

## Week 3

# Extensions; Computability & Complexity

Last week's lecture ended with a mathematical explanation of why modal logic (and the modal fragment) are interesting. This week we briefly discuss an alternative answer: it's interesting because they work. We will do so by considering a number of languages.

### 3.1 Modal Extensions and Variations

The second reason why it's special: because it fits 'application' needs. Many people have found a variety of modeling and computational uses for a wide range of modal and modal-like languages. Here, we mention some of the most prominent ones.

#### 3.1.1 Multi-Modal Logic

Three readings of diamond and box have been extremely influential. First,  $\diamond\phi$  can be read as 'it is *possibly* the case that  $\phi$ .' Under this reading,  $\Box\phi$  means 'it is not possible that not  $\phi$ ,' that is, '*necessarily*  $\phi$ ,' and examples of formulas we would probably regard as correct principles include all instances of  $\Box\phi \rightarrow \diamond\phi$  ('whatever is necessary is possible') and all instances of  $\phi \rightarrow \diamond\phi$  ('whatever is, is possible'). The status of other formulas is harder to decide. Should  $\phi \rightarrow \Box\diamond\phi$  ('whatever is, is *necessarily* possible') be regarded as a general truth about necessity and possibility? Should  $\diamond\phi \rightarrow \Box\diamond\phi$  ('whatever is possible, is necessarily possible')? Are any of these formulas linked by a modal notion of logical consequence, or are they independent claims about necessity and possibility? These are difficult (and historically important) questions.

Second, in *provability logic*  $\Box\phi$  is read as 'it is *provable* (in some arithmetical theory) that  $\phi$ .' A central theme in provability logic is the search for a complete axiomatization of the provability principles that are valid for various arithmetical theories (such as Peano Arithmetic). The *Löb* formula  $\Box(\Box p \rightarrow p) \rightarrow \Box p$  plays a key role here.

Third, in *epistemic logic* the basic modal language is used to reason about knowledge, though instead of writing  $\Box\phi$  for 'the agent knows that  $\phi$ ' it is usual to write  $K\phi$ . Given that we are talking about knowledge (as opposed to, say, belief or rumor), it seems natural to view all instances of  $K\phi \rightarrow \phi$  as true: if the agent really *knows* that  $\phi$ , then  $\phi$  must hold. On the other hand (assuming that the agent is not omniscient) we would regard  $\phi \rightarrow K\phi$  as false. But the legitimacy of other principles is harder to judge (if an agent knows that  $\phi$ , does she know that she knows it?).

For the latter, epistemic reading it is natural to consider languages with multiple boxes  $K_a, K_b, \dots$ , corresponding to multiple agents  $a, b, \dots$ . This would allow us to talk about one agent's knowledge

about another agent:  $K_a p \rightarrow K_b p$  ( $b$  knows whatever  $a$  knows) or  $K_a K_b K_a p$  ( $a$  knows that  $b$  knows that  $a$  knows  $p$ ) — a great tool for specifying scripts for soap operas!

Let's make things more precise. Let  $A$  be some set of indices,  $A = a_0, a_1, \dots$ . Then, formulas of the multi-modal language (over  $A$ ) are given by the following rule:

$$\text{FORMS} := \top \mid p_i \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle \phi,$$

while  $[a]\phi$  is shorthand for  $\neg\langle a \rangle\neg\phi$ .

Multi-modal formulas are interpreted on relational structures of the form  $(W, \{R_a \mid a \in A\}, V)$ , that is: on labeled transition systems. The only new thing is that each modal operator  $\langle a \rangle$  is interpreted using its own binary relation  $R_a$ :  $\mathcal{M}, w \Vdash \langle a \rangle \phi$  iff for some  $v$  with  $R_a wv$  we have  $\mathcal{M}, v \Vdash \phi$ .

### 3.1.2 Temporal Logic

Temporal logic is a special kind of multi-modal logic with two modal operators, exploring a single binary relation. Arthur Prior founded temporal logic (or as he called it, *tense logic*) in the early 1950s. He invented the basic temporal language and many other temporal languages, both modal and non-modal.

The basic temporal language is built using a set of unary operators  $\langle F \rangle$ ,  $\langle P \rangle$ . The intended interpretation of a formula  $\langle F \rangle \phi$  is ‘ $\phi$  will be true at some *Future* time,’ and the intended interpretation of  $\langle P \rangle \phi$  is ‘ $\phi$  was true at some *Past* time.’ This language is called the *basic temporal language*, and it is the core language underlying a branch of modal logic called *temporal logic*. It is traditional to write  $\langle F \rangle$  as  $F$  and  $\langle P \rangle$  as  $P$ , and their duals are written as  $G$  and  $H$ , respectively. (The mnemonics here are: ‘it is always *Going* to be the case’ and ‘it always *Has* been the case.’)

We can express many interesting assertions about time with this language. For example,  $P\phi \rightarrow GP\phi$ , says ‘whatever has happened will always have happened,’ and this seems a plausible candidate for a general truth about time. On the other hand, if we insist that  $F\phi \rightarrow FF\phi$  must always be true, it shows that we are thinking of time as *dense*: between any two instants there is always a third. And if we insist that  $GFp \rightarrow FGP$  (the *McKinsey formula*) is true, for all propositional symbols  $p$ , we are insisting that atomic information true somewhere in the future eventually settles down to being always true. (We might think of this as reflecting a ‘thermodynamic’ view of information distribution.)

The basic temporal language has two unary operators  $F$  and  $P$ . Thus, according to our earlier discussion on multi-modal logic, models for this language consist of a set bearing two binary relations,  $R_F$  (the into-the-future relation) and  $R_P$  (the into-the-past relation), which are used to interpret  $F$  and  $P$  respectively. However, given the intended reading of the operators, most such models are inappropriate: clearly we ought to insist on working with models based on frames in which  $R_P$  is the *converse* of  $R_F$  (that is, frames in which  $\forall xy (R_F xy \leftrightarrow R_P yx)$ ).

Let us denote the converse of a relation  $R$  by  $\check{R}$ . We will call a frame of the form  $(T, R, \check{R})$  a *bidirectional frame*, and a model built over such a frame a *bidirectional model*. From now on, we will only interpret the basic temporal language in bidirectional models. That is, if  $\mathcal{M} = (T, R, \check{R}, V)$  is a bidirectional model then:

$$\begin{aligned} \mathcal{M}, t \Vdash F\phi & \text{ iff } \exists s (Rts \wedge \mathcal{M}, s \Vdash \phi) \\ \mathcal{M}, t \Vdash P\phi & \text{ iff } \exists s (\check{R}ts \wedge \mathcal{M}, s \Vdash \phi). \end{aligned}$$

But of course, once we've made this restriction, we don't need to mention  $\check{R}$  explicitly any more: once  $R$  has been fixed, its converse is fixed too. That is, we are free to interpret the basic temporal

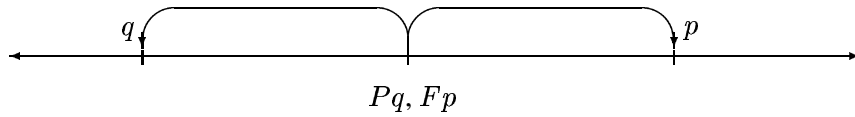
languages on frames  $(T, R)$  for the basic modal language using the clauses

$$\begin{aligned} \mathcal{M}, t \Vdash F\phi & \text{ iff } \exists s (Rts \wedge \mathcal{M}, s \Vdash \phi) \\ \mathcal{M}, t \Vdash P\phi & \text{ iff } \exists s (Rst \wedge \mathcal{M}, s \Vdash \phi). \end{aligned}$$

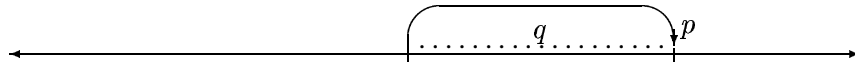
These clauses clearly capture a crucial part of the intended semantics:  $F$  looks forward along  $R$ , and  $P$  looks backwards along  $R$ . Of course, our models will only start looking genuinely *temporal* when we insist that  $R$  has further properties (notably transitivity, to capture the flow of time), but at least we have pinned down the fundamental interaction between the two modalities.

The binary until operator  $U(\phi, \psi)$  allows to do two things: to claim the existence of a future point satisfying a formula  $\phi$ , and to insist that the points in between now and that future point all satisfy a condition  $\psi$ . This is best understood using a picture.

The  $F$  and  $P$  operators allow us to say things like ‘Something good will happen’ and ‘Something bad has happened.’



But in several application areas this is not enough. For example, in the semantics of concurrent programs one often needs to be able to express properties of executions of programs that have the general format ‘Something good is going to happen, *and until that time nothing bad will happen.*’



Such properties are sometimes called *guarantee properties* in the computational literature. To state them, the binary *until* operator  $U$  can be used; its satisfaction definition reads:

$$t \Vdash U(\phi, \psi) \text{ iff there is a } v > t \text{ such that } v \Vdash \phi \text{ and for all } s \text{ with } t < s < v: s \Vdash \psi.$$

The mirror image of  $U$  is the *since* operator  $S$ :

$$t \Vdash S(\phi, \psi) \text{ iff there is a } v < t \text{ such that } v \Vdash \phi \text{ and for all } s \text{ with } v < s < t: s \Vdash \psi.$$

That’s the basic idea. The set of  $S, U$ -formulas is built up from a collection  $\Phi$  of proposition letters, the usual boolean connectives, and the *binary* operators  $S$  and  $U$ . The *mirror image* of a formula  $\phi$  is obtained by simultaneously substituting  $S$  for  $U$  and  $U$  for  $S$  in  $\phi$ .

### 3.1.3 Propositional Dynamic Logic

Another important branch of modal logic, again involving only unary modalities, is *propositional dynamic logic*. PDL, the language of propositional dynamic logic, has an infinite collection of diamonds. Each of these diamonds has the form  $\langle \pi \rangle$ , where  $\pi$  denotes a (non-deterministic) *program*. The intended interpretation of  $\langle \pi \rangle \phi$  is ‘some terminating execution of  $\pi$  from the present state leads to a

state bearing the information  $\phi$ .’ The dual assertion  $[\pi]\phi$  states that ‘every execution of  $\pi$  from the present state leads to a state bearing the information  $\phi$ .’

So far, there’s nothing really new — but a simple idea is going to ensure that PDL is highly expressive: we will make the inductive structure of the programs explicit in PDL’s syntax. Complex programs are built out of basic programs using some repertoire of program constructors. By using diamonds which reflect this structure, we obtain a powerful and flexible language.

Let us examine the core language of PDL. Suppose we have fixed some set of basic programs  $a$ ,  $b$ ,  $c$ , and so on (thus we have basic modalities  $\langle a \rangle$ ,  $\langle b \rangle$ ,  $\langle c \rangle$ , ... at our disposal). Then we are allowed to define complex programs  $\pi$  (and hence, modal operators  $\langle \pi \rangle$ ) over this base as follows:

**choice:** if  $\pi_1$  and  $\pi_2$  are programs, then so is  $\pi_1 \cup \pi_2$ .

The program  $\pi_1 \cup \pi_2$  (non-deterministically) executes  $\pi_1$  or  $\pi_2$ .

**composition:** if  $\pi_1$  and  $\pi_2$  are programs, then so is  $\pi_1 ; \pi_2$ .

This program first executes  $\pi_1$  and then  $\pi_2$ .

**iteration:** if  $\pi$  is a program, then so is  $\pi^*$ .

$\pi^*$  is a program that executes  $\pi$  a finite (possibly zero) number of times.

For the collection of diamonds this means that if  $\langle \pi_1 \rangle$  and  $\langle \pi_2 \rangle$  are modal operators, then so are  $\langle \pi_1 \cup \pi_2 \rangle$ ,  $\langle \pi_1 ; \pi_2 \rangle$  and  $\langle \pi_1^* \rangle$ . This notation makes it straightforward to describe properties of program execution. Here is a fairly straightforward example. The formula  $\langle \pi^* \rangle \phi \leftrightarrow \phi \vee \langle \pi ; \pi^* \rangle \phi$  says that a state bearing the information  $\phi$  can be reached by executing  $\pi$  a finite number of times if and only if either we already have the information  $\phi$  in the current state, or we can execute  $\pi$  once and then find a state bearing the information  $\phi$  after finitely many more iterations of  $\pi$ . Here’s a far more demanding example:

$$[\pi^*](\phi \rightarrow [\pi]\phi) \rightarrow (\phi \rightarrow [\pi^*]\phi).$$

This is *Seegerberg’s axiom* (or the *induction axiom*) and the reader should try working out what exactly it is that this formula says.

If we confine ourselves to these three constructors (and in this book for the most part we do) we are working with a version of PDL called *regular PDL*. (This is because the three constructors are the ones used in Kleene’s well-known analysis of regular programs.) However, a wide range of other constructors have been studied. Here are two:

**intersection:** if  $\pi_1$  and  $\pi_2$  are programs, then so is  $\pi_1 \cap \pi_2$ .

The intended meaning of  $\pi_1 \cap \pi_2$  is: execute both  $\pi_1$  and  $\pi_2$ , in parallel.

**test:** if  $\phi$  is a formula, then  $\phi?$  is a program.

This program tests whether  $\phi$  holds, and if so, continues; if not, it fails.

To flesh this out a little, the intended reading of  $\langle \pi_1 \cap \pi_2 \rangle \phi$  is that if we execute both  $\pi_1$  and  $\pi_2$  in the present state, then there is at least one state reachable by both programs which bears the information  $\phi$ . This is a natural constructor for a variety of purposes.

The key point to note about the test constructor is its unusual syntax: it allows us to make a modality out of a formula. Intuitively, this modality accesses the *current* state if the current state satisfies  $\phi$ . On its own such a constructor is uninteresting ( $\langle \phi? \rangle \psi$  simply means  $\phi \wedge \psi$ ). However, when other constructors are present, it can be used to build interesting programs. For example,  $(p? ; a) \cup (\neg p? ; b)$  is ‘if  $p$  then  $a$  else  $b$ .’

Let's turn to models for PDL now. Now, the language of PDL is just a multi-modal language, so a model for this language has the form

$$(W, \{R_\pi \mid \pi \text{ is a program}\}, V).$$

That is, a model is a labeled transition system together with a valuation. However, given our reading of the PDL operators, most of these models are uninteresting. As with the basic temporal language, we must insist on working with a class of models that does justice to our intentions.

Now, there is no problem with the interpretation of the basic programs: any binary relation can be regarded as a transition relation for a non-deterministic program. Of course, if we were particularly interested in *deterministic* programs we would insist that each basic program be interpreted by a partial function, but let us ignore this possibility and turn to the key question: which relations should interpret the structured modalities? Given our readings of  $\cup$ ,  $;$  and  $*$ , as choice, composition, and iteration, it is clear that we are only interested in relations constructed using the following inductive clauses:

$$\begin{aligned} R_{\pi_1 \cup \pi_2} &= R_{\pi_1} \cup R_{\pi_2} \\ R_{\pi_1 ; \pi_2} &= R_{\pi_1} \circ R_{\pi_2} (= \{(x, y) \mid \exists z (R_{\pi_1} xz \wedge R_{\pi_2} zy)\}) \\ R_{\pi_1^*} &= (R_{\pi_1})^*, \text{ the reflexive transitive closure of } R_{\pi_1}. \end{aligned}$$

These inductive clauses completely determine how each modality should be interpreted. Once the interpretation of the basic programs has been fixed, the relation corresponding to each complex program is fixed too. This leads to the following definition.

Suppose we have fixed a set of basic programs. Let  $\Pi$  be the smallest set of programs containing the basic programs and all programs constructed over them using the regular constructors  $\cup$ ,  $;$  and  $*$ . Then a *regular frame for  $\Pi$*  is a labeled transition system  $(W, \{R_\pi \mid \pi \in \Pi\})$  such that  $R_a$  is an arbitrary binary relation for each basic program  $a$ , and for all complex programs  $\pi$ ,  $R_\pi$  is the binary relation inductively constructed in accordance with the previous clauses. A *regular model for  $\Pi$*  is a model built over a regular frame; that is, a regular model is regular frame together with a valuation. When working with the language of PDL over the programs in  $\Pi$ , we will only be interested in regular models for  $\Pi$ , for these are the models that capture the intended interpretation.

What about the  $\cap$  and  $?$  constructors? Clearly the intended reading of  $\cap$  demands that  $R_{\pi_1 \cap \pi_2} = R_{\pi_1} \cap R_{\pi_2}$ . As for  $?$ , it is clear that we want the following definition:

$$R_{\phi?} = \{(x, y) \mid x = y \text{ and } y \Vdash \phi\}.$$

This is indeed the clause we want, but note that it is rather different from the others: it is not a *frame* condition. Rather, in order to determine the relation  $R_{\phi?}$ , we need information about the *truth* of the formula  $\phi$ , and this can only be provided at the level of *models*.

### 3.1.4 Description Logic

To be supplied.

## 3.2 Computability and Complexity

We recall the basic ideas of computability theory (the study of which problems are and which problems are not computationally solvable), and provide some background information on complexity theory (the study of the computational resources required to solve complexity problems).

### 3.2.1 Computability and Uncomputability

To prove theorems about computability — and in particular to prove that some problem is *not* computable — we need a mathematical model of computability. One of the most widely used models is the *Turing machine*: it is a device which manipulates symbols written on a tape. The symbols are taken from some alphabet fixed in advance (often the alphabet simply consists of the two symbols 0 and 1). The tape is subdivided into squares, and only one symbol can be written on each square (squares containing no symbols are called blank). The tape is used to present input, to receive output, and acts as working memory. The tape is assumed to be infinitely long in both directions (so no finite upper bound on the amount of working memory is assumed).

Turing machines scan the squares of such tapes and act on the information they see; they can only scan one square at a time. A Turing machine has a finite number of internal states, and a finite number of rules which tell it what to do when it is in a certain state scanning a certain symbol. Turing machines can perform three basic actions: (1) move to the square immediately to the left of the square they are currently scanning, (2) move to the square immediately to the right of the square they are currently scanning, or (3) write a symbol (from the alphabet) on the square currently being scanned (thereby overwriting any symbol already written on that square). In addition to specifying which of these three actions will be performed, the rules also specify which internal state the Turing machine is to move into on completing the action.

**Definition 3.1 (Turing machines)** A Turing machine is a 5-tuple  $(S, s, H, \Sigma, \rho)$  where  $S$  is a finite set of states,  $s \in S$  is the initial state,  $H \subseteq S$  is the set of halting states,  $\Sigma$  (the alphabet) is a finite set of symbols, and  $\rho$  is a function from  $(S \setminus H) \times \Sigma$  to  $S \times (\Sigma \cup \{left, right\})$ .  $\dashv$

For example, the rule *if you are in state 57 scanning the symbol 1, move one square to the left and go into state 14* amounts to saying that  $\rho(57, 1) = (14, left)$ . Since  $\rho$  is a function, the action of such a machine is *deterministic*: when the machine is put in the initial state scanning some tape, what it does (if it does anything) is fixed.

Let  $f$  be a function, and suppose we have fixed some convention about how the elements of the domain and range of the function are to be represented. Then  $f$  is *computable* (or *recursive*) if there is a Turing machine that when given (the representation of) any item  $x$  in the domain of  $f$  will halt after finitely many steps, leaving on an otherwise blank tape (the representation of)  $f(x)$ .

We can also use Turing machines to provide yes/no answers to problems. Many logical problems — for example, is some formula  $\phi$  satisfiable or not — are of this type. Given a suitable encoding conventions, some strings over the alphabet represent problem instances for which the answer is *yes*, while other represent problem instances for which the answer is *no*. A problem is *computable* (or *recursive*, or *decidable*) if there is a Turing machine which when given (the representation of) any instance of the problem, halts after finitely many steps leaving the (representation of) the correct answer on an otherwise blank tape. Such Turing machines essentially provide answers to set membership problems: a *yes* answer means that the input belongs to some set of interest (for example, the set of satisfiable formulas) while a *no* means it does not. Thus it is common to talk of computable (or recursive, or decidable) *sets*. Another important notion is that of a *recursively enumerable* set. A set is recursively enumerable (r.e.) if there is a Turing machine which successively writes, on an otherwise blank tape, all and only the members of the set. If the set is infinite, this listing process will never finish — but after some finite time, any given element of an r.e. set will eventually be listed.

It is common practice to identify problems with the set of those strings of symbols that provide the answer *yes* to the problem. For once an alphabet has been fixed, each subset of the set of all finite

strings over the alphabet can be regarded as the encoding of the problem. This abstract perspective is a convenient way of stating abstract computability and complexity results.

An important variation is the use of *non-deterministic* Turing machines. The action of such machines is not fixed by the symbol it is scanning and the state it is in: for any such combination, it may have a (finite) range of options. (Formally, we drop the requirement that the  $\rho$  of Definition 3.1 be a function and let it be an arbitrary relation.) We think of such a machine as following all the options allowed by  $\rho$  simultaneously, and say that such a machine solves a problem if at least one such computation path halts leaving the correct answer on an otherwise blank tape. The beauty of non-determinism is that it factors out search. Many problems involve require us to find a candidate solution and then see if it works, and if not, to look for another candidate, and so on. This process may be the major computational overhead. Non-deterministic machines abstract away from this. But as far as *computability* is concerned, non-determinism adds nothing: if a function (or problem) is computable using a non-deterministic Turing machine, it is also computable using a deterministic Turing machine, for we can (laboriously) work through all possible choices.

Such observations gives rise to Church's thesis:

**Definition 3.2 (Church's Thesis)** A function is computable (a problem decidable) precisely when it can be computed (solved) using a Turing machine.  $\dashv$

Readers with programming experience may like to view Church's Thesis as saying that computable functions and problems are those which can be calculated/solved by writing a program in their favorite programming language when no limitations are placed on memory or execution time.

The most important benefit of having a robust definition of computability is that it gives us a way of proving that some function or problem is *not* decidable. And many — indeed most — functions and problems are *not* computable. For let  $M$  be a set of natural numbers. Is each such  $M$  decidable? A simple cardinality argument shows that the answer is *no*. Every Turing machine is a finite function over a finite set of states and a finite alphabet. It follows that there are only countably many Turing machines — but there are uncountably many  $M$ , so they can't all be computable.

It is usually convenient to show problems are undecidable via reductions:

**Definition 3.3** Let  $\Sigma$  be an alphabet and let  $L_1, L_2 \subseteq \Sigma^*$  be problems. A reduction from  $L_1$  to  $L_2$  is a computable function  $\tau : \Sigma^* \rightarrow \Sigma^*$  such that  $s \in L_1$  iff  $\tau(s) \in L_2$ .  $\dashv$

**Proposition 3.4** Let  $L_1, L_2 \subseteq \Sigma^*$  be problems, and  $\tau$  be a reduction from  $L_1$  to  $L_2$ . If  $L_1$  is undecidable, then so is  $L_2$ .

Nowadays a vast range of problems are known to be undecidable, and we can try to prove undecidability results by reduction from any one of these problems.

One final remark: not all undecidable problems are alike. There is a precise sense in which some are worse than others. The key idea is to equip Turing machines with *oracles*. A Turing machine equipped with an oracle is allowed to temporarily halt in the middle of some computation, consult the oracle, and proceed with its computation taking the oracle's answer into account.

Oracles provide answers to *undecidable* problems (an oracle that provided answers to decidable problems would offer nothing new: it could always be replaced by a Turing machine). They are basically a mathematical abstraction which allow us to remove the limitation to finitary computation inherent in Church's thesis. It is common to specify what oracles can do in logical terms — for example, we might imagine we have an Turing machine hooked to an oracle that is able to determine whether an arbitrary second-order sentence has a model or not. It turns out that undecidable problems

are not all the same: when we measure their difficulty with respect to the oracles required to solve them, there is a whole hierarchy of difficulty. A problem that is not merely undecidable, but requires the help of some such oracle to solve it is called *highly undecidable*.

### 3.2.2 Complexity theory

Complexity theory studies the computational resources required to solve (decidable) problems. The two main resources studied are *time* (the number of computation steps required) and *space* (the amount of memory required). Both time required and space required are measured as functions of the length of the input.

Ideally, complexity theory would give us a precise bound on the resources required to solve any problem that interested us. Unsurprisingly, this goal is far too ambitious. Instead, complexity theory classifies problems into various classes. In this course we mention the classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq NEXPTIME \quad (*)$$

and we devote a lot of attention to NP, PSPACE, and EXPTIME. Before defining these classes, some general remarks. It is currently unknown whether the inclusions in (\*) are strict or not. It is widely conjectured that they are, but nobody has been able to prove (or disprove) any of these strict inclusions. All we know for sure is that  $P \neq EXPTIME$ .

Second, although a problem that belongs to any of these classes is decidable, P (the class at the bottom of this putative hierarchy) is widely taken to be the class of problems that are *tractable*, or *efficiently solvable*.

**Definition 3.5** A deterministic Turing machine is *polynomially time bounded* if there is a polynomial  $p(n)$  such that the machine always halts after at most  $p(n)$  steps, where  $n$  is the length of the input. A problem is *solvable in polynomial time* (a function  $f$  is solvable in polynomial time) if there is a polynomially bounded Turing machine that solves it (that computes it). The class of all problems solvable in polynomial time is called P. A problem is called *tractable* if it belongs to P.  $\dashv$

Identifying tractable problems with those in P isn't unproblematic, but it has proved useful. If a problem is solvable in polynomial time, then typically the polynomial is of low degree.

### Hardness and Completeness

In order to define the complexity classes of interest, we need some additional concepts. We have already met the idea of reducing one problem to another (see Definition 3.3). To make further progress, we need the notion of *tractably* reducing one problem to another. As we have identified 'tractable' with 'computable in polynomial time', the following notion is what we require:

**Definition 3.6 (Polytime reductions)** Let  $L_1, L_2 \subseteq \Sigma^*$  be problems. A polynomial time computable function  $\tau : \Sigma^* \rightarrow \Sigma^*$  is called a polynomial time reduction (or: a polytime reduction) from  $L_1$  to  $L_2$  if for each  $s \in \Sigma^*$  we have that  $s \in L_1$  iff  $\tau(s) \in L_2$ .  $\dashv$

A polytime reduction from  $L_1$  to  $L_2$  is essentially a *tractable* way of compiling problem  $L_1$  down to problem  $L_2$ . It follows that if  $L_2$  is solvable in polynomial time (that is, if  $L_2$  is tractable), then so is  $L_1$ : to test whether a string  $x$  is in  $L_1$ , simply compute  $\tau(x)$  (this compilation step is polynomial time computable) and then test whether  $\tau(x) \in L_2$  (which by assumption is polynomial time solvable). As  $x \in L_2$  iff  $\tau(x) \in L_2$ , and as the composition of two polynomials is a polynomial, we have efficiently

computed an answer to our original problem. On the other hand, if  $L_1$  is *not* solvable in polynomial time, then neither is  $L_2$ , as the reader should verify. Summing up: if there is a polytime reduction from  $L_1$  to  $L_2$ , then  $L_2$  is *at least as hard* as  $L_1$ , and this observation leads us to the following fundamental definition:

**Definition 3.7 (Hardness and completeness)** Let  $C$  be a class of problems. A problem  $L$  is *C-hard* (with respect to polynomial time reductions) if every problem in  $C$  is polynomial time reducible to  $L$ . A problem  $L$  is *C-complete* if it is  $C$ -hard and moreover  $L \in C$ . That is, the  $C$ -complete problems are the hardest problems in  $C$ .  $\dashv$

### The class P

This fundamental class does not play a direct role in the book, for it is widely believed that the problem of deciding whether a formula of classical propositional logic is satisfiable is *not* in P. (No proof of this is known — it is one of the best-known open problems in theoretical computer science.) As the modal logics discussed in this book contain classical propositional logic as a subpart, their satisfiability problems probably don't lie in P either.

### The class NP

There are many naturally occurring problems which do not seem to belong to P but which can be solved efficiently using a non-deterministic Turing machine.

**Definition 3.8** A non-deterministic Turing machine is polynomially time bounded if there is a polynomial  $p(n)$  such that no computation of the machine continues for more than  $p(n)$  steps where  $n$  is the length of the input. NP is the class of all problems decided by a polynomially bounded nondeterministic machine.  $\dashv$

The problems that seem *not* to be in P but which *are* in NP typically involve search. The classic example is the satisfiability problem for propositional logic: given a propositional formula  $\phi$ , is there an assignment of truth values (0 and 1) to its propositional letters that makes the formula evaluate to 1? No deterministic polynomial time algorithm for propositional satisfiability is known, and it is widely believed that none exists. But it is easy to design an NP algorithm to solve propositional satisfiability.

Does this mean that propositional satisfiability is really an easy problems to solve? Unfortunately, no. The only known way of implementing non-determinism is to simulate it on a deterministic Turing machine. All known simulations require exponential time to perform, and it is widely believed (though not proved) that no efficient simulation exists. Non-deterministic Turing machines are probably not a realistic model of efficient computation.

But non-determinism has proved to be a very useful way of thinking about problems consisting of a *search* for a solution, followed by a *verification* step that can be conducted in deterministic polynomial time. An extraordinary range of interesting problems have this general profile, and by reducing the search to a single non-deterministic step, we see that such problems belong to NP.

Now for a more demanding question: are there any NP-hard problems? The celebrated Cook-Levine theorem tells us that there are:

**Theorem 3.9 (Cook-Levine Theorem)** *The propositional satisfiability problem is NP-complete.*

*Proof.* An elegant proof is given on pages 309–317 of the second edition of Lewis and Papadimitriou [Lewis and Papadimitriou, 1981].  $\dashv$

Once we have shown that one problem is NP-complete, it becomes much easier to show that other problems are NP-complete. Given a problem  $L$  which we suspect to be NP-complete, all we have to do is (a) show that it is in NP, and (b) show that some problem known to be NP-hard is polynomial time reducible to  $L$ . In this book, showing point (b) is trivial: all the logics we are interested in are extensions of classical propositional logic, so NP-hardness is immediate by Cook's Theorem.

The classes NP and coNP (that is, the class of problems whose complements are in NP) seem to have very different complexity profiles. A classic problem in coNP is the *validity* problem for propositional calculus — the problem of deciding whether *all* assignments of truth values satisfy a propositional formula. The validity problem is widely believed not to be in P. However (unlike the satisfiability problem) it doesn't seem to belong to NP either: because we need to consider all possible truth assignments, non-determinism doesn't seem to help us solve it. But we face another open problem here: although it is standardly conjectured that  $NP \neq coNP$ , no-one has been able to prove or disprove it.

### The class PSPACE

PSPACE is the complexity class of most relevance to modal logic. It is defined in terms of *deterministic* Turing machines.

**Definition 3.10** A deterministic Turing machine is *polynomially space bounded* if there is a polynomial  $p(n)$  such that no computation of the machine scans more than  $p(n)$  tape squares, where  $n$  is the length of the input. PSPACE is the class of all problems that are decided by a polynomially space bounded deterministic Turing machine.  $\dashv$

**Proposition 3.11**  $NP \subseteq PSPACE$ .

What's the intuition behind this theorem? As we have already remarked, a deterministic Turing machine can simulate a non-deterministic Turing machine, though it is widely believed that the simulation will in general run exponentially slower than the non-deterministic machine. Proposition 3.11 tells us for any problem in NP, it is always possible to carry out the simulation in such a way that there is no blow-up in *space* requirements. Roughly speaking, we work systematically through the search space, with the search for each item taking only polynomial space. When it's time to search for the next item, we reuse the same squares. There is a bookkeeping overhead (we need to keep track of where we are in the search space) but (with careful management) this can be done using a relatively small number of squares. So while the simulation may take a long time, we don't need much memory.

PSPACE is defined in terms of *deterministic* Turing machines. NPSPACE, the class of problems computable by non-deterministic polynomial space bounded Turing machines, is defined by rephrasing the definition in terms of non-deterministic Turing machines. Intriguingly, NPSPACE contains nothing new:

**Theorem 3.12 (Savitch's Theorem)**  $PSPACE = NPSPACE$ .

*Proof.* See Theorem 7.5 and the surrounding discussion in Papadimitriou [Papadimitriou, 1994].  $\dashv$

So if we want to show that a problem is in PSPACE, we can do so by showing that it is in NPSPACE.

Finally,  $PSPACE = coPSPACE$ . Why? Well, any deterministic Turing machine that decides a problem  $L$  in PSPACE can be converted to a machine that decides  $\bar{L}$  simply by flipping *yeses* to *nos* and vice versa. This invariance under complementation has nothing much to do with PSPACE: for any time or space class  $\mathcal{C}$  defined in terms of *deterministic* Turing machines,  $\mathcal{C} = co\mathcal{C}$ , as the 'switch the outputs' argument shows. (Note that this argument does not work with non-deterministic machines.)

**The class EXPTIME**

EXPTIME is defined in terms of deterministic Turing machines:

**Definition 3.13** A deterministic Turing machine is *exponentially time bounded* if there is a polynomial  $p(n)$  such that the machine always halts after at most  $2^{p(n)}$  steps, where  $n$  is the length of the input. A problem is *solvable in exponential time* if there is an exponentially time bounded Turing machine that solves it. The class of problems solvable in exponential time is called EXPTIME.  $\dashv$

EXPTIME-hard problems are intractable. There are problems in EXPTIME which provably do *not* belong to P (see, for example, Theorem 6.1.2 in the second edition of Lewis and Papadimitriou [Lewis and Papadimitriou, 1981]), hence any EXPTIME-hard problem is at least as hard as such intractable problems.

**The class NEXPTIME**

NEXPTIME is the class of problems solvable using an exponentially bounded *nondeterministic* Turing machine. Like NP algorithms, NEXPTIME algorithms have a ‘guess and check’ profile. The crucial difference is that guessed information may be exponentially large in the size of the input, thus the deterministic checking that follows may take exponentially many steps in the size of the input.

We won’t mention NEXPTIME much, but it is implicitly present: when modal logics are proved decidable using the finite model property, a NEXPTIME algorithm is usually being employed.

**An Overview of Satisfiability Costs**

The following table lists complexity results for the satisfiability problem for a number of modal logics:

	NP-complete	PSPACE-complete	EXPTIME-complete
Prop Log	×		
Basic Mod Log		×	
Multi-Mod Log		×	
Temp Log ( $F, P$ )		×	
Prop Dyn Log			×

**What is “Tractable”?**

Following general practice in computer science, and in complexity theory, we have defined tractable to mean ‘decidable in P’. But if a problem is decidable in polynomial time, this also means “directedness” and “no guessing” or “no pondering.” While, if we are interested in inference, it seems natural that we need to consider different situations. So asking for P in the setting of automated reasoning is surely too strong. For the logics that we will be considering, NP is always a lower bound, and we will usually be working with reasoning problems that are complete for PSPACE or EXPTIME.

But if a problem is *not* solvable in polynomial time, then (some instances of it) will be very hard to solve indeed. For example, if a problem requires resources exponential in the length of the input (for example,  $2^n$ , where  $n$  is the length of the input), then no algorithm is going to solve all instances of the problem efficiently: on some input, even for quite small values of  $n$ , the computation will not halt within the expected lifetime of the universe. So, from a practical point of view, what’s the difference between EXPTIME and undecidability? Usually undecidability is “bad behavior that comes easy”. That is, you can usually encode undecidable problems with simple means.

## Notes

[Goldblatt, 1987] is an accessible introduction to temporal and dynamic logics. An introduction to description logics may be found in [Donini *et al.*, 1996].

For detailed discussions of computability, see [Rogers, 1967] or [Odifreddi, 1989]. For accessible introductions to the subject, see [Boolos and Jeffrey, 1989], or [Cutland, 1980]. But the single most useful source is probably the (second edition of) [Lewis and Papadimitriou, 1981]; this introduces computability theory, and then goes on to treat computational complexity. For more on computational complexity, try [Garey and Johnson, 1979] and [Papadimitriou, 1994].

## Bibliography

- [Boolos and Jeffrey, 1989] G. Boolos and R. Jeffrey. *Computability and Logic*. Cambridge University Press, 1989.
- [Brewka, 1996] G. Brewka, editor. *Principles of Knowledge Representation*. Studies in Logic, Language and Information. CSLI Publications, 1996.
- [Cutland, 1980] N. Cutland. *Computability. An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
- [Donini *et al.*, 1996] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In Brewka [1996], pages 191–236.
- [Garey and Johnson, 1979] M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
- [Goldblatt, 1987] R. Goldblatt. *Logics of Time and Computation*, volume 7 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford University, 1987.
- [Lewis and Papadimitriou, 1981] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.
- [Odifreddi, 1989] P. Odifreddi. *Classical recursion theory*. North-Holland Publishing Company, 1989.
- [Papadimitriou, 1994] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [Rogers, 1967] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, 1967.