Topics in Logic and Complexity

Handout 1

Anuj Dawar

http://www.cl.cam.ac.uk/teaching/2122/L15

What is This Course About?

Complexity Theory is the study of what makes some algorithmic problems inherently difficult to solve.

Difficult in the sense that there is no efficient algorithm.

Mathematical Logic is the study of formal mathematical reasoning.

It gives a mathematical account of meta-mathematical notions such as structure, language and proof.

In this course we will see how logic can be used to study complexity theory. In particular, we will look at how complexity relates to *definability*.

Computational Complexity

Complexity is usually defined in terms of *running time* or *space* asymptotically required by an algorithm. *E.g.*

- Merge Sort runs in time $O(n \log n)$.
- Any sorting algorithm that can sort an arbitrary list of n numbers requires time $\Omega(n \log n)$.

Complexity theory is concerned with the hardness of *problems* rather than specific algorithms.

> *We will mostly be concerned with broad classification of complexity: logarithmic vs. polynomial vs. exponential.*

Graph Properties

For simplicity, we often focus on *decision problems*.

As an example, consider the following three decision problems on *graphs*.

1. Given a graph $G = (V, E)$ does it contain a *triangle*?
2. Given a directed graph $G = (V, E)$ and two of its vertices $s, t \in V$, does G contain a *path* from s to t?
3. Given a graph $G = (V, E)$ is it *3-colourable*? That is,
   is there a function $\chi : V \to \{1, 2, 3\}$ *so that whenever* $(u, v) \in E$, $\chi(u) \neq \chi(v)$.

Graph Properties

1. Checking if G contains a triangle can be solved in *polynomial time* and *logarithmic space*.

2. Checking if G contains a path from s to t can be done in *polynomial time*.
Can it be done in *logarithmic space*?
Unlikely. It is NL*-complete.*

3. Checking if G is 3-colourable can be done in *exponential time* and *polynomial space*.
Can it be done in *polynomial time*?
Unlikely. It is NP*-complete.*

Logical Definability

In what kind of formal language can these decision problems be *specified* or *defined*?

The graph $G = (V, E)$ contains a triangle.

$$\exists x \in V \, \exists y \in V \, \exists z \in V (x \neq y \wedge y \neq z \wedge x \neq z \wedge E(x, y) \wedge E(x, z) \wedge E(y, z))$$

The other two properties are *provably* not definable with only first-order quantification over vertices.

Second-Order Quantifiers

3-Colourability and *reachability* can be defined with quantification over *sets of vertices*.

$$\begin{aligned}\exists R \subseteq V\, \exists B \subseteq V\, \exists G \subseteq V\ & \\ & \forall x(Rx \vee Bx \vee Gx) \wedge \\ & \forall x(\neg(Rx \wedge Bx) \wedge \neg(Bx \wedge Gx) \wedge \neg(Rx \wedge Gx)) \wedge \\ & \forall x \forall y(Exy \rightarrow (\neg(Rx \wedge Ry) \wedge \\ & \qquad \neg(Bx \wedge By) \wedge \\ & \qquad \neg(Gx \wedge Gy)))\end{aligned}$$

$$\forall S \subseteq V(s \in S \wedge \forall x \forall y((x \in S \wedge E(x,y)) \rightarrow y \in S) \rightarrow t \in S)$$

Course Outline

This course is concerned with the questions of (1) how definability relates to computational complexity and (2) how to analyse definability.

1. Complexity Theory—a review of the major complexity classes and their interrelationships (3L).
2. First-order and second-order logic—their expressive power and computational complexity (3L).
3. Lower bounds on expressive power—the use of games and locality (3L).
4. Fixed-point logics and descriptive complexity (3L).
5. Complexity of Constraint Satisfaction Problems (4L).

Useful Information

Some useful books:

- C.H. Papadimitriou. Computational Complexity. Addison-Wesley. 1994.
- S. Arora and B. Barak. Computational Complexity. CUP. 2009.
- H.-D. Ebbinghaus and J. Flum. Finite Model Theory (2nd ed.) 1999.
- N. Immerman. Descriptive Complexity. Springer. 1999.
- L. Libkin. Elements of Finite Model Theory. Springer. 2004.
- E. Grädel et al. Finite Model Theory and its Applications. Springer. 2007.

Course website: `http://www.cl.cam.ac.uk/teaching/2122/L15/`

Decision Problems and Algorithms

Formally, a *decision problem* is a set of strings $L \subseteq \Sigma^*$ over a finite alphabet Σ.

The problem is *decidable* if there is an *algorithm* which given any input $x \in \Sigma^*$ will determine whether $x \in L$ or not.

The notion of an *algorithm* is formally defined by a *machine model*: A *Turing Machine*; *Random Access Machine* or even a *Java program*. The choice of machine model doesn't affect what is or is not decidable.

Similarly, we say a function $f : \Sigma^* \to \Delta^*$ is *computable* if there is an algorithm which takes input $x \in \Sigma^*$ and gives output $f(x)$.

Turing Machines

For our purposes, a Turing Machine consists of:

- K — a finite set of states;
- Σ — a finite set of symbols, including $\sqcup$ and $\triangleright$.
- $s \in K$ — an initial state;
- $\delta : (K \times \Sigma) \to (K \cup \{\text{acc}, \text{rej}\}) \times \Sigma \times \{L, R, S\}$
  A transition function that specifies, for each state and symbol a next state (or accept acc or reject rej), a symbol to overwrite the current symbol, and a direction for the tape head to move (L – left, R – right, or S - stationary)

Configurations

A complete description of the configuration of a machine can be given if we know what state it is in, what are the contents of its tape, and what is the position of its head. This can be summed up in a simple triple:

Definition

A *configuration* is a triple (q, w, u), where $q \in K$ and $w, u \in \Sigma^\star$

The intuition is that (q, w, u) represents a machine in state q with the string wu on its tape, and the head pointing at the last symbol in w.

The configuration of a machine completely determines the future behaviour of the machine.

Computations

Given a machine $M = (K, \Sigma, s, \delta)$ we say that a configuration (q, w, u) *yields in one step* (q', w', u'), written

$$(q, w, u) \rightarrow_M (q', w', u')$$

if

- $w = va$;
- $\delta(q, a) = (q', b, D)$; and
- either $D = L$ and $w' = v$ $u' = bu$
  or $D = S$ and $w' = vb$ and $u' = u$
  or $D = R$ and $w' = vbc$ and $u' = x$, where $u = cx$. If u is empty, then $w' = vb\sqcup$ and u' is empty.

Computations

The relation $\rightarrow_M^\star$ is the reflexive and transitive closure of $\rightarrow_M$.
A sequence of configurations $c_1, \ldots, c_n$, where for each i, $c_i \rightarrow_M c_{i+1}$, is called a *computation* of M.
The language $L(M) \subseteq \Sigma^\star$ *accepted* by the machine M is the set of strings

$$\{x \mid (s, \triangleright, x) \rightarrow_M^\star (\mathsf{acc}, w, u) \text{ for some } w \text{ and } u\}$$

A machine M is said to *halt on input* x if for some w and u, either $(s, \triangleright, x) \rightarrow_M^\star (\mathsf{acc}, w, u)$ or $(s, \triangleright, x) \rightarrow_M^\star (\mathsf{rej}, w, u)$

Complexity

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

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

Similarly, we define $\mathrm{SPACE}(f(n))$ 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.

Nondeterminism

If, in the definition of a Turing machine, we relax the condition on δ being a function and instead allow an arbitrary relation, we obtain a *nondeterministic Turing machine*.

$$\delta \subseteq (K \times \Sigma) \times (K \cup \{\text{acc}, \text{rej}\} \times \Sigma \times \{R, L, S\}).$$

The yields relation $\rightarrow_M$ is also no longer functional.

We still define the language accepted by M by:

$$L(M) = \{x \mid (s, \triangleright, x) \rightarrow_M^\star (\text{acc}, w, u) \text{ for some } w \text{ and } u\}$$

though, for some x, there may be computations leading to accepting as well as rejecting states.

Nondeterministic Complexity

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

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

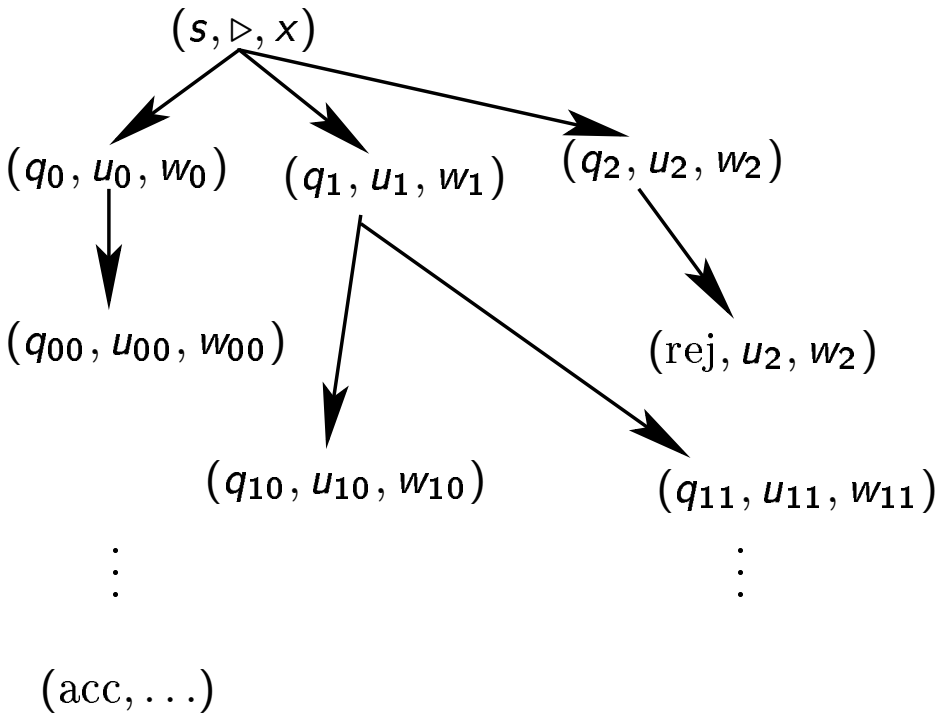
The last statement means that for each $x \in L(M)$, there is a computation of M that accepts x and whose length is bounded by $O(f(|x|))$.

Similarly, we define NSPACE($f(n)$) to be the languages accepted by a *nondeterminstic* machine which uses $O(f(n))$ tape cells on inputs of length n.

As before, in reckoning space complexity, we only count work space.

Computation Trees

With a nondeterministic machine, each configuration gives rise to a tree of successive configurations.



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 Computation

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

The class of languages decidable in polynomial time.
The complexity class P plays an important role in complexity theory.

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

Nondeterministic Polynomial Time

$$\mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k)$$

That is, NP is the class of languages accepted by a *nondeterministic* machine running in polynomial time.

Since a deterministic machine is just a nondeterministic machine in which the transition relation is *functional*, $\mathsf{P} \subseteq \mathsf{NP}$.

Succinct Certificates

The complexity class NP can be characterised as the collection of languages of the form:

$$L = \{x \mid \exists y\ R(x, y)\}$$

where R is a relation on strings satisfying two key conditions

1. R is decidable in polynomial time.
2. R is *polynomially balanced*. That is, there is a polynomial p such that if $R(x, y)$ and the length of x is n, then the length of y is no more than $p(n)$.

Equivalence of Definitions

If $L = \{x \mid \exists y\ R(x, y)\}$ we can define a nondeterministic machine M that accepts L.

The machine first uses nondeterministic branching to *guess* a value for y, and then checks whether $R(x, y)$ holds.

In the other direction, suppose we are given a nondeterministic machine M which runs in time $p(n)$.
Suppose that for each $(q, \sigma) \in K \times \Sigma$ (i.e. each state, symbol pair) there are at most k elements in $\delta(q, \sigma)$.

Equivalence of Definitions

For y a string over the alphabet $\{1, \ldots, k\}$, we define the relation $R(x, y)$ by:

- $|y| \leq p(|x|)$; and
- the computation of M on input x which, at step i takes the "$y[i]$th transition" is an accepting computation.

Then, $L(M) = \{x \mid \exists y\ R(x, y)\}$

Space Complexity Classes

$\mathsf{L} = \mathsf{SPACE}(\log n)$

The class of languages decidable in logarithmic space.

$\mathsf{NL} = \mathsf{NSPACE}(\log n)$

The class of languages decidable by a nondeterministic machine in logarithmic space.

$\mathsf{PSPACE} = \bigcup_{k=1}^{\infty} \mathsf{SPACE}(n^k)$

The class of languages decidable in polynomial space.

$\mathsf{NPSPACE} = \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(n^k)$

Inclusions between Classes

We have the following inclusions:

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXP}$$

where $\mathsf{EXP} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(2^{n^k})$

Of these, the following are direct from the definitions:

$$\mathsf{L} \subseteq \mathsf{NL}$$

$$\mathsf{P} \subseteq \mathsf{NP}$$

$$\mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$$

NP $\subseteq$ PSPACE

To simulate a nondeterministic machine M running in time $t(n)$ by a deterministic one, it suffices to carry out a *depth-first* search of the computation tree.
We keep a counter to cut off branches that exceed $t(n)$ steps.

The space required is:

- a *counter* to count up to $t(n)$; and
- a *stack* of configurations, each of size at most $O(t(n))$.

The depth of the stack is at most $t(n)$.
Thus, if t is a polynomial, the total space required is polynomial.

NL $\subseteq$ P

Given a nondeterministic machine M that works with *work space* bounded by $s(n)$ and an input x of length n, there is some constant c such that

> *the total number of possible configurations of M within space bounds $s(n)$ is bounded by $n \cdot c^{s(n)}$.*

Define the *configuration graph* of M, x to be the graph whose nodes are the possible configurations, and there is an edge from i to j if, and only if, $i \rightarrow_M j$.

Reachability in the Configuration Graph

M accepts x if, and only if, some accepting configuration is reachable from the starting configuration in the configuration graph of M, x.

Using an $O(n^2)$ algorithm for Reachability, we get that M can be simulated by a deterministic machine operating in time

$$c'(nc^{s(n)})^2 \sim c'c^{2(\log n + s(n))} \sim d^{(\log n + s(n))}$$

for some constant d.

When $s(n) = O(\log n)$, this is polynomial and so $\mathsf{NL} \subseteq \mathsf{P}$.
When $s(n)$ is polynomial this is exponential in n and so $\mathsf{NPSPACE} \subseteq \mathsf{EXP}$.

Nondeterministic Space Classes

If Reachability were solvable by a *deterministic* machine with logarithmic space, then

$$\mathsf{L} = \mathsf{NL}.$$

In fact, Reachability is solvable by a deterministic machine with space $O((\log n)^2)$.

This implies

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{SPACE}((s(n)^2)).$$

In particular $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Reachability in $O((\log n)^2)$

$O((\log n)^2)$ space Reachability algorithm:

Path(a, b, i)
if $i = 1$ and $a \neq b$ and (a, b) is not an edge reject
else if (a, b) is an edge or $a = b$ accept
else, for each node x, check:

1. is there a path $a - x$ of length $i/2$; and
2. is there a path $x - b$ of length $i/2$?

if such an x is found, then accept, else reject.

The maximum depth of recursion is $\log n$, and the number of bits of information kept at each stage is $3 \log n$.

Inclusions between Classes

This leaves us with the following:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$$

Hierarchy Theorems proved by *diagonalization* can show that:

$$L \neq PSPACE \quad NL \neq NPSPACE \quad P \neq EXP$$

For other inclusions above, it remains an open question whether they are strict.

Complement Classes

If we interchange accepting and rejecting states in a deterministic machine that accepts the language L, we get one that accepts $\overline{L}$.

If a language $L \in \mathsf{P}$, then also $\overline{L} \in \mathsf{P}$.

Complexity classes defined in terms of nondeterministic machine models are not necessarily closed under complementation of languages.

Define,
co-NP – the languages whose complements are in NP.
co-NL – the languages whose complements are in NL.

Relationships

$P \subseteq NP \cap co\text{-}NP$ and any of the situations is consistent with our present state of knowledge:

- $P = NP = co\text{-}NP$
- $P = NP \cap co\text{-}NP \neq NP \neq co\text{-}NP$
- $P \neq NP \cap co\text{-}NP = NP = co\text{-}NP$
- $P \neq NP \cap co\text{-}NP \neq NP \neq co\text{-}NP$

It follows from the fact that $PSPACE = NPSPACE$ that $NPSPACE$ is closed under complementation.

Also, **Immerman and Szelepcsényi** showed that $NL = co\text{-}NL$.

Reductions

Given two languages $L_1 \subseteq \Sigma_1^\star$, and $L_2 \subseteq \Sigma_2^\star$,

a *reduction* of L_1 to L_2 is a *computable* function

$$f : \Sigma_1^\star \rightarrow \Sigma_2^\star$$

such that for every string $x \in \Sigma_1^\star$,

$$f(x) \in L_2 \text{ if, and only if, } x \in L_1$$

Resource Bounded Reductions

If f is computable by a polynomial time algorithm, we say that L_1 is *polynomial time reducible* to L_2.

$$L_1 \leq_P L_2$$

If f is also computable in $\mathrm{SPACE}(\log n)$, we write

$$L_1 \leq_L L_2$$

Reductions 2

If $L_1 \leq L_2$ we understand that L_1 is no more difficult to solve than L_2.

That is to say, for any of the complexity classes $\mathcal{C}$ we consider,

If $L_1 \leq L_2$ *and* $L_2 \in \mathcal{C}$, *then* $L_1 \in \mathcal{C}$

We can get an algorithm to decide L_1 by first computing f, and then using the $\mathcal{C}$-algorithm for L_2.
Provided that $\mathcal{C}$ is *closed* under such reductions.

Completeness

The usefulness of reductions is that they allow us to establish the *relative* complexity of problems, even when we cannot prove absolute lower bounds.

Cook and independently **Levin** first showed that there are problems in NP that are maximally difficult.

For any complexity class $\mathcal{C}$, a language L is said to be *$\mathcal{C}$-hard* if for every language $A \in \mathcal{C}$, $A \leq L$.

A language L is *$\mathcal{C}$-complete* if it is in $\mathcal{C}$ and it is $\mathcal{C}$-hard.

Complete Problems

Examples of complete problems for various complexity classes.

NL	Reachability
P	Game, Circuit Value Problem
NP	Satisfiability of Boolean Formulas, Graph 3-Colourability, Hamiltonian Cycle
co-NP	Validity of Boolean Formulas, Non 3-colourability
PSPACE	Geography, The game of HEX

P-complete Problems

Game

Input: A directed graph $G = (V, E)$ with a partition $V = V_1 \cup V_2$ of the vertices and two distinguished vertices $s, t \in V$.
Decide: whether *Player 1* can force a token from s to t in the game where when the token is on $v \in V_1$, *Player 1* moves it along an edge leaving v and when it is on $v \in V_2$, *Player 2* moves it along an edge leaving v.

Circuit Value Problem

A *Circuit* is a *directed acyclic graph* $G = (V, E)$ where each node has *in-degree* $0, 1$ or 2 and there is exactly one vertex t with no outgoing edges, along with a labelling which assigns:

- to each node of indegree 0 a value of 0 or 1
- to each node of indegree 1 a label $\neg$
- to each node of indegree 2 a label $\wedge$ or $\vee$

The problem CVP is, given a circuit, decide if the target node t evaluates to 1.

NP-complete Problems

SAT

Input: A Boolean formula ϕ

Decide: if there is an assignment of truth values to the variables of ϕ that makes ϕ true.

Hamiltonicity

Input: A graph $G = (V, E)$

Decide: if there is a cycle in G that visits every vertex exactly once.

co-NP-complete Problems

VAL

Input: A Boolean formula ϕ

Decide: if every assignment of truth values to the variables of ϕ makes ϕ true.

Non-3-colourability

Input: A graph $G = (V, E)$

Decide: if there is no function $\chi : V \to \{1, 2, 3\}$ such that the two endpoints of every edge are differently coloured.

PSPACE-complete Problems

Geography is very much like *Game* but now players are not allowed to visit a vertex that has been previously visitied.

HEX is a game played by two players on a graph $G = (V, E)$ with a source and target $s, t \in V$.
The two players take turns selecting vertices from V—neither player can choose a vertex that has been previously selected. Player 1 wins if, at any point, the vertices she has selected include a path from s to t. Player 2 wins if all vertices have been selected and no such path is formed.
The problem is to decide which player has a winning strategy.