a direct graph that has a direct path from } s \text{ to } t\}$

Theorem 7.14 $PATH \in P$

Proof idea: constrict a polynomial time algorithm that decides *PATH*.

Note: the brute-force algorithm that examines all potential paths in G and determine whether any is a direct path from s to t is not fast enough.

Analyzing M

- Stages 1 and 3 are executed only once, hence they are bound by $\mathcal{O}(m)$.
- Stage 2 runs at most m times because each time except the last it marks an additional node of G . Hence, it is bound by $\mathcal{O}(m)$
- The total time is bounded by $2\mathcal{O}(m) + \mathcal{O}(m) = \mathcal{O}(m)$

Proof

A polynomial time algorithm M for $PATH$ follows:

$M =$ "On input (G, s, t) where G is a direct graph with nodes s and t :

1. Place a mark on node s
2. Repeat until no additional nodes are marked:
 - (a) 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 .
3. If t is marked *accept*. Otherwise *reject*."

Proof idea

- Brute-force algorithm to solve $RELPRIME$: search through all possible divisors of x and y and accept if none is greater than 1
- The magnitude of a number represented in any base $k \geq 2$ is exponential in its representation base.
- That is, brute-force search is an exponential running time algorithm

Another example

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

For $x, y \in \mathcal{N}$, x and y are relative prime if 1 is the largest integer that evenly divide x and y .

Theorem 7.15 $RELPRIME \in P$

Algorithm solving *RELPRIME*

The following algorithm R solves *RELPRIME* using E :

$R =$ "On input $\langle x, y \rangle$ where $x, y \in \mathcal{N}$:

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

Note: clearly if E runs in polynomial time so does R hence we only need to analyze E for time complexity and correctness.

A better idea

Use **Euclidean algorithm** that determines the greatest common divisor of two number, $gcd(x, y)$, to solve this problem. Then if $gcd(x, y) = 1$ accept, otherwise reject.

Euclidean algorithm, E:

$E =$ "On input $\langle x, y \rangle$ where $x, y \in \mathcal{N}$:

1. Repeat until $y = 0$
 - (a) Let $x := x \bmod y$
 - (b) Exchange x with y
2. Output x ."

Analyzing E

- Every execution of 1(a) (except possible first) cuts the value of x by at least half because $x \bmod y = \text{remainder}(x \text{ div } y)$ and $y \geq 2$
- Since $\text{remainder}(x \text{ div } y) < y$ after stage 1(b) $x < y$
- The values of x and y are reduced by at least half every time through the loop 1
- The maximum number of times loop 1 is executed is less than $2 \log_2 x$ and $2 \log_2 y$. These logarithms are proportional to the length n of representations.

Thus the number of times loop executes is bounded by $\mathcal{O}(n)$ and E is polynomial.