Complexity Theory
Lecture 3

Anuj Dawar

University of Cambridge Computer Laboratory
Easter Term 2013

http://www.cl.cam.ac.uk/teaching/1213/Complexity/

Complexity

For any function $f : \mathbb{N} \to \mathbb{N}$, we say that a language L is in $\mathsf{TIME}(f(n))$ if there is a machine $M = (Q, \Sigma, s, \delta)$, such that:

- $L = L(M)$; and
- The running time of M is $O(f(n))$.

Similarly, we define $\mathsf{SPACE}(f(n))$ to be the languages accepted by a machine which uses $O(f(n))$ tape cells on inputs of length n.

In defining space complexity, we assume a machine M, which has a read-only input tape, and a separate work tape. We only count cells on the work tape towards the complexity.

Complexity Classes

A complexity class is a collection of languages determined by three things:

- A model of computation (such as a deterministic Turing machine, or a nondeterministic TM, or a parallel Random Access Machine).
- A resource (such as time, space or number of processors).
- A set of bounds. This is a set of functions that are used to bound the amount of resource we can use.

Polynomial Bounds

By making the bounds broad enough, we can make our definitions fairly independent of the model of computation.

> The collection of languages recognised in *polynomial time* is the same whether we consider Turing machines, register machines, or any other deterministic model of computation.
>
> The collection of languages recognised in *linear time*, on the other hand, is different on a one-tape and a two-tape Turing machine.

We can say that being recognisable in polynomial time is a property of the language, while being recognisable in linear time is sensitive to the model of computation.

Polynomial Time

$$P = \bigcup_{k=1}^{\infty} \mathsf{TIME}(n^k)$$

The class of languages decidable in polynomial time.

The complexity class P plays an important role in our theory.

- It is robust, as explained.
- It serves as our formal definition of what is *feasibly computable*

One could argue whether an algorithm running in time $\theta(n^{100})$ is feasible, but it will eventually run faster than one that takes time $\theta(2^n)$.

Making the distinction between polynomial and exponential results in a useful and elegant theory.

Example: Reachability

The Reachability decision problem is, given a *directed* graph $G = (V, E)$ and two nodes $a, b \in V$, to determine whether there is a path from a to b in G.

A simple search algorithm as follows solves it:

1. mark node a, leaving other nodes unmarked, and initialise set S to $\{a\}$;
2. while S is not empty, choose node i in S: remove i from S and for all j such that there is an edge (i, j) and j is unmarked, mark j and add j to S;
3. if b is marked, accept else reject.

Analysis

This algorithm requires $O(n^2)$ time and $O(n)$ space.

The description of the algorithm would have to be refined for an implementation on a Turing machine, but it is easy enough to show that:

$$\mathsf{Reachability} \in \mathsf{P}$$

To formally define Reachability as a language, we would have to also choose a way of representing the input (V, E, a, b) as a string.

Example: Euclid's Algorithm

Consider the decision problem (or *language*) RelPrime defined by:

$$\{(x, y) \mid \gcd(x, y) = 1\}$$

The standard algorithm for solving it is due to Euclid:

1. Input (x, y).
2. Repeat until $y = 0$: $x \leftarrow x \bmod y$; Swap x and y
3. If $x = 1$ then accept else reject.

Analysis

The number of repetitions at step 2 of the algorithm is at most $O(\log x)$.

why?

This implies that RelPrime is in P.

If the algorithm took $\theta(x)$ steps to terminate, it would not be a polynomial time algorithm, as x is not polynomial in the *length* of the input.

Primality

Consider the decision problem (or *language*) Prime defined by:

$$\{x \mid x \text{ is prime}\}$$

The obvious algorithm:

For all y with $1 < y \leq \sqrt{x}$ check whether $y|x$.

requires $\Omega(\sqrt{x})$ steps and is therefore *not* polynomial in the length of the input.

Is Prime $\in$ P?

Boolean Expressions

Boolean expressions are built up from an infinite set of variables

$$X = \{x_1, x_2, \ldots\}$$

and the two constants `true` and `false` by the rules:

- a constant or variable by itself is an expression;
- if ϕ is a Boolean expression, then so is $(\neg\phi)$;
- if ϕ and ψ are both Boolean expressions, then so are $(\phi \wedge \psi)$ and $(\phi \vee \psi)$.

Evaluation

If an expression contains no variables, then it can be evaluated to either `true` or `false`.

Otherwise, it can be evaluated, *given* a truth assignment to its variables.

Examples:
$(\texttt{true} \vee \texttt{false}) \wedge (\neg\texttt{false})$
$(x_1 \vee \texttt{false}) \wedge ((\neg x_1) \vee x_2)$
$(x_1 \vee \texttt{false}) \wedge (\neg x_1)$
$(x_1 \vee (\neg x_1)) \wedge \texttt{true}$

Boolean Evaluation

There is a deterministic Turing machine, which given a Boolean expression *without variables* of length n will determine, in time $O(n^2)$ whether the expression evaluates to `true`.

The algorithm works by scanning the input, rewriting formulas according to the following rules:

Rules

- $(\texttt{true} \vee \phi) \Rightarrow \texttt{true}$
- $(\phi \vee \texttt{true}) \Rightarrow \texttt{true}$
- $(\texttt{false} \vee \phi) \Rightarrow \phi$
- $(\texttt{false} \wedge \phi) \Rightarrow \texttt{false}$
- $(\phi \wedge \texttt{false}) \Rightarrow \texttt{false}$
- $(\texttt{true} \wedge \phi) \Rightarrow \phi$
- $(\neg\texttt{true}) \Rightarrow \texttt{false}$
- $(\neg\texttt{false}) \Rightarrow \texttt{true}$

Analysis

Each scan of the input ($O(n)$ steps) must find at least one subexpression matching one of the rule patterns.

Applying a rule always eliminates at least one symbol from the formula.

Thus, there are at most $O(n)$ scans required.

The algorithm works in $O(n^2)$ steps.

Satisfiability

For Boolean expressions ϕ that contain variables, we can ask

> Is there an assignment of truth values to the variables which would make the formula evaluate to `true`?

The set of Boolean expressions for which this is true is the language SAT of *satisfiable* expressions.

This can be decided by a deterministic Turing machine in time $O(n^2 2^n)$.

An expression of length n can contain at most n variables.

For each of the 2^n possible truth assignments to these variables, we check whether it results in a Boolean expression that evaluates to `true`.

Is $\mathsf{SAT} \in \mathsf{P}$?

Circuits

A circuit is a directed graph $G = (V, E)$, with $V = \{1, \ldots, n\}$ together with a labeling: $l : V \to \{\mathtt{true}, \mathtt{false}, \wedge, \vee, \neg\}$, satisfying:

- If there is an edge (i, j), then $i < j$;
- Every node in V has *indegree* at most 2.
- A node v has
  indegree 0 iff $l(v) \in \{\mathtt{true}, \mathtt{false}\}$;
  indegree 1 iff $l(v) = \neg$;
  indegree 2 iff $l(v) \in \{\vee, \wedge\}$

The value of the expression is given by the value at node n.

CVP

A circuit is a more compact way of representing a Boolean expression.

Identical subexpressions need not be repeated.

CVP - the *circuit value problem* is, given a circuit, determine the value of the result node n.

CVP is solvable in polynomial time, by the algorithm which examines the nodes in increasing order, assigning a value `true` or `false` to each node.

Composites

Consider the decision problem (or *language*) Composite defined by:

$$\{x \mid x \text{ is not prime}\}$$

This is the complement of the language Prime.

Is Composite $\in$ P?

Clearly, the answer is yes if, and only if, Prime $\in$ P.