On Syntactic and Semantic Complexity Classes

Anuj Dawar

University of Cambridge Computer Laboratory

Spitalfields Day, Isaac Newton Institute, 9 January 2012

Semantics and Syntax

Semantics and Syntax: A Legacy of Alan Turing

Syntax: symbolic manipulation.

Semantics: interpreted structure.

The notion of *computability* elucidates when semantics can be reduced to syntax.

Complexity Theory

Complexity Theory studies the computational complexity of algorithmic problems.

Classify *decision problems* according to *resources* required for their solution on some *machine model* of computation.

For example,

P—languages decidable by a deterministic Turing machine running in polynomial time.

NP—languages decidable by a nondeterministic Turing machine running in polynomial time.

Descriptive Complexity provides an alternative, *syntactic*, perspective on Computational Complexity.

Graph Problems

Consider some decision problems where the input is a graph.

1. *Connectedness*

   Given: a graph G

   Decide: is G connected?

2. *3-Colouring*

   Given: a graph G

   Decide: is there an assignment of three colours r, b, g to the vertices of G so that the endpoints of every edge are distinctly coloured?

3. *Hamiltonicity*

   Given: a graph G

   Decide: does G contain a cycle that visits every vertex exactly once?

First-Order Logic

Consider *first-order predicate logic* as a way of specifying a problem.

A collection X of variables, and formulas:

$$E(x,y) \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \exists x\varphi \mid \forall x\varphi$$

In addition, we may sometimes allow colours $R(x)$ and constants $E(c,x)$.

A formula φ without free variables specifies a property of graphs.

$$\forall x \forall y \forall z(\neg E(x,y) \vee \neg E(x,z) \vee \neg E(y,z))$$

defines the graphs that do not contain a triangle.

Weakness of First-Order Logic

For any fixed φ, the class of graphs G such that $G \models \varphi$ is decidable in *polynomial time* and *logarithmic space*.

There are computationally easy classes that are not defined by any first-order sentence, including *connectedness*.

None of the problems listed before is defiable in first-order logic.

Second-Order Logic

Second-order logic is obtained by adding to the defining rules of first-order logic two further clauses:

atomic formulae – $X(t_1, \ldots, t_a)$, where X is a *second-order variable*

second-order quantifiers – $\exists X \varphi$, $\forall X \varphi$

Second-order logic can express all three of the problems above.

Indeed, it can express every *NP-complete* problem.

Example

3-Colourability

$$\exists R \exists B \exists G \quad \forall x(Rx \vee Bx \vee Gx) \wedge$$
$$\forall x(\quad \neg(Rx \wedge Bx) \wedge \neg(Bx \wedge Gx) \wedge \neg(Rx \wedge Gx)) \wedge$$
$$\forall x \forall y(Exy \rightarrow (\quad \neg(Rx \wedge Ry) \wedge$$
$$\neg(Bx \wedge By) \wedge$$
$$\neg(Gx \wedge Gy)))$$

A Syntactic Characterisation of NP

Theorem (Fagin 1974):

A class of graphs is definable in existential second-order logic if, and only if, it is in the class NP.

A major open problem (first posed by Chandra and Harel in 1982) is to establish whether there is a similar descriptive characterisation of P.

Does P admit a syntactic characterisation?

Can the class P be “built up from below” by finitely many operations?

What would a *negative* answer to these questions look like?

Complexity Classes

What *type* of object is a complexity class?

Fix an alphabet of symbols, say $\{0, 1\}$.

A *string* is a finite sequence of symbols.

A *language* is a set of strings.

A *complexity class* is a set of languages.

There are $2^{2^{\aleph_0}}$ such sets.

Mostly, we're concerned with countable sets of *decidable* languages.

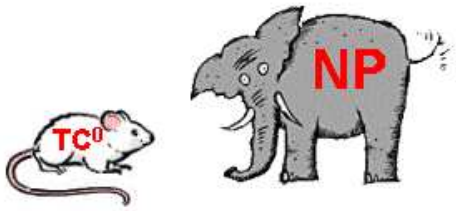
Complexity Classes

There is no *mathematical* definition of what constitutes a complexity class.

We aim to classify *decision problems* according to *resources* required for their solution on some *machine model* of computation.

- a machine model may be a *Turing machine*, *parallel RAM*, family of *Boolean circuits*, *interactive protocols*, etc.
- resources may be *running time*, *space*, *depth*, *number of rounds*, etc.

Zoo of Complexity Classes



Scott Aaronson and others have compiled an online *zoo* of complexity classes, which has 495 entries and counting.

P—languages decidable by a deterministic Turing machine running in polynomial time.

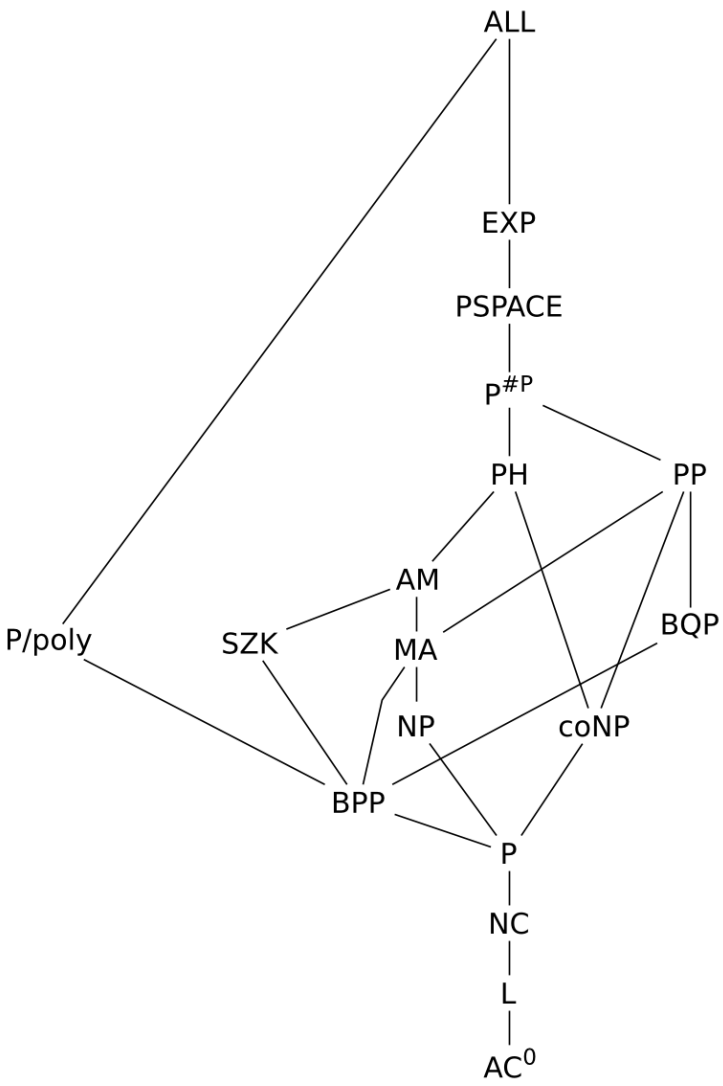
NP—languages decidable by a nondeterministic Turing machine running in polynomial time.

co-NP—languages whose complements are in NP.

BPP—probabilistic polyonomial time.

AC^0—languages decided by a uniform family of constant-depth, polynomial-size Boolean circuits.

Some Inclusions among Classes



Note: P/poly and All are uncountable classes that necessarily contain undecidable languages.

AC^0 is properly included in L.

L is known to be properly included in PSPACE but none of theindividual inclusions in between is known to be proper.

Enumerating Complexity Classes

Given a complexity class $\mathcal{C}$, can we enumerate its members?

Fix an enumeration of *Turing machines* and write L_i for the language accepted by machine M_i.

Can we decide the set $\{i \mid L_i \in \mathcal{C}\}$?

No—Rice's theorem.

Say that $\mathcal{C}$ is *weakly indexed* by a set $I \subseteq \mathbb{N}$ if:

- $i \in I \quad \Rightarrow \quad L_i \in \mathcal{C}$
- $L \in \mathcal{C} \quad \Rightarrow \quad \exists i \in I \; L = L_i$

Can we computably enumerate a weak index set for $\mathcal{C}$?

Syntactic Classes

Usually we want something more of a syntactic characterisation of $\mathcal{C}$ then just a computably enumerable weak index set. We want the machines M_i to *"witness"* that L_i is in $\mathcal{C}$.

For instance, fix an enumeration of pairs (M, p) where M is a deterministic Turing machine and p is a polynomial.

Let I be the range of the function that takes (M, p) to the code of the Turing machine that simulates M for $p(n)$ steps on inputs of length n.

I is an *effective syntax* for P.

Syntactic Classes

NP can similarly be indexed by pairs (M, p) where M is a *nondeterministic* Turing machine and p is a polynomial.

What about NP $\cap$ co-NP?

An index set is obtained by taking

$$(M, M', p) \quad \text{such that} \quad L(M) = L(M')$$

$$\uparrow$$

undecidable condition

So we say P and NP are *syntactic* classes, while NP $\cap$ co-NP is a *semantic* class.

It is an open question whether NP $\cap$ co-NP has a computably enumerable index set.

BPP

BPP is the complexity class of problems solvable by *randomised* polynomial time algorithms.

Formally, a language L is in BPP if there is a *nondeterministic* Turing machine M, running in polynomial time such that

- if $x \in L$ then $> \frac{2}{3}$ of the computations of M on input x are accepting; and
- if $x \notin L$ then $< \frac{1}{3}$ of the computations of M on input x are accepting.

Say that a machine M is *well-formed* for BPP if, for every string x, it is the case that either $< \frac{1}{3}$ or $> \frac{2}{3}$ of the computations of M on input x are accepting.

Index set for BPP

We can obtain an index set for BPP by enumerating all pairs

$$(M, p)$$

where M is a nondeterministic Turing machine and p is a polynomial such that, M clocked by p is *well-formed* for BPP.

The well-formedness condition is *undecidable*.

It is an open question whether there is a computably enumerable index set for BPP.

Complete Problems

Say a language L_1 is (AC^0)-*reducible* to L_2

$$L_1 \leq_{(AC^0)} L_2$$

if there is an (AC^0)-computable function f such that

$$f(x) \in L_2 \quad \Leftrightarrow \quad x \in L_1.$$

We say L is $\mathcal{C}$-*complete* if $L \in \mathcal{C}$ and for every L' we have $L' \leq L$.

SAT, *3-colourability* and *Hamiltonicity* are all NP-complete under AC^0 reductions.

There are many natural P-complete problems under AC^0 reductions.

Syntax from Complete Problems

If a complexity class $\mathcal{C}$ has a complete problem L, it is a *syntactic* class.

We can enumerate all AC^0-computable functions: $(f_i)_{i\in\omega}$.

Let M_i be the machine that on input x computes $f_i(x)$ and checks whether it is in L.

It is an open question whether either of NP $\cap$ co-NP or BPP has complete problems.

Complete Problems from Syntax

Under certain conditions, there is a converse to the above. That is, from a syntactic characterisation, we can derive a complete problem.

For instance, if there is a recursive set of triples (M, M', p) with $L(M) = L(M')$ that indexes NP $\cap$ co-NP, then the class has complete problems.

Similarly, if there is a recursive set of *well-formed* machines (M, p) that indexes BPP, then the class has complete problems.

On the other hand, there are complexity classes, such as PH and PolyLogSpace which have syntactic characterisations but no complete problems.

Constructing Complete Problems

Given an enumerable index set for BPP of the form $(M_i, p_i)_{i \in \omega}$ or for NP $\cap$ co-NP of the form $(M_i, M_i', p_i)_{i \in \omega}$, we can construct a complete problem.

The general form of the construction is:

$$\{(i, x, 1^{p_i(|x|)}) \mid M_i \text{ accepts } x \text{ within } p_i(|x|) \text{ steps}\}$$

Graph Problems

Consider decision problems where the input is a graph: *Connectedness*, *3-Colouring*, *Hamiltonicity*.

We can encode graphs as strings over $\{0, 1\}$ by, for instance, enumerating the *adjacency matrix*.

There are up to $n!$ distinct encodings of a given n vertex graph G.

For $x, y \in \{0, 1\}^*$ write $x \sim y$ to indicate that they are encodings of the same graph.

And, for a language L, we say it is $\sim$-invariant if

$$x \in L \text{ and } x \sim y \quad \Rightarrow \quad y \in L.$$

Invariant Complexity Classes

Let inv-NP and inv-P be the classes of all languages that are in NP and P respectively and are $\sim$-invariant.

inv-NP is indexed by the set of pairs (M, p) where M is a nondeterminsitic Turing machine, p is a polynomial *and* the language accepted by M when clocked by p is $\sim$-invariant.

The invariance condition is undecidable.

Fagin's theorem tells us that, nonetheless, inv-NP has an *effective syntax*. It is indexed by machines obtained from existential second-order sentences.

inv-P

The open question, posed by Chandra and Harel, is whether there is an effective syntax for inv-P.

This was formalised by Gurevich (1988), who conjectured a negative answer.

inv-P has an effective syntax if there is a recursive set I and a Turing machine M such that

- on input $i \in I$, M produces the code for a machine $M(i)$ and a polynomial p_i
- $M(i)$, clocked by p_i is accepts a language that is $\sim$-invariant.
- for each language $L \in$ inv-P, there is an i such that $M(i)$ clocked by p_i accepts L.

Complete Problems

It is known that inv-P has an effective syntax *if, and only if,* there are graph problems complete under FO reductions.

(D. 1995)

FO-reductions (i.e. reductions definable in first-order logic) are AC^0-reductions and necessarily $\sim$-invariant.

Two Possible Worlds

Either

- there is no effective syntax for inv-P
- there is no classification possible of polynomial-time graph problems
- there is an *inexhaustible* supply of efficient algorithmic techniques to be discovered
- P $\neq$ NP

Or,

- there is an effective syntax for inv-P
- there is a P-complete graph problem under FO-reductions
- all polynomial-time graph problems can be solved by easy variations of one algorithm.