

## Chapter 2

# Background

In this chapter we define the formal machinery needed in the rest of the lecture for describing different planning problems and algorithms. We give the basic definitions related to the classical propositional logic, theory of computational complexity, and the definition of the transition system model that is the basis of most work on planning. The transition systems in this lecture are closely related to finite automata and transition systems in other areas of computer science.

### 2.1 Transition systems

The most important way of modeling the application underlying a planning problem is based on the notion of a *transition system*. A transition system consists of a set of *states*, which represent the world at a given instant, and a number of *actions* that describe the possible changes in the world that can be caused by the agent/robot/something. The states form the *state space*.

The actions are best understood as directed graphs with the states as the nodes.

Now a transition system is a 2-tuple  $\langle S, A \rangle$  where  $A$  is a finite set of actions  $a \subseteq S \times S$ .

In the beginning we consider deterministic actions only. An action  $a \in A$  is *deterministic* if and only if it is a (partial) function on  $S$ , that is, for every  $s \in S$  there is at most one  $s' \in S$  such that  $(s, s') \in a$ . For *nondeterministic* actions the number of successor states  $s'$  may be higher than one.

Later in Section 5.1 we will not just associate more than one successor state with a state, but a probability distribution on the states so that some of the successor states can be more likely than others.

#### 2.1.1 Incidence matrices

Graphs can be represented graphically, or in terms of incidence matrices  $M$  (adjacency matrices) in which element  $M_{i,j}$  indicates that a transition from state  $i$  to  $j$  is possible. We will later derive representations of transition systems as propositional formulae that are best understood as succinct representations of the kind of incidence matrices described here. Matrix operations like sum and product have counterparts as operations on propositional formulae, and they are used in some of the algorithms that we will discuss later.

Figure 2.1 depicts the transition graph of an action and the corresponding incidence matrix. The action can be seen to be deterministic because for every state there is at most one arrow going out of it, and each row of the matrix contains at most one non-zero element.

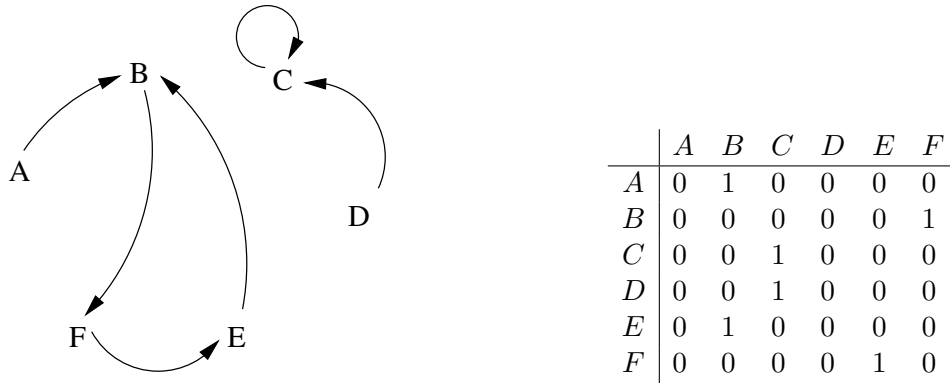


Figure 2.1: The transition graph and the incidence matrix of a deterministic action

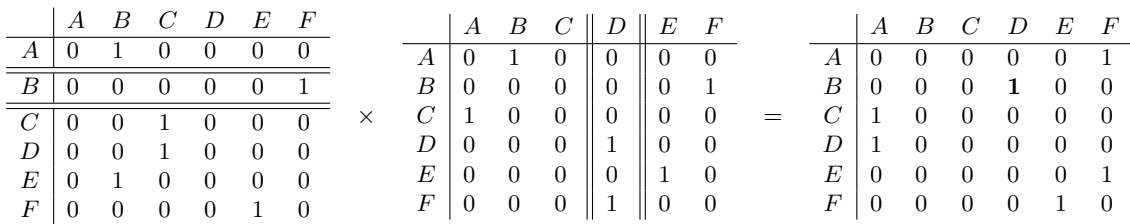


Figure 2.2: Matrix product corresponds to sequential composition.

For matrices  $M_1, \dots, M_n$  that represent the transition relations of actions  $a_1, \dots, a_n$ , the combined transition relation is  $M = M_1 + M_2 + \dots + M_n$ . The matrix  $M$  now tells whether a state can be reached from another state by at least one of the actions.

Here  $+$  is the usual matrix addition that uses the Boolean addition for integers 0 and 1, which is defined as  $0 + 0 = 0$ , and  $b + b' = 1$  if  $b = 1$  or  $b' = 1$ . Later in Chapter 5 we will use normal addition and interpret the matrix elements as probabilities of nondeterministic transitions.

Boolean addition is used because later in the presence of nondeterminism we could have 1 for both of two transitions from A to B and from A to C. Later, when the matrix elements represent transition probabilities, we will be using the ordinary arithmetic addition for real numbers.

### 2.1.2 Reachability as product of matrices

The incidence matrix corresponding to first taking action  $a_1$  and then  $a_2$  is  $M_1M_2$ . This is illustrated by Figure 2.2 The inner product of two vectors in the definition of matrix product corresponds to the reachability of a state from another state through all possible intermediate states.

Now we can compute for all pairs  $s, s'$  of states whether  $s'$  is reachable from  $s$  by a sequence of actions.

Let  $M$  be the matrix that is the (Boolean) sum of the matrices of the individual actions. Then define

$$\begin{aligned}
 R_0 &= I_{n \times n} \\
 R_i &= R_{i-1} + MR_{i-1} \text{ for all } i \geq 1
 \end{aligned}$$

Here  $n$  is the number of states and  $I_{n \times n}$  is the unit matrix of size  $n$ .

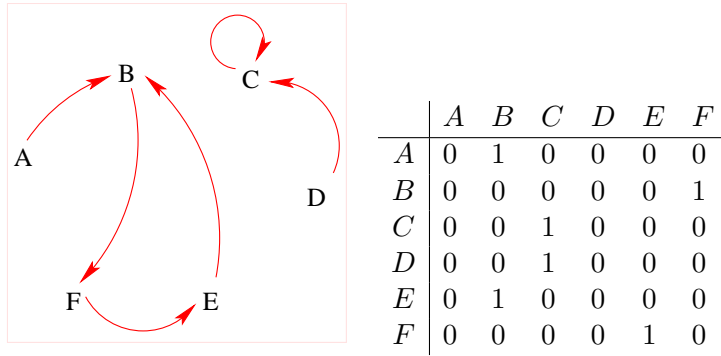


Figure 2.3: A transition graph and the corresponding matrix  $M$

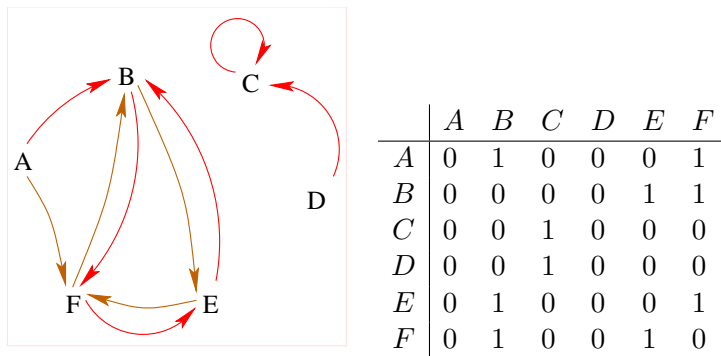


Figure 2.4: A transition graph extended with composed paths of length 2 and the corresponding matrix  $M + M^2$

This computation ends because every element that is 1 for some  $i$ , is 1 also for all  $j > i$ , and because of this monotonicity property there is a fixpoint by Tarski's fixpoint theorem. Matrix  $R_i$  represents reachability by  $i$  actions.

Matrix  $R_i = M^0 \cup M^1 \cup \dots \cup M^i$  represents reachability by  $i$  actions or less.

$R_i = R_j$  for some  $i \in \{1, \dots, n\}$  and all  $j \geq i$ .

## 2.2 Classical propositional logic

Let  $P$  be a set of atomic propositions. We define the set of propositional formulae inductively as follows.

1. For all  $p \in P$ ,  $p$  is a propositional formula.
2. If  $\phi$  is a propositional formula, then so is  $\neg\phi$ .
3. If  $\phi$  and  $\phi'$  are propositional formulae, then so is  $\phi \vee \phi'$ .
4. If  $\phi$  and  $\phi'$  are propositional formulae, then so is  $\phi \wedge \phi'$ .
5. The symbols  $\perp$  and  $\top$ , respectively denoting truth-values false and true, are propositional formulae.

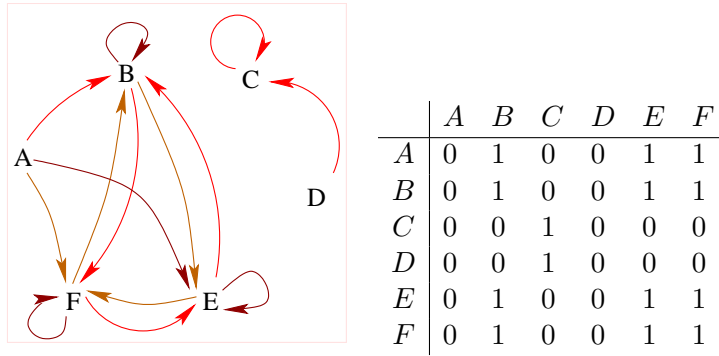


Figure 2.5: A transition graph extended with composed paths of length 3 and the corresponding matrix  $M + M^2 + M^3$

We define the implication  $\phi \rightarrow \phi'$  as an abbreviation for  $\neg\phi \vee \phi'$ , and the equivalence  $\phi \leftrightarrow \phi'$  as an abbreviation for  $(\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$ .

A valuation on  $P$  is a function  $v : P \rightarrow \{0, 1\}$ . Here 0 denotes false and 1 denotes true. For propositions  $p \in P$  we define  $v \models p$  if and only if  $v(p) = 1$ . Given a valuation of the propositions  $P$ , we can extend it to a valuation of all propositional formulae over  $P$  as follows.

1.  $v \models \neg\phi$  if and only if  $v \not\models \phi$
2.  $v \models \phi \vee \phi'$  if and only if  $v \models \phi$  or  $v \models \phi'$
3.  $v \models \phi \wedge \phi'$  if and only if  $v \models \phi$  and  $v \models \phi'$
4.  $v \models \top$
5.  $v \not\models \perp$

Computing the truth-value of a formula under a given valuation of propositional variables is polynomial time in the size of the formula by the obvious recursive procedure.

A propositional formula  $\phi$  is *satisfiable* (*consistent*) if there is at least one valuation  $v$  so that  $v \models \phi$ . Otherwise it is *unsatisfiable* (*inconsistent*). A propositional formula  $\phi$  is *valid* or a *tautology* if  $v \models \phi$  for all valuations  $v$ . We write this as  $\models \phi$ . A propositional formula  $\phi$  is a *logical consequence* of a propositional formula  $\phi'$ , written  $\phi' \models \phi$ , if  $v \models \phi$  for all valuations  $v$  such that  $v \models \phi'$ . A propositional formula that is a proposition  $p$  or a negated proposition  $\neg p$  for some  $p \in P$  is a *literal*. A formula that is a disjunction of literals is a *clause*.

### 2.2.1 Quantified Boolean formulae

There is an extension of the satisfiability and validity problems of the classical propositional logic that introduces quantification over the truth-values of propositional variables. Syntactically, *quantified Boolean formulae* (QBF) are defined like propositional formulae, but there are two new syntactic rules for the quantifiers.

6. If  $\phi$  is a formula and  $p \in P$ , then  $\forall p\phi$  is a formula.
7. If  $\phi$  is a formula and  $p \in P$ , then  $\exists p\phi$  is a formula.

The truth-value of these formulae is defined if the following two conditions are fulfilled.

- For every  $p \in P$  occurring in  $\phi$ , there is exactly one occurrence of  $\exists p$  or  $\forall p$  in  $\phi$ .
- All occurrences of  $p \in P$  are inside  $\exists p$  or  $\forall p$ .

Define  $\phi[\psi/x]$  as the formula obtained from  $\phi$  by replacing occurrences of the propositional variable  $x$  by  $\psi$ .

We define the truth-value of QBF by reducing them to ordinary propositional formulae without occurrences of propositional variables. The atomic formulae in these formulae are the constants  $\top$  and  $\perp$ . The truth-value of these formulae is independent of the valuation, and is recursively computed by the Boolean function associated with the connectives  $\vee$ ,  $\wedge$  and  $\neg$ .

**Definition 2.1 (Truth of QBF)** *A formula  $\exists x\phi$  is true if and only if  $\phi[\top/x] \vee \phi[\perp/x]$  is true. (Equivalently, if  $\phi[\top/x]$  is true or  $\phi[\perp/x]$  is true.)*

*A formula  $\forall x\phi$  is true if and only if  $\phi[\top/x] \wedge \phi[\perp/x]$  is true. (Equivalently, if  $\phi[\top/x]$  is true and  $\phi[\perp/x]$  is true.)*

*A formula  $\phi$  with an empty prefix (and consequently without occurrences of propositional variables) is true if and only if  $\phi$  is satisfiable (equivalently, valid: for formulae without propositional variables validity coincides with satisfiability.)*

**Example 2.2** The formulae  $\forall x\exists y(x \leftrightarrow y)$  and  $\exists x\exists y(x \wedge y)$  are true.

The formulae  $\exists x\forall y(x \leftrightarrow y)$  and  $\forall x\forall y(x \vee y)$  are false. ■

Notice that a QBF with only existential quantifiers is true if and only if the formula stripped from the quantifiers is satisfiable. Similarly, truth of QBF with only universal quantifiers coincides with the validity of the corresponding formulae without quantifiers.

Changing the order of two consecutive variables quantified by the same quantifier does not affect the truth-value of the formula. It is often useful to ignore the ordering in these cases, and view each quantifier as quantifying a set of formulae, for example  $\exists x_1x_2\forall y_1y_2\phi$ .

Quantified Boolean formulae are interesting because evaluating their truth-value is PSPACE-complete [Meyer and Stockmeyer, 1972], and several computational problems that presumably cannot be translated to the satisfiability of the propositional logic in polynomial time (assuming that  $\text{NP} \neq \text{PSPACE}$ ) can be efficiently translated to QBF.

### 2.2.2 Binary decision diagrams

Propositional formulae can be transformed to different normal forms. The most well-known normal forms are the conjunctive normal form (CNF) and the disjunctive normal form (DNF). Formulae in conjunctive normal form are conjunctions of disjunctions of literals, and in disjunctive normal form they are disjunctions of conjunctions of literals. For every propositional formula there is a logically equivalent one in both of these normal forms. However, the formula in normal form may be exponentially bigger.

Normal forms are useful for at least two reasons. First, certain types of algorithms are easier to describe when assumptions of the syntactic form of the formulae can be made. For example, the resolution rule which is the basis of many theorem-proving algorithms, is defined for formulae in the conjunctive normal form only (the clausal form). Defining resolution for non-clausal formulae is more difficult.

The second reason is that certain computational problems can be solved more efficiently for formulae in normal form. For example, testing the validity of propositional formulae is in general co-NP-hard, but if the formulae are in CNF then it is polynomial time: just check whether every conjunct contains both  $p$  and  $\neg p$  for some proposition  $p$ .

Transformation into a normal form in general is not a good solution to any computationally intractable problem like validity testing, because for example in the case of CNF, polynomial-time validity testing became possible only by allowing a potentially exponential increase in the size of the formula.

However, there are certain normal forms for propositional formulae that have proved very useful in various types of reasoning needed in planning and other related areas, like model-checking in computer-aided verification.

In this section we discuss (ordered) binary decision diagrams (BDDs) [Bryant, 1992]. Other normal forms of propositional formulae that have found use in AI and could be applied to planning include the decomposable negation normal form [Darwiche, 2001] which is less restricted than binary decision diagrams (formulae in DNNF can be viewed as a superclass of BDDs) and are sometimes much smaller. However, smaller size means that some of the logical operations that can be performed in polynomial time for BDDs, like equivalence testing, are NP-hard for formulae in DNNF.

The main reason for using BDDs is that the logical equivalence of BDDs coincides with syntactic equivalence: two BDDs are logically equivalent if and only if they are the same BDD. Propositional formulae in general, or formulae in CNF or in DNF do not have this property. Furthermore, computing a BDD that represents the conjunction or disjunction of two BDDs or the negation of a BDDs also takes only polynomial time.

However, like with other normal forms, a BDD can be exponentially bigger than a corresponding unrestricted propositional formula. One example of such a propositional formulae is the binary multiplier: Any BDD representation of  $n$ -bit multipliers has a size exponential in  $n$ . Also, even though many of the basic operations on BDDs can be computed in polynomial time in the size of the component BDDs, iterating these operations may increase the size exponentially: some of these operator may double the size of the BDD, and doubling  $n$  times is exponential in  $n$  and in the size of the original BDD.

A main application of BDDs has been model-checking in computer-aided verification [Burch *et al.*, 1994; Clarke *et al.*, 1994], and in recent years these same techniques have been applied to AI planning as well. We will discuss BDD-based planning algorithms in Chapter 4.

BDDs are expressed in terms of the ternary Boolean operator if-then-else  $ite(p, \phi_1, \phi_2)$  defined as  $(p \wedge \phi_1) \vee (\neg p \wedge \phi_2)$ , where  $p$  is a proposition. Any Boolean formula can be represented by using this operator together with propositions and the constants  $\top$  and  $\perp$ . Figure 2.6 depicts a BDD for the formula  $(A \vee B) \wedge (B \vee C)$ . The normal arrow coming from a node for  $P$  corresponds to the case in which  $P$  is true, and the dotted arrow to the case in which  $P$  is false. Note that BDDs are graphs, not trees like formulae, and this provides a further reduction in the BDD size as a subformula never occurs more than once.

There is an ordering condition on BDDs: the occurrences of propositions on any path from the root to a leaf node must obey a fixed ordering of the propositions. This ordering condition together with the graph representation is required for the good computational properties of BDDs, like the polynomial time equivalence test.

A BDD corresponding to a propositional formula can be obtained by repeated application of an

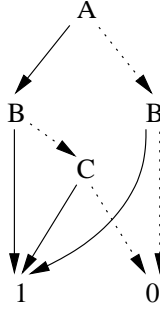


Figure 2.6: A BDD

equivalence called the Shannon expansion.

$$\phi \equiv (p \wedge \phi[\top/p]) \vee (\neg p \wedge \phi[\perp/p]) \equiv ite(p, \phi[\top/p], \phi[\perp/p])$$

**Example 2.3** We show how the BDD for  $(A \vee B) \wedge (B \vee C)$  is produced by repeated application of the Shannon expansion. We use the variable ordering  $A, B, C$  and use the Shannon expansion to eliminate the variables in this order.

$$\begin{aligned} & (A \vee B) \wedge (B \vee C) \\ \equiv & ite(A, (\top \vee B) \wedge (B \vee C), (\perp \vee B) \wedge (B \vee C)) \\ \equiv & ite(A, B \vee C, B) \\ \equiv & ite(A, ite(B, \top \vee C, \perp \vee C), ite(B, \top, \perp)) \\ \equiv & ite(A, ite(B, \top, C), ite(B, \top, \perp)) \\ \equiv & ite(A, ite(B, \top, ite(C, \top, \perp)), ite(B, \top, \perp)) \end{aligned}$$

The simplifications in the intermediate steps are by the equivalences  $\top \vee \phi \equiv \top$  and  $\perp \vee \phi \equiv \phi$  and  $\top \wedge \phi \equiv \phi$  and  $\perp \wedge \phi \equiv \perp$ . When

$$ite(A, ite(B, \top, ite(C, \top, \perp)), ite(B, \top, \perp))$$

is first turned into a tree and then equivalent subtrees are identified, we get the BDD in Figure 2.6. The terminal node 1 corresponds to  $\top$  and the terminal node 0 to  $\perp$ . ■

There are many operations on BDDs that are computable in polynomial time. These include forming the conjunction  $\wedge$  and the disjunction  $\vee$  of two BDDs, and forming the negation  $\neg$  of a BDD. However, conjunction and disjunction of  $n$  BDDs may have a size that is exponential in  $n$ , as adding a new disjunct or conjunct may double the size of the BDD.

An important operation in many applications of BDDs is the existential abstraction operation  $\exists p.\phi$ , which is defined by

$$\exists p.\phi = \phi[\top/p] \vee \phi[\perp/p]$$

where  $\phi[\psi/p]$  means replacing all occurrences of  $p$  in  $\phi$  by  $\psi$ . Also this is computable in polynomial time, but existentially abstracting  $n$  variables may result in a BDD that has size exponential in  $n$ , and hence may take exponential time. Existential abstraction can of course be used for any propositional formulae, not only for BDDs.

The formula  $\phi'$  obtained from  $\phi$  by existentially abstracting  $p$  is in general not equivalent to  $\phi$ , but has many properties that make the abstraction operation useful.

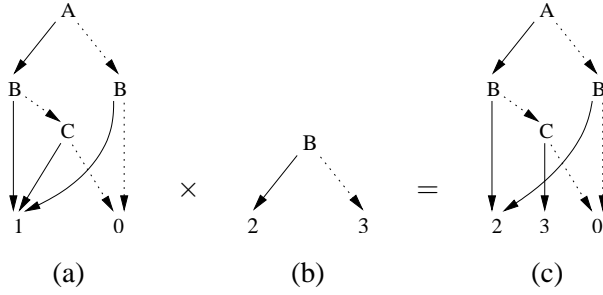


Figure 2.7: Three ADDs, the first of which is also a BDD.

**Lemma 2.4** Let  $\phi$  be a formula and  $p$  a proposition. Let  $\phi' = \exists p.\phi = \phi[\top/p] \vee \phi[\perp/p]$ . Now the following hold.

1.  $\phi$  is satisfiable if and only if  $\phi'$  is.
2.  $\phi$  is valid if and only if  $\phi'$  is.
3. If  $\chi$  is a formula without occurrences of  $p$ , then  $\phi \models \chi$  if and only if  $\phi' \models \chi$ .

**Example 2.5**

$$\begin{aligned} & \exists B.((A \rightarrow B) \wedge (B \rightarrow C)) \\ &= ((A \rightarrow \top) \wedge (\top \rightarrow C)) \vee ((A \rightarrow \perp) \wedge (\perp \rightarrow C)) \\ &\equiv C \vee \neg A \equiv A \rightarrow C \end{aligned}$$

$$\exists AB.(A \vee B) = \exists B.(\top \vee B) \vee (\perp \vee B) = ((\top \vee \top) \vee (\perp \vee \top)) \vee ((\top \vee \perp) \vee (\perp \vee \perp))$$

■

### 2.2.3 Algebraic decision diagrams

Algebraic decision diagrams (ADDs) [Fujita *et al.*, 1997; Bahar *et al.*, 1997] are a generalization of binary decision diagrams that has been applied to many kinds of probabilistic extensions of problems solved by BDDs. BDDs have only two terminal nodes, 1 and 0, and ADDs generalize this to a finite number of real numbers.

While BDDs represent Boolean functions, ADDs represent mapping from valuations to real numbers. The Boolean operations on BDDs, like taking the disjunction or conjunction of two BDDs, generalize to the arithmetic operations to take the arithmetic sum or the arithmetic product of two functions. There are further operations on ADDs that have no counterpart for BDDs, like constructing a function that on any valuation equals the maximum of two functions.

Figure 2.7 depicts three ADDs, the first of which is also a BDD. The product of ADDs is a generalization of conjunction of BDDs: if for some valuation/state ADD  $A$  assigns the value  $r_1$  and ADD  $B$  assigns the value  $r_2$ , then the product ADD  $A \cdot B$  assigns the value  $r_1 \cdot r_2$  to the valuation.

The following are some of the operations typically available in implementations of ADDs. Here we denote ADDs by  $f$  and  $g$  and view them as functions from valuations  $x$  to real numbers.

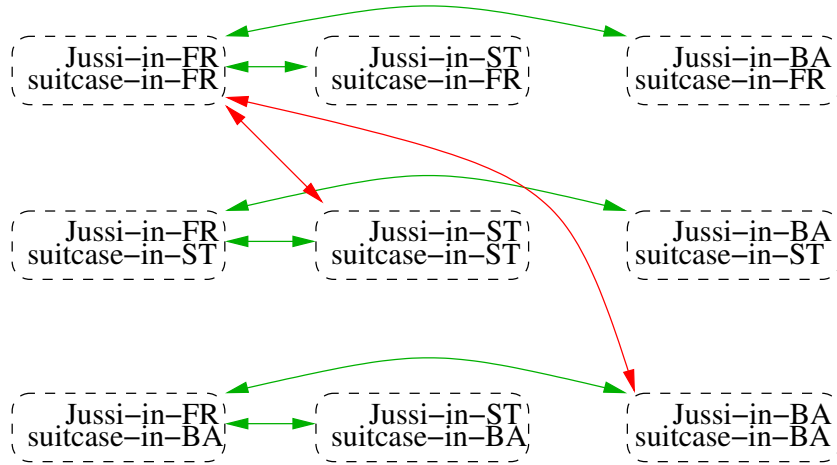


Figure 2.8: A simple transition system based on state variables

operation	notation	meaning
sum	$f + g$	$(f + g)(x) = f(x) + g(x)$
product	$f \cdot g$	$(f \cdot g)(x) = f(x) \cdot g(x)$
maximization	$\max(f, g)$	$(\max(f, g))(x) = \max(f(x), g(x))$

There is an operation for ADDs that corresponds to the existential abstraction operation on BDDs, and that is used in multiplication of matrices represented as ADDs, just like existential abstraction is used in multiplication of Boolean matrices represented as BDDs.

Let  $f$  be an ADD and  $p$  a proposition. Then *arithmetic existential abstraction* of  $f$ , written  $\exists p.f$ , is an ADD that satisfies the following.

$$(\exists p.f)(x) = (f[\top/p])(x) + (f[\perp/p])(x)$$

## 2.3 Operators and state variables

Transition systems are widely used in AI planning and other areas of computer science, and it is a model that can very well be used for describing all kinds of systems, especially man-made systems and abstractions of the real-world used by human beings.

However, describing a transition system by giving a set of states and then relations representing the actions is usually not the most natural nor the most concise description. This is because the individual states usually have a certain meaning, which determines which actions are possible in the state and what the possible successor states of the state under the given action are.

The common type of description of states is based on *state variables*. Let  $A$  be a finite set of state variables. Each state variable  $p \in A$  can have a finite number of different values  $R$ . Now a state  $s$  can be understood as a valuation  $s : A \rightarrow R$  that assigns a value to each state variable.

In this lecture we will restrict to Boolean state variables with  $R = \{0, 1\}$ , but almost everything in the lecture directly generalizes to any finite set  $R$  of values.

The state space  $S$  is now the set of all valuations of  $A$ .

**Example 2.6** Figure 2.8 illustrates a small transition system induced by state variables. We have

depicted each state by enumerating the state variables that have the value *true* in it (exactly two in each of the states in the figure), and left out states that do not correspond to the intuitive meaning of the states. Each state variable indicates whether one of the two objects is in one of the three locations (Freiburg, Strassburg, Basel.)

The two actions respectively correspond to traveling with and without the suitcase.

Clearly, if we were using many-valued state variables, it would suffice to have only two of them, each having three possible values corresponding to the three locations. ■

It remains to give a description of the set  $A$  of actions in terms of state variables. Intuitively, we have to say whether an action  $a \in A$  is applicable in a given state  $s$ , and what the successor state  $s'$  of that state under the given action is.<sup>1</sup> Actions are represented as *operators*  $\langle c, e \rangle$ , where  $c$  is a propositional formula over  $A$  that has to be satisfied by the valuation  $s$  for the action to be possible, and  $e$  describes how  $s'$  is obtained by changing the values of state variables in  $s$ .

Atomic effects in general are of the form  $p := r$  for  $p \in A$  and  $r \in R$ . In the Boolean case it is common to simply write  $p$  for  $p := 1$  and  $\neg p$  for  $p := 0$ , and also we will do so. Be careful to avoid confusion with an effect like  $e = p_1 \wedge \neg p_2$  and exactly the same looking formula  $\phi = p_1 \wedge \neg p_2$ . After the effect  $e$  the formula  $\phi$  will be true, but this is the only direct relationship between formulae and effects; in particular, there are no disjunctions  $\vee$  in effects and there is nothing in the propositional logic that corresponds to  $\triangleright$ .

**Definition 2.7** *Let  $A$  be a set of state variables. An operator is a pair  $\langle c, e \rangle$  where  $c$  is a propositional formula over  $A$  describing the precondition, and  $e$  is an effect over  $A$ . Effects are recursively defined as follows.*

1.  $\top$  is an effect (the dummy effect).
2.  $p$  and  $\neg p$  for state variables  $p \in A$  are effects.
3.  $e_1 \wedge \dots \wedge e_n$  is an effect if  $e_1, \dots, e_n$  are effects over  $A$  (the special case with  $n = 0$  is the empty conjunction  $\top$ .)
4.  $c \triangleright e$  is an effect if  $c$  is a formula over  $A$  and  $e$  is an effect over  $A$ .

Notice that the representation of transition systems in terms of state variables and operators opens the possibility that the size of the transition system, the number of states in the transition system, may be exponential in the size of the set of operators. This idea of *succinct representations* of various objects is present in very many areas of computer science. As we will see later in this lecture, succinctness usually means that algorithms for reasoning about the objects in question increases: if a computational problem, like finding shortest paths in transition systems represented as graphs, is solvable in polynomial time, solving the same problem for succinctly represented transition systems will be much higher.

**Definition 2.8 (Operator application)** *Let  $\langle c, e \rangle$  be an operator over  $A$ . Let  $s$  be a state, that is an assignment of truth values to  $A$ . The operator is applicable in  $s$  if  $s \models c$ .*

*Recursively assign each effect  $e$  a set  $[e]_s$  of literals  $p$  and  $\neg p$  for  $p \in A$  (the active effects.)*

1.  $[\top]_s = \emptyset$

---

<sup>1</sup>We discuss nondeterministic actions in Chapter 4.

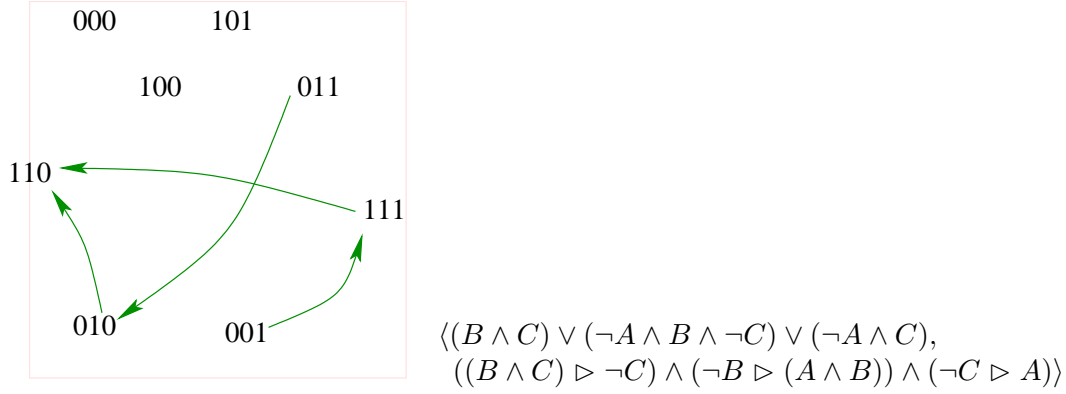


Figure 2.9: A transition graph with valuations of  $A$ ,  $B$  and  $C$  as states and, a corresponding operator

2.  $[p]_s = \{p\}$  for  $p \in A$ .
3.  $[\neg p]_s = \{\neg p\}$  for  $p \in A$ .
4.  $[e_1 \wedge \dots \wedge e_n]_s = [e_1]_s \cup \dots \cup [e_n]_s$ .
5.  $[c' \triangleright e]_s = [e]_s$  if  $s \models c'$  and  $[c' \triangleright e]_s = \emptyset$  otherwise.

The successor state of  $s$  under the operator is the one that is obtained from  $s$  by making the literals in  $[e]_s$  true and retaining the truth-values of state variables not occurring in  $[e]_s$ . This state is denoted by  $app_o(s)$ . We call the process of computing the successor state of a state with respect to an operator as progression.

**Example 2.9** Consider the operator  $\langle a, e \rangle$  where  $e = \neg a \wedge (\neg c \triangleright \neg b)$  and a state  $s$  with  $a$ ,  $b$  and  $c$  all true. The operator is applicable because  $s \models a$ . Now  $[e]_s = \{\neg a\}$  and  $app_{\langle a, e \rangle}(s) \models \neg a \wedge b \wedge c$ . ■

**Example 2.10** Figure 2.9 depicts a transition graph with valuations of three state variables  $A$ ,  $B$  and  $C$  as nodes, and a corresponding operator. ■

### 2.3.1 Extensions

The basic language for effects could be extended with further constructs. A natural construct would be *sequential composition* of effects. If  $e$  and  $e'$  are effects, then also  $e; e'$  is an effect that corresponds to first executing  $e$  and then  $e'$ . We do not discuss this topic further in this lecture. Definition 3.11 and Theorem 3.12 show how sequential composition can be eliminated from effects.

### 2.3.2 Normal forms

We introduce a normal form for effects that will be used in later sections for defining operations on propositional formulae describing sets of states.

Table 2.1 lists a number of equivalences on effects. Their proofs of correctness with Definition 2.8 are straightforward. An effect  $e$  is equivalent to  $\top \wedge e$ , and conjunctions of effects can be

$$c \triangleright (e_1 \wedge \cdots \wedge e_n) \equiv (c \triangleright e_1) \wedge \cdots \wedge (c \triangleright e_n) \quad (2.1)$$

$$c \triangleright (c' \triangleright e) \equiv (c \wedge c') \triangleright e \quad (2.2)$$

$$(c_1 \triangleright e) \wedge (c_2 \triangleright e) \equiv (c_1 \vee c_2) \triangleright e \quad (2.3)$$

$$e \wedge (c \triangleright e) \equiv e \quad (2.4)$$

$$e \equiv \top \triangleright e \quad (2.5)$$

$$e_1 \wedge (e_2 \wedge e_3) \equiv (e_1 \wedge e_2) \wedge e_3 \quad (2.6)$$

$$e_1 \wedge e_2 \equiv e_2 \wedge e_1 \quad (2.7)$$

$$c \triangleright \top \equiv \top \quad (2.8)$$

$$e \wedge \top \equiv e \quad (2.9)$$

$$(2.10)$$

Table 2.1: Equivalences on effects

arbitrarily reordered without affecting the meaning of the operator. These trivial equivalences will later be used without explicitly mentioning them, for example in the definitions of the normal forms and when applying equivalences.

The normal form corresponds to moving the conditionals inside so that their consequents are atomic effects, and it is useful for example in the computation of properties satisfied by predecessor states by regression in Section 3.2.2.

**Definition 2.11** *An effect  $e$  is in normal form if it is  $\top$  or a conjunction of one or more effects of the form  $c \triangleright p$  and  $c \triangleright \neg p$  where  $p$  is a state variable, and there is at most one occurrence of atomic effects  $p$  and  $\neg p$  for any state variable  $p$ . An operator  $\langle c, e \rangle$  is in normal form if  $e$  is in normal form.*

**Theorem 2.12** *For every operator there is an equivalent one in normal form. There is one that has a size that is polynomial in the size of the operator.*

*Proof:* We can transform any operator into normal form by using the equivalences 2.1, 2.2, 2.3, 2.6, 2.7, and 2.8 in Table 2.1.

The proof is by structural induction on the effect  $e$  of the operator  $\langle c, e \rangle$ .

Induction hypothesis: the effect  $e$  can be transformed to normal form.

Base case 1,  $e = \top$ : This is already in normal form.

Base case 2,  $e = p$  or  $e = \neg p$ : An equivalent effect in normal form is  $\top \triangleright e$  by Equivalence 2.5.

Inductive case 1,  $e = e_1 \wedge e_2$ : By the induction hypothesis  $e_1$  and  $e_2$  can be transformed into normal form, so assume that they already are. If one of  $e_1$  and  $e_2$  is  $\top$ , by Equivalence 2.9 we can eliminate it.

Assume  $e_1$  contains  $c_1 \triangleright l$  for some literal  $l$  and  $e_2$  contains  $c_2 \triangleright l$ . We can reorder  $e_1 \wedge e_2$  with Equivalences 2.6 and 2.7 so that one of the conjuncts is  $(c_1 \triangleright l) \wedge (c_2 \triangleright l)$ . Then by Equivalence 2.3 this conjunct can be replaced by  $(c_1 \vee c_2) \triangleright l$ . Because this can be done repeatedly for every literal  $l$ , we can transform  $e_1 \wedge e_2$  into normal form.

Inductive case 1,  $e = z \triangleright e_1$ : By the induction hypothesis  $e_1$  can be transformed to normal form, so assume that it already is.

If  $e_1$  is  $\top$ ,  $e$  can be replaced with the equivalent effect  $\top$ .

If  $e_1 = z' \triangleright e_2$  for some  $z'$  and  $e_2$ , then  $e$  can be replaced by the equivalent (by Equivalence 2.2) effect  $(z \wedge z') \triangleright e_2$  in normal form.

Otherwise,  $e_1$  is a conjunction of effects  $z \triangleright l$ . By Equivalence 2.1 we can move  $z$  inside the conjunction. Applications of Equivalences 2.2 transform the effect into normal form.

In this transformation the conditions  $c$  in  $c \triangleright e$  are copied into front of the atomic effects. Let  $m$  be the sum of the sizes of all the conditions  $c$ , and let  $n$  be the number of occurrences of atomic effects  $a$  and  $\neg a$  in the effect. An upper bound on size increase is  $\mathcal{O}(nm)$ , which is polynomial in the size of the original operator.  $\square$

A further reduction in the size of the descriptions of transition systems is obtained by using *schematic operators* instead of operators as described above.

There are often regularities in the set of operators and corresponding regularities in the transition system. A common regularity is that there are several almost identical *objects* that behave in the same way. For example, operators describing driving car 1 and car 2 between cities are otherwise identical except that in one case a reference to state variables about car 1 are used and in the other state variables about car 2. This kind of regularities are ubiquitous, and operators allowing easy expression of such sets of operators are used by almost all implementations of planning algorithms.

**Example 2.13** Consider the schematic operator

$$\langle \text{in}(x, t_1), \text{in}(x, t_2) \wedge \neg \text{in}(x, t_1) \rangle$$

where the schema variables  $x$ ,  $t_1$  and  $t_2$  take values as follows.

$$\begin{aligned} x &\in \{\text{car1}, \text{car2}\} \\ t_1 &\in \{\text{Freiburg}, \text{Strassburg}, \text{Basel}\} \\ t_2 &\in \{\text{Freiburg}, \text{Strassburg}, \text{Basel}\} \\ t_1 &\neq t_2 \end{aligned}$$

This schematic operator corresponds to the following set of operators.

$$\begin{aligned} \{ & \langle \text{in}(\text{car1}, \text{Freiburg}), \text{in}(\text{car1}, \text{Basel}) \wedge \neg \text{in}(\text{car1}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Freiburg}), \text{in}(\text{car1}, \text{Strassburg}) \wedge \neg \text{in}(\text{car1}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Strassburg}), \text{in}(\text{car1}, \text{Freiburg}) \wedge \neg \text{in}(\text{car1}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Strassburg}), \text{in}(\text{car1}, \text{Basel}) \wedge \neg \text{in}(\text{car1}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Basel}), \text{in}(\text{car1}, \text{Freiburg}) \wedge \neg \text{in}(\text{car1}, \text{Basel}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Basel}), \text{in}(\text{car1}, \text{Strassburg}) \wedge \neg \text{in}(\text{car1}, \text{Basel}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Freiburg}), \text{in}(\text{car2}, \text{Basel}) \wedge \neg \text{in}(\text{car2}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Freiburg}), \text{in}(\text{car2}, \text{Strassburg}) \wedge \neg \text{in}(\text{car2}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Strassburg}), \text{in}(\text{car2}, \text{Freiburg}) \wedge \neg \text{in}(\text{car2}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Strassburg}), \text{in}(\text{car2}, \text{Basel}) \wedge \neg \text{in}(\text{car2}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Basel}), \text{in}(\text{car2}, \text{Freiburg}) \wedge \neg \text{in}(\text{car2}, \text{Basel}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Basel}), \text{in}(\text{car2}, \text{Strassburg}) \wedge \neg \text{in}(\text{car2}, \text{Basel}) \rangle \} \end{aligned}$$

■

Schematic operators may also allow *existential* and *universal* quantification over sets of objects for encoding disjunctions and conjunctions more concisely. For example,  $\exists x \in \{A, B, C\} \text{in}(x, \text{Freiburg})$  is a short-hand for  $\text{in}(A, \text{Freiburg}) \vee \text{in}(B, \text{Freiburg}) \vee \text{in}(C, \text{Freiburg})$ .

Non-schematic operators are often called *ground operators*, and the process of producing a set of ground operators from a schematic operator is called *grounding*. In this lecture we will be using ground operators only. Most planning programs take schematic operators as input, and have a preprocessor that grounds them.

### 2.3.3 Sets of states as propositional formulae

Because we identified states with valuations of state variables, we can now identify sets of states with propositional formulae over the state variables. This allows us to perform set-theoretic operations on sets as logical operations, and test relations between sets by inference in the propositional logic.

operation on sets	operation on formulae
$A \cup B$	$A \vee B$
$A \cap B$	$A \wedge B$
$A \setminus B$	$A \wedge \neg B$
question about sets	question about formulae
$A \subseteq B?$	$\models A \rightarrow B?$
$A \subset B?$	$\models A \rightarrow B$ and not $\models B \rightarrow A?$
$A = B?$	$\models A \leftrightarrow B?$

Any inconsistent formula, like  $A \wedge \neg A$  or  $\perp$ , is not true in any state, and therefore represents the empty set. Similarly, any valid formula, for instance  $\top$  or  $A \vee \neg A$ , represents the set of all states (all valuations of the state variables.)

## 2.4 Computational complexity

In this section we discuss deterministic, nondeterministic and alternating Turing machines (DTMs, NDTMs and ATMs) and define several complexity classes in terms of them. For a detailed introduction to computational complexity see any of the standard textbooks [Balcázar *et al.*, 1988; 1990; Papadimitriou, 1994].

The definition of ATMs we use is like that of Balcázar *et al.* [1990] but without a separate input tape. Deterministic and nondeterministic Turing machines (DTMs, NDTMs) are a special case of a alternating Turing machines.

**Definition 2.14** An alternating Turing machine is a tuple  $\langle \Sigma, Q, \delta, q_0, g \rangle$  where

- $Q$  is a finite set of states (the internal states of the ATM),
- $\Sigma$  is a finite alphabet (the contents of tape cells),
- $\delta$  is a transition function  $\delta : Q \times \Sigma \cup \{ \sqcup, \square \} \rightarrow 2^{\Sigma \cup \{ \sqcup, \square \} \times Q \times \{L, N, R\}}$ ,
- $q_0$  is the initial state, and
- $g : Q \rightarrow \{ \forall, \exists, \text{accept}, \text{reject} \}$  is a labeling of the states.

The symbols  $|$  and  $\square$ , the end-of-tape symbol and the blank symbol, in the definition of  $\delta$  respectively refer to the beginning of the tape and to the end of the tape. It is required that  $s = |$  and  $m = R$  for all  $\langle s, q', m \rangle \in \delta(q, |)$  for any  $q \in Q$ , that is, at the left end of the tape the movement is always to the right and the end-of-tape symbol  $|$  may not be changed. For  $s \in \Sigma$  we restrict  $s'$  in  $\langle s', q', m \rangle \in \delta(q, s)$  to  $s' \in \Sigma$ , that is,  $|$  gets written onto the tape only in the special case when the R/W head is on the end-of-tape symbol. Notice that the transition function is a total function, and the ATM computation terminated upon reaching an accepting or a rejecting state.

A configuration of an ATM is  $\langle q, \sigma, \sigma' \rangle$  where  $q$  is the current state,  $\sigma$  is the tape contents left of the R/W head with the rightmost symbol under the R/W head, and  $\sigma'$  is the tape contents strictly right of the R/W head. This is a finite representation of the finite non-blank segment of the tape of the ATM.

The computation of an ATM starts from the initial configuration  $\langle q_0, |a, \sigma \rangle$  where  $a\sigma$  is the input string of the Turing machine. Below  $\epsilon$  denotes the empty string.

The configuration of an ATM changes as follows.

1. From  $\langle q, \sigma a, \sigma' \rangle$  to  $\langle q', \sigma, a' \sigma' \rangle$  when  $\delta(q, a) = \langle a', q', L \rangle$ .
2. From  $\langle q, \sigma a, \sigma' \rangle$  to  $\langle q', \sigma a', \sigma' \rangle$  when  $\delta(q, a) = \langle a', q', N \rangle$ .
3. From  $\langle q, \sigma a, b \sigma' \rangle$  to  $\langle q', \sigma a' b, \sigma' \rangle$  when  $\delta(q, a) = \langle a', q', R \rangle$ .
4. From  $\langle q, \sigma a, \epsilon \rangle$  to  $\langle q', \sigma a' \square, \epsilon \rangle$  when  $\delta(q, a) = \langle a', q', R \rangle$ .

A configuration  $\langle q, \sigma, \sigma' \rangle$  of an ATM is *final* if  $g(q) = \text{accept}$  or  $g(q) = \text{reject}$ .

The acceptance of an input string by an ATM is defined recursively starting from final configurations. A final configuration is accepting if  $g(q) = \text{accept}$ . Non-final configurations are accepting if the state is universal ( $\forall$ ) and all the immediate successor configurations are accepting, or if the state is existential ( $\exists$ ) and at least one of the immediate successor configurations is accepting. Finally, the ATM accepts a given input string if the initial configuration is accepting.

A nondeterministic Turing machine is an ATM without universal states. A deterministic Turing machine is an ATM with  $|\delta(q, s)| = 1$  for all  $q \in Q$  and  $s \in \Sigma$ .

The complexity classes used in this lecture are the following. PSPACE is the class of decision problems solvable by deterministic Turing machines that use a number of tape cells bounded by a polynomial on the input length  $n$ . Formally,

$$\text{PSPACE} = \bigcup_{k \geq 0} \text{DSPACE}(n^k).$$

Similarly other complexity classes are defined in terms of the time consumption (DTIME( $f(n)$ )) on a deterministic Turing machine, time consumption (NTIME( $f(n)$ )) on a nondeterministic Turing machine, or time or space consumption on alternating Turing machines (ATIME( $f(n)$ )) or

ASPACE( $f(n)$ ) [Balcázar *et al.*, 1988; 1990].

$$\begin{aligned}
 P &= \bigcup_{k \geq 0} \text{DTIME}(n^k) \\
 NP &= \bigcup_{k \geq 0} \text{NTIME}(n^k) \\
 EXP &= \bigcup_{k \geq 0} \text{DTIME}(2^{n^k}) \\
 NEXP &= \bigcup_{k \geq 0} \text{NTIME}(2^{n^k}) \\
 EXPSPACE &= \bigcup_{k \geq 0} \text{DSpace}(2^{n^k}) \\
 2\text{-EXP} &= \bigcup_{k \geq 0} \text{DTIME}(2^{2^{n^k}}) \\
 2\text{-NEXP} &= \bigcup_{k \geq 0} \text{NTIME}(2^{2^{n^k}}) \\
 \\ 
 ASPACE &= \bigcup_{k \geq 0} \text{ASPACE}(n^k) \\
 AEXPSPACE &= \bigcup_{k \geq 0} \text{ASPACE}(2^{n^k})
 \end{aligned}$$

There are many useful connections between complexity classes defined in terms of deterministic and alternating Turing machines [Chandra *et al.*, 1981], for example

$$\begin{aligned}
 EXP &= \text{ASPACE} \\
 2\text{-EXP} &= \text{AEXPSPACE}.
 \end{aligned}$$

Roughly, an exponential deterministic time bound corresponds to a polynomial alternating space bound.

We have defined all the complexity classes in terms of Turing machines. However, for all purposes of this lecture, we can equivalently use conventional programming languages (like C or Java) or simplified variants of them for describing computation. The main difference between conventional programming languages and Turing machines is that the former use random-access memory whereas memory access in Turing machines is local and only the current tape cell can be directly accessed. However, these two computational models can be simulated with each other with a polynomial overhead and are therefore for our purposes equivalent. The differences show up in complexity classes with very strict (subpolynomial) restrictions on time and space consumption.

Later in this lecture, in some of the proofs that a given computational problem belongs to a certain class we will usually give a program in a simple programming language comparable to a small subset of C or Java, instead of giving a formal description of a Turing machine, because the latter would usually be very complicated and difficult to understand.

A problem  $L$  is *C-hard* (where  $C$  is any of the complexity classes) if all problems in the class  $C$  are polynomial time *many-one reducible* to it; that is, for all problems  $L' \in C$  there is a function  $f_{L'}$  that can be computed in polynomial time on the size of its input and  $f_{L'}(x) \in L$  if and only if  $x \in L'$ . We say that the function  $f_{L'}$  is a translation from  $L'$  to  $L$ . A problem is *C-complete* if it belongs to the class  $C$  and is  $C$ -hard.

In complexity theory the most important distinction between computational problems is that between *tractable* and *intractable* problems. A problem is considered to be tractable, efficiently solvable, if it can be solved in polynomial time. Otherwise it is intractable. Most planning problems are highly intractable, but for many algorithmic approaches to planning it is important that certain basic steps in these algorithms can be guaranteed to be tractable.

In this lecture we analyze the complexity of many computational problems, showing them to be complete problems for some of the classes mentioned above. The proofs consist of two parts. We show that the problem belongs to the class. This is typically by giving an algorithm for the

problem, possibly a nondeterministic one, and then showing that the algorithm obeys the resource bounds on time or memory consumption as required by the complexity class. Then we show the hardness of the problem for the class, that is, we can reduce any problem in the class to the problem in polynomial time. This can be either by simulating all Turing machines that represent computation in the class, or by reducing a complete problem in the class to the problem in question in polynomial time (a many-one reduction).

For almost all commonly used complexity classes there are more or less natural complete problems that often have a central role in proving the completeness of other problems for the class in question. Some complete problems for the complexity classes mentioned above are the following.<sup>2</sup>

class	complete problem
P	truth-value of formulae in the propositional logic in a given valuation
NP	satisfiability of formulae in the propositional logic (SAT)
PSPACE	truth-value of quantified Boolean formulae

Complete problems for classes like EXP and NEXP can be obtained from the P-complete and NP-problems by representing propositional formulae succinctly in terms of other propositional formulae [Papadimitriou and Yannakakis, 1986]. We will not discuss this topic further in this lecture.

## 2.5 Exercises

**2.1** Show that for any transition system  $\langle S, \{a_1, \dots, a_n\} \rangle$  in which the states  $s \in S$  are valuations of a set  $P$  of propositional variables (as in Example 2.10), the actions  $a_1, \dots, a_n$  can be represented in terms of operators.

Show that conditional effects with  $\triangleright$  are necessary, that is, find a transition system that cannot be represented with operators that do not contain conditional effects with  $\triangleright$ . *Hint:* There is an example with two states and one state variable.

---

<sup>2</sup>For definition of P-hard problems we have to use more restricted many-one reductions that use only logarithmic space instead of polynomial time. Otherwise all non-trivial problems in P would be P-hard and P-complete.