Chapter 2

Background

In this chapter we will define the formal machinery which is needed for describing different planning problems and algorithms. We will give the basic definitions related to the classical propositional logic and the transition system model which is the basis of most work on planning and which is closely related to finite automata and transition systems in other areas of computer science.

2.1 Transition systems

We define transition systems in which states are atomic objects and actions are represented as binary relations on the set of states.

Definition 2.1 *A* transition system *is a 5-tuple* $\Pi = \langle S, I, O, G, P \rangle$ *where*

1. S *is a finite set of states,*
2. $I \subseteq S$ *is the set of initial states,*
3. O *is a finite set of actions* $o \subseteq S \times S$*,*
4. $G \subseteq S$ *is the set of goal states, and*
5. $P = (C_1, \ldots, C_n)$ *is a partition of* S *to non-empty classes of observationally indistinguishable states satisfying* $\bigcup\{C_1, \ldots, C_n\} = S$ *and* $C_i \cap C_j = \emptyset$ *for all* i, j *such that* $1 \leq i < j \leq n$*.*

Making an observation tells which set C_i the current state belongs to. Distinguishing states within a given C_i is not possible by observations. If two states are observationally distinguishable then plan execution can proceed differently for them.

The number n of components in the partition P determines different classes of planning problems with respect to observability restrictions. If $n = |S|$ then every state is observationally distinguishable from every other state. This is called *full observability*. If $n = 1$ then no observations are possible and the transition system is *unobservable*. The general case $n \in \{1, \ldots, |S|\}$ is called *partial observability*.

An action o is *applicable* in states for which it associates at least one successor state. We define *images* of states as $img_o(s) = \{s' \in S | sos'\}$ and (weak) *preimages* of states as $preimg_o(s') = \{s \in S | sos'\}$. Generalization to sets of states is $img_o(T) = \bigcup_{s \in T} img_o(s)$ and $preimg_o(T) =$

$\bigcup_{s \in T} preimg_o(s)$. For sequences $o_1, \ldots, o_n$ of actions $img_{o_1;\ldots;o_n}(T) = img_{o_n}(\cdots img_{o_1}(T) \cdots)$ and $preimg_{o_1;\ldots;o_n}(T) = preimg_{o_1}(\cdots preimg_{o_n}(T) \cdots)$. The *strong preimage* of a set T of states is the set of states for which all successor states are in T, defined as $spreimg_o(T) = \{s \in S | s' \in T, sos', img_o(s) \subseteq T\}$.

Lemma 2.2 *Images, strong preimages and weak preimages of sets of states are related to each other as follows. Let o be any action and S and S' any sets of states.*

1. $spreimg_o(T) \subseteq preimg_o(T)$
2. $img_o(spreimg_o(T)) \subseteq T$
3. *If* $T \subseteq T'$ *then* $img_o(T) \subseteq img_o(T')$.
4. $preimg_o(T \cup T') = preimg_o(T) \cup preimg_o(T')$.
5. $s' \in img_o(s)$ *if and only if* $s \in preimg_o(s)$.

Proof:

1. $spreimg_o(T) = \{s \in S | s' \in T, sos', img_o(s) \subseteq T\} \subseteq \{s \in S | s' \in T, sos'\} = \bigcup_{s' \in T} \{s \in S | sos'\} = \bigcup_{s' \in T} preimg_o(s') = preimg_o(T)$.
2. Take any $s' \in img_o(spreimg_o(T))$. Hence there is $s \in spreimg_o(T)$ so that sos'. As $s \in spreimg_o(T)$, $img_o(s) \subseteq T$. Since $s' \in img_o(s)$, $s' \in T$.
3. Assume $T \subseteq T'$ and $s' \in img_o(T)$. Hence sos' for some $s \in T$ by definition of images. Hence sos' for some $s \in T'$ because $T \subseteq T'$. Hence $s' \in img_o(T')$ by definition of images.
4. $preimg_o(T \cup T') = \bigcup_{s' \in T \cup T'} \{s \in S | sos'\} = \bigcup_{s' \in T} \{s \in S | sos'\} \cup \bigcup_{s' \in T'} \{s \in S | sos'\} = preimg_o(T) \cup preimg_o(T')$
5. $s' \in img_o(s)$ iff sos' iff $s \in preimg_o(s)$.

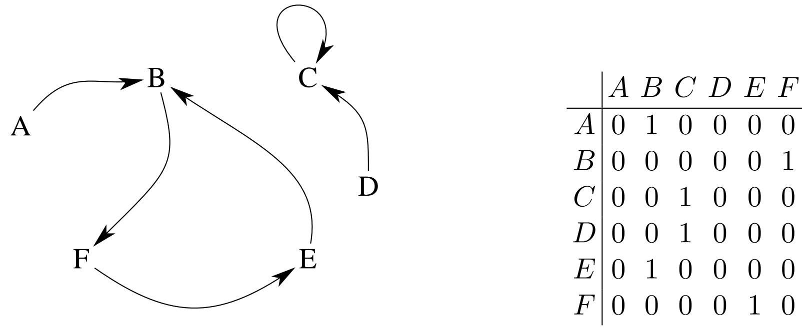
□

2.1.1 Deterministic transition systems

Transition systems which we use in Chapter 3 have only one initial state and deterministic actions. For this subclass observability is irrelevant because the state of the transition system after a given sequence of actions can be predicted exactly. We use a simpler formalization of them.

Definition 2.3 *A* deterministic transition system *is a 4-tuple* $\Pi = \langle S, I, O, G \rangle$ *where*

1. S *is a finite set of states,*
2. $I \in S$ *is the initial state,*
3. O *is a finite set of actions* $o \subseteq S \times S$ *that are partial functions, and*
4. $G \subseteq S$ *is the set of goal states.*



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

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

That the actions are partial functions means that for any $s \in S$ and $o \in O$ there is at most one state s' such that sos'. We denote the unique successor state s' of a state s in which operator o is applicable by $s' = app_o(s)$. For sequences $o_1; \ldots; o_n$ of operators we define $app_{o_1;\ldots;o_n}(s)$ as $app_{o_n}(\cdots app_{o_1}(s)\cdots)$.

2.1.2 Incidence matrices

Actions and other binary relations can be represented in terms of incidence matrices M (adjacency matrices) in which the element in row i and column j indicates whether a transition from state i to j is possible.

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.

For matrices $M_1, \ldots, M_n$ which represent the transition relations of actions $a_1, \ldots, a_n$ the combined transition relation is $M = M_1 + M_2 + \cdots + M_n$. 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 $b' = 1$. Boolean addition is used because in the presence of nondeterminism we could have 1 for both of two transitions from A to B and from A to C. For probabilistic planning problems normal addition is used and matrix elements are interpreted as probabilities of nondeterministic transitions.

The incidence matrix corresponding to first taking action a_1 and then a_2 is $M_1 M_2$. 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

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

Here n is the number of states and $I_{n \times n}$ is the unit matrix of size n. By Tarski's fixpoint theorem $R_i = R_j$ for some $i \geq 0$ and all $j \geq i$ because of the monotonicity property that every element that is 1 for some i is 1 also for all $j > i$. Matrix $R_i = M^0 \cup M^1 \cup \cdots \cup M^i$ represents reachability

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

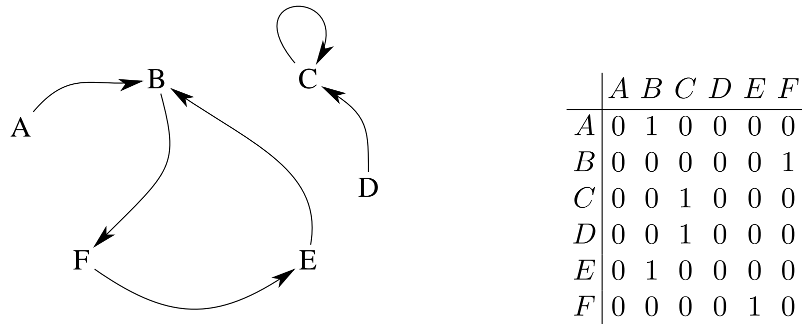
$\times$

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	1	0	0	0	0	0
D	0	0	0	1	0	0
E	0	0	0	0	1	0
F	0	0	0	1	0	0

$=$

	A	B	C	D	E	F
A	0	0	0	0	0	1
B	0	0	0	**1**	0	0
C	1	0	0	0	0	0
D	1	0	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	1	0

Figure 2.2: Matrix product corresponds to sequential composition.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

Figure 2.3: A transition graph and the corresponding matrix M

by i actions or less.

2.2 Classical propositional logic

Let A be a set of propositional variables (atomic propositions). We define the set of propositional formulae inductively as follows.

1. For all $a \in A$, a is a propositional formula.

2. If ϕ is a propositional formula, then so is $\neg\phi$.

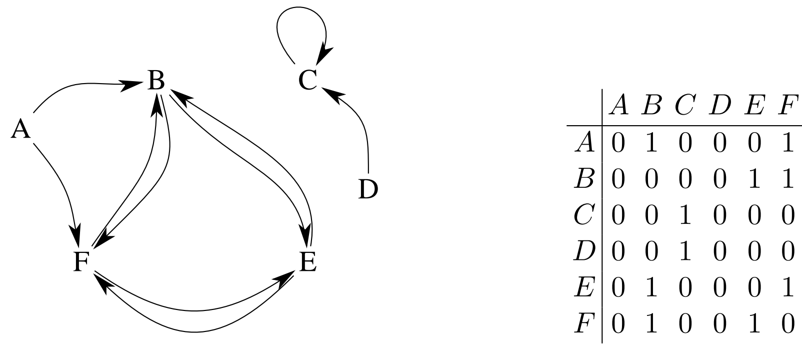
3. If ϕ and ϕ' are propositional formulae, then so is $\phi \vee \phi'$.

4. If ϕ and ϕ' 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.

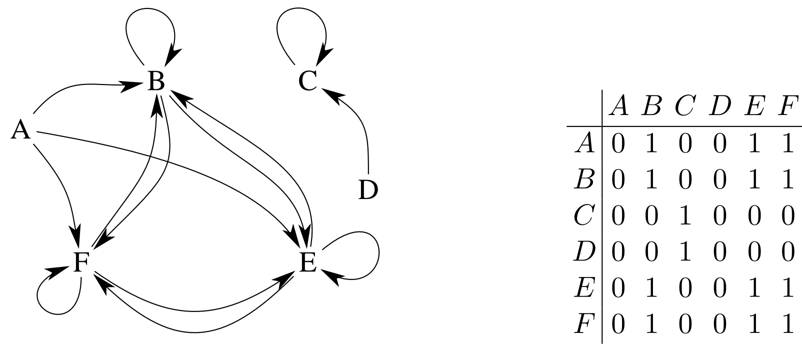
The symbols $\wedge$, $\vee$ and $\neg$ are *connectives* respectively denoting the *conjunction*, *disjunction* and *negation*. We define the implication $\phi \rightarrow \phi'$ as an abbreviation for $\neg\phi \vee \phi'$, and the equivalence $\phi \leftrightarrow \phi'$ as an abbreviation for $(\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$.

A valuation of A is a function $v : A \rightarrow \{0, 1\}$ where 0 denotes false and 1 denotes true. Valuations are also known as *assignments* or *models*. For propositional variables $a \in A$ we define



	A	B	C	D	E	F
A	0	1	0	0	0	1
B	0	0	0	0	1	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	1
F	0	1	0	0	1	0

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



	A	B	C	D	E	F
A	0	1	0	0	1	1
B	0	1	0	0	1	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	1	1
F	0	1	0	0	1	1

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

$v \models a$ if and only if $v(a) = 1$. A valuation of the propositional variables in A can be extended to a valuation of all propositional formulae over A 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. $v \not\models \bot$

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 ϕ is *satisfiable* (*consistent*) if there is at least one valuation v so that $v \models \phi$. Otherwise it is *unsatisfiable* (*inconsistent*). A finite set F of formulae is satisfiable if $\bigwedge_{\phi \in F} \phi$ is. A propositional formula ϕ is *valid* or a *tautology* if $v \models \phi$ for all valuations v. We denote this by $\models \phi$. A propositional formula ϕ is *a logical consequence* of a propositional formula ϕ', written $\phi' \models \phi$, if $v \models \phi$ for all valuations v such that $v \models \phi'$. A propositional formula that

is a proposition variable a or a negated propositional variable $\neg a$ for some $a \in A$ is *a literal*. A formula that is a disjunction of literals is *a clause*.

A formula ϕ is in *negation normal form* (NNF) if all occurrences of negations are directly in front of propositional variables. Any formula can be transformed to negation normal form by applications of the De Morgan rules $\neg(\phi \vee \phi') \equiv \neg\phi \wedge \neg\phi'$ and $\neg(\phi \wedge \phi') \equiv \neg\phi \vee \neg\phi'$, the double negation rule $\neg\neg\phi \equiv \phi$. A formula ϕ is in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions of literals. A formula ϕ is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals. Any formula in CNF or in DNF is also in NNF.

2.2.1 Quantified Boolean formulae

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

6. If ϕ is a formula and $a \in A$, then $\forall a\phi$ is a formula.

7. If ϕ is a formula and $a \in A$, then $\exists a\phi$ is a formula.

Further, there is the requirement that every variable is quantified, that is, every occurrence of $a \in A$ in a QBF is in the scope of either $\exists a$ or $\forall a$.

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

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 $\bot$. The truth-value of these formulae is independent of the valuation, and is recursively computed by the Boolean functions associated with the connectives $\vee$, $\wedge$ and $\neg$.

Definition 2.4 (Truth of QBF) *A formula $\exists x\phi$ is true if and only if $\phi[\top/x] \vee \phi[\bot/x]$ is true. (Equivalently, if $\phi[\top/x]$ is true or $\phi[\bot/x]$ is true.)*

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

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

Example 2.5 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. ∎

Note that a QBF with only existential quantifiers is true if and only if the formula without 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 to 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 many computational problems that presumably cannot be translated into the satisfiability problem of the propositional logic in polynomial time (assuming that NP$\neq$PSPACE) can be efficiently translated into 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.

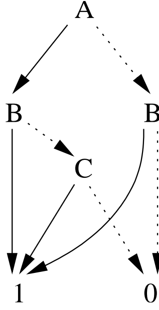


Figure 2.6: A BDD

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 $\bot$. 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 equivalence called the Shannon expansion.

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

Example 2.6 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.

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

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

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

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 $\bot$. ■

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

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

where $\phi[\psi/p]$ means replacing all occurrences of p in ϕ by ψ. Also this is computable in polynomial time, and in contrast to repeated conjunction and disjunction, repeated existential abstraction of several variables remains a polynomial time operation. Existential abstraction can of course be used for any propositional formulae, not only for BDDs.

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

Lemma 2.7 *Let ϕ be a formula and p a proposition. Let $\phi' = \exists p.\phi = \phi[\top/p] \vee \phi[\bot/p]$. Now the following hold.*

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

Example 2.8

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

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

■

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.

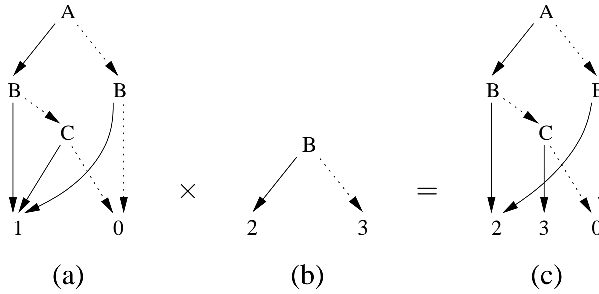


Figure 2.7: Three ADDs, the first of which is also a BDD.

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.

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

2.3 Succinct transition systems

It is often more natural to represent the states of a transition system as valuations of state variables instead of enumeratively as in Section 2.1. The binary relations that correspond to actions can often be represented compactly in terms of the changes the actions cause to the values of state variables.

We represent states in terms of a set A of Boolean state variables which take the values *true* or *false*. Each *state* is a valuation of A (a function $s : A \to \{0, 1\}$.)

Since we identify 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 as summarized in Table 2.1

The actions of a succinct transition system are described by operators. An operator has two components. The precondition describes the set of states in which the action can be taken. The effect describes the successor states of each state in terms of the changes made to the values of the state variables.

Definition 2.9 *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 (the* precondition*), and e is an* effect *over A. Effects over A are recursively defined as follows.*

set	formula
$T \cup U$	$T \vee U$
$T \cap U$	$T \wedge U$
$\overline{T}$	$\neg T$
$T \backslash U$	$T \wedge \neg U$
$\emptyset$	$\bot$
the universal set	$\top$

question about sets	question about formulae
$T \subseteq U$?	$\models T \rightarrow U$?
$T \subset U$?	$\models T \rightarrow U$ and $\not\models U \rightarrow T$?
$T = U$?	$\models T \leftrightarrow U$?

Table 2.1: Correspondence between set-theoretical and logical operations

1. *a and $\neg a$ for state variables $a \in A$ are effects over A.*
2. *$e_1 \wedge \cdots \wedge e_n$ is an effect over A if $e_1, \ldots, e_n$ are effects over A (the special case with $n = 0$ is the empty effect $\top$).*
3. *$c \rhd e$ is an effect over A if c is a formula over A and e is an effect over A.*
4. *$e_1 | \cdots | e_n$ is an effect over A if $e_1, \ldots, e_n$ for $n \geq 2$ are effects over A.*

The compound effects $e_1 \wedge \cdots \wedge e_n$ denote executing all the effects $e_1, \ldots, e_n$ simultaneously. In conditional effects $c \rhd e$ the effect e is executed if c is true in the current state. The effects $e_1 | \cdots | e_n$ denote nondeterministic choice between the effects $e_1, \ldots, e_n$. Exactly one of these effects is chosen randomly.

Operators describe a binary relation on the set of states as follows.

Definition 2.10 (Operator application) *Let $\langle c, e \rangle$ be an operator over A. Let s be a state (a valuation of A). The operator is* applicable *in s if $s \models c$ and every set $E \in [e]_s$ is consistent. The set $[e]_s$ is recursively defined as follows.*

1. *$[a]_s = \{\{a\}\}$ and $[\neg a]_s = \{\{\neg a\}\}$ for $a \in A$.*
2. *$[e_1 \wedge \cdots \wedge e_n]_s = \{\bigcup_{i=1}^{n} E_i | E_1 \in [e_1]_s, \ldots, E_n \in [e_n]_s\}$.*
3. *$[c' \rhd e]_s = [e]_s$ if $s \models c'$ and $[c' \rhd e]_s = \{\emptyset\}$ otherwise.*
4. *$[e_1 | \cdots | e_n]_s = [e_1]_s \cup \cdots \cup [e_n]_s$.*

An operator $\langle c, e \rangle$ induces a binary relation $R\langle c, e \rangle$ on states as follows: states s and s' are related by $R\langle c, e \rangle$ if $s \models c$ and s' is obtained from s by making the literals in some $E \in [e]_s$ true and retaining the values of state variables not occurring in E.

We define images and preimages for operators o in terms of $R(o)$, for instance by $\mathit{preimg}_o(s) = \mathit{preimg}_{R(o)}(s)$.

Definition 2.11 *A* succinct transition system *is a 5-tuple $\Pi = \langle A, I, O, G, V \rangle$ where*

1. *A is a finite set of state variables,*
2. *I is a formula over A describing the initial states,*
3. *O is a finite set of operators over A,*
4. *G is a formula over A describing the goal states, and*
5. *$V \subseteq A$ is the set of observable state variables.*

Succinct transition systems with $V = A$ are *fully observable*, and succinct transition systems with $V = \emptyset$ are *unobservable*. Without restrictions on V the succinct transition systems are *partially observable*.

We can associate a transition system with every succinct transition system.

Definition 2.12 *Given a succinct transition system $\Pi = \langle A, I, O, G, V \rangle$, define the transition system $F(\Pi) = \langle S, I', O', G', P \rangle$ where*

1. *S is the set of all Boolean valuations of A,*
2. *$I' = \{s \in S | s \models I\}$,*
3. *$O' = \{R(o) | o \in O\}$,*
4. *$G' = \{s \in S | s \models G\}$, and*
5. *$P = (C_1, \ldots, C_n)$ where $v_1, \ldots, v_n$ for $n = 2^{|V|}$ are all the Boolean valuations of V and $C_i = \{s \in S | s(a) = v_i(a)$ for all $a \in V\}$ for all $i \in \{1, \ldots, n\}$.*

The transition system may have a size that is exponential in the size of the succinct transition system. However, the construction takes only polynomial time in the size of the transition system.

2.3.1 Deterministic succinct transition systems

A deterministic operator has no occurrences of $|$ in the effect. Further, in this special case the definition of operator application is slightly simpler.

Definition 2.13 (Operator application) *Let $\langle c, e \rangle$ be a deterministic operator over A. Let s be a state (a valuation of A). The operator is* applicable *in s if $s \models c$ and the set $[e]_s^{det}$ is consistent. The set $[e]_s^{det}$ is recursively defined as follows.*

1. *$[a]_s^{det} = \{a\}$ and $[\neg a]_s^{det} = \{\neg a\}$ for $a \in A$.*
2. *$[e_1 \wedge \cdots \wedge e_n]_s^{det} = \bigcup_{i=1}^{n} [e_i]_s^{det}$.*
3. *$[c' \rhd e]_s^{det} = [e]_s^{det}$ if $s \models c'$ and $[c' \rhd e]_s^{det} = \emptyset$ otherwise.*

A deterministic operator $\langle c, e \rangle$ induces a partial function $R\langle c, e \rangle$ on states as follows: two states s and s' are related by $R\langle c, e \rangle$ if $s \models c$ and s' is obtained from s by making the literals in $[e]_s^{det}$ true and retaining the truth-values of state variables not occurring in $[e]_s^{det}$.

We define $app_o(s) = s'$ by $sR(o)s'$ and $app_{o_1;\ldots;o_n}(s) = s'$ by $app_{o_n}(\ldots app_{o_1}(s)\ldots)$, just like for non-succinct transition systems.

We formally define deterministic succinct transition systems.

Definition 2.14 *A* deterministic succinct transition system *is a 4-tuple* $\Pi = \langle A, I, O, G\rangle$ *where*

1. *A is a finite set of state variables,*
2. *I is an initial state,*
3. *O is a finite set of operators over A, and*
4. *G is a formula over A describing the goal states.*

We can associate a deterministic transition system with every deterministic succinct transition system.

Definition 2.15 *Given a deterministic succinct transition system $\Pi = \langle A, I, O, G\rangle$, define the deterministic transition system $F(\Pi) = \langle S, I, O', G'\rangle$ where*

1. *S is the set of all Boolean valuations of A,*
2. *$O' = \{R(o)|o \in O\}$, and*
3. *$G' = \{s \in S|s \models G\}$.*

A subclass of operators considered in many early and recent works restrict to *STRIPS* operators. An operator $\langle c, e\rangle$ is a STRIPS operator if c is a conjunction of state variables and e is a conjunction of literals. STRIPS operators do not allow disjunctivity in formulae nor conditional effects. This class of operators is sufficient in the sense that any transition system can be expressed in terms of STRIPS operators only if the identities of operators are not important: when expressing a transition system in terms of STRIPS operators only some operators correspond to an exponential number of STRIPS operators.

Example 2.16 Let $A = \{a_1, \ldots, a_n\}$ be the set of state variables. Let $o = \langle \top, e\rangle$ where

$$e = (a_1 \rhd \neg a_1) \wedge (\neg a_1 \rhd a_1) \wedge \cdots \wedge (a_n \rhd \neg a_n) \wedge (\neg a_n \rhd a_n))\rangle.$$

This operator reverses the values of all state variables. As its set of active effects $[e]_s^{det}$ is different in every one of 2^n states, this operator corresponds to 2^n STRIPS operators.

$$\begin{aligned}
o_0 &= \langle \neg a_1 \wedge \neg a_2 \wedge \cdots \wedge \neg a_n, a_1 \wedge a_2 \wedge \cdots \wedge a_n\rangle \\
o_1 &= \langle a_1 \wedge \neg a_2 \wedge \cdots \wedge \neg a_n, \neg a_1 \wedge a_2 \wedge \cdots \wedge a_n\rangle \\
o_2 &= \langle \neg a_1 \wedge a_2 \wedge \cdots \wedge \neg a_n, a_1 \wedge \neg a_2 \wedge \cdots \wedge a_n\rangle \\
o_3 &= \langle a_1 \wedge a_2 \wedge \cdots \wedge \neg a_n, \neg a_1 \wedge \neg a_2 \wedge \cdots \wedge a_n\rangle \\
&\vdots \\
o_{2^n-1} &= \langle a_1 \wedge a_2 \wedge \cdots \wedge a_n, \neg a_1 \wedge \neg a_2 \cdots \wedge \neg a_n\rangle
\end{aligned}$$

■

$$c \rhd (e_1 \wedge \cdots \wedge e_n) \equiv (c \rhd e_1) \wedge \cdots \wedge (c \rhd e_n) \tag{2.1}$$
$$c_1 \rhd (c_2 \rhd e) \equiv (c_1 \wedge c_2) \rhd e \tag{2.2}$$
$$(c_1 \rhd e) \wedge (c_2 \rhd e) \equiv (c_1 \vee c_2) \rhd e \tag{2.3}$$
$$e \wedge (c \rhd e) \equiv e \tag{2.4}$$
$$e \equiv \top \rhd e \tag{2.5}$$
$$e_1 \wedge (e_2 \wedge e_3) \equiv (e_1 \wedge e_2) \wedge e_3 \tag{2.6}$$
$$e_1 \wedge e_2 \equiv e_2 \wedge e_1 \tag{2.7}$$
$$c \rhd \top \equiv \top \tag{2.8}$$
$$e \wedge \top \equiv e \tag{2.9}$$

Table 2.2: Equivalences on effects

2.3.2 Extensions

The basic language for effects could be extended with further constructs. A natural construct is *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'. Definition 3.11 and Theorem 3.12 show how effects with sequential composition can be reduced to effects without sequential composition.

2.3.3 Normal form for deterministic operators

Deterministic operators can be transformed to a particularly simple form without nesting of conditionality $\rhd$ and with only atomic effects e as antecedents of conditionals $\phi \rhd e$. Normal forms are useful as they allow concentrating on a particularly simple form of effects.

Table 2.2 lists a number of equivalences on effects. Their proofs of correctness with Definition 2.13 are straightforward. An effect e is equivalent to $\top \wedge e$, and conjunctions of effects can be 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 all occurrences of $\rhd$ inside $\wedge$ so that the consequents of $\rhd$ are atomic effects.

Definition 2.17 *A deterministic effect e is in* normal form *if it is $\top$ or a conjunction of one or more effects $c \rhd a$ and $c \rhd \neg a$ with at most one occurrence of atomic effect a and $\neg a$ for any $a \in A$. An operator $\langle c, e\rangle$ is in normal form if e is in normal form.*

Theorem 2.18 *For every deterministic 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 deterministic operator into normal form by using the equivalences in Table 2.2. 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, $e = a$ or $e = \neg a$: An equivalent effect in normal form is $\top \rhd e$ 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 $c_1 \rhd l$ for some literal l and e_2 contains $c_2 \rhd l$. We can reorder $e_1 \wedge e_2$ with Equivalences 2.6 and 2.7 so that one of the conjuncts is $(c_1 \rhd l) \wedge (c_2 \rhd l)$. Then by Equivalence 2.3 it can be replaced by $(c_1 \vee c_2) \rhd l$. Since this can be done repeatedly for every literal l, we can transform $e_1 \wedge e_2$ into normal form.

Inductive case 2, $e = z \rhd e_1$: 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 by $\top$ which is in normal form.

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

Otherwise, e_1 is a conjunction of effects $z \rhd l$. 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 $c \rhd e$ 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 of the new effect is $\mathcal{O}(nm)$ which is polynomial in the size of the original effect. □

2.3.4 Normal forms for nondeterministic operators

We can generalize the normal form defined in Section 2.3.3 to nondeterministic effects and operators. In the normal form nondeterministic choices and conjunctions are the outermost constructs, and consequents e of conditional effects $c \rhd e$ are atomic effects.

Definition 2.19 (Normal form for nondeterministic operators) *A deterministic effect is in normal form if it is $\top$ or a conjunction of one or more effects $c \rhd a$ and $c \rhd \neg a$ with at most one occurrence of a and $\neg a$ for any $a \in A$.*

A nondeterministic effect is in normal form if it is $e_1|\cdots|e_n$ or $e_1 \wedge \cdots \wedge e_n$ for effects e_i that are in normal form.

A nondeterministic operator $\langle c, e\rangle$ is in normal form if e is in normal form.

For showing that every nondeterministic effect can be transformed into normal form we use further equivalences that are given in Table 2.3.

Theorem 2.20 *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 former.*

Proof: Transformation to normal form is like in the proof of Theorem 2.18. Additional equivalences needed for nondeterministic choices are 2.10 and 2.11. □

Example 2.21 The effect

$$a \rhd (b|(c \wedge f)) \wedge ((d \wedge e)|(b \rhd e))$$

$$c \rhd (e_1|\cdots|e_n) \equiv (c \rhd e_1)|\cdots|(c \rhd e_n) \quad (2.10)$$
$$e \wedge (e_1|\cdots|e_n) \equiv (e \wedge e_1)|\cdots|(e \wedge e_n) \quad (2.11)$$
$$(e'_1|\cdots|e'_{n'})|e_2|\cdots|e_n \equiv e'_1|\cdots|e'_{n'}|e_2|\cdots|e_n \quad (2.12)$$
$$(e' \wedge (c \rhd e_1))|e_2|\cdots|e_n \equiv (c \rhd ((e' \wedge e_1)|e_2|\cdots|e_n)) \wedge (\neg c \rhd (e'|e_2|\cdots|e_n)) \quad (2.13)$$

Table 2.3: Equivalences on nondeterministic effects

in normal form is

$$((a \rhd b)|((a \rhd c) \wedge (a \rhd f))) \wedge (((\top \rhd d) \wedge (\top \rhd e))|(b \rhd e)).$$

■

For some applications a still simpler form of operators is useful. In the second normal form for nondeterministic operators nondeterminism may appear only at the outermost structure in the effect.

Definition 2.22 (Normal form II for nondeterministic operators) *A deterministic effect is in normal formal II if it is $\top$ or a conjunction of one or more effects $c \rhd a$ and $c \rhd \neg a$ with at most one occurrence of a and $\neg a$ for any $a \in A$.*

A nondeterministic effect is in normal form II if it is of form $e_1|\cdots|e_n$ where e_i are deterministic effects in normal form II.

A nondeterministic operator $\langle c, e\rangle$ is in normal form II if e is in normal form II.

Theorem 2.23 *For every operator there is an equivalent one in normal form II.*

Proof: By Theorem 2.20 there is an equivalent operator in normal form. The transformation further into normal form II requires equivalences 2.11 and 2.12. □

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 alternating Turing machines.

Definition 2.24 *An* alternating Turing machine *is a tuple $\langle \Sigma, Q, \delta, q_0, g\rangle$ where*

- *Σ is a finite alphabet (the contents of tape cells),*
- *Q is a finite set of states (the internal states of the ATM),*

- δ *is a transition function* $\delta : Q \times (\Sigma \cup \{|, \Box\}) \rightarrow 2^{(\Sigma \cup \{|\}) \times Q \times \{L,N,R\}}$,
- q_0 *is the initial state, and*
- $g : Q \rightarrow \{\forall, \exists, \mathit{accept}, \mathit{reject}\}$ *is a labeling of the states.*

The symbols $|$ and $\Box$, the end-of-tape symbol and the blank symbol, in the definition of δ 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. Note 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, σ is the tape contents left of the R/W head with the rightmost symbol under the R/W head, and σ' 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 configuration is universal ($\forall$) if $g(q) = \forall$, and existential ($\exists$) if $g(q) = \exists$.

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 ϵ denotes the empty string.

Successor configurations are defined as follows.

1. A successor of $\langle q, \sigma a, \sigma'\rangle$ is $\langle q', \sigma, a'\sigma'\rangle$ if $\langle a', q', L\rangle \in \delta(q, a)$.
2. A successor of $\langle q, \sigma a, \sigma'\rangle$ is $\langle q', \sigma a', \sigma'\rangle$ if $\langle a', q', N\rangle \in \delta(q, a)$.
3. A successor of $\langle q, \sigma a, b\sigma'\rangle$ is $\langle q', \sigma a' b, \sigma'\rangle$ if $\langle a', q', R\rangle \in \delta(q, a)$.
4. A successor of $\langle q, \sigma a, \epsilon\rangle$ is $\langle q', \sigma a' \Box, \epsilon\rangle$ if $\langle a', q', R\rangle \in \delta(q, a)$.

We write $\langle q, \sigma\rangle \vdash \langle q', \sigma'\rangle$ if the latter is a successor configuration of the former. A configuration $\langle q, \sigma, \sigma'\rangle$ of an ATM is *final* if $g(q) =$ accept or $g(q) =$ reject.

The acceptance of an input string by an ATM is defined recursively starting from final configurations. A final configuration is 0-accepting if $g(q) =$ accept. A non-final universal configuration is n-accepting for $n \geq 1$ if its every successor configuration is m-accepting for some $m < n$ and one of its successor configurations is $n-1$-accepting. A non-final existential configuration is n-accepting for $n \geq 1$ if at least one of its successor configurations is $n-1$-accepting and it has no m-accepting successor configurations for any $m < n-1$. Finally, an ATM accepts a given input string if its initial configuration is n-accepting for some $n \geq 0$. A configuration is *accepting* if it is n-accepting for some $n \geq 0$.

If an ATM accepts a given input string, then we can define *an accepting computation subtree* of the ATM and the input string as a set T of accepting configurations such that

1. the initial configuration is in T,
2. if $c \in T$ is a $\forall$-configuration then $c' \in T$ for all configurations c' such that $c \vdash c'$,
3. if $c \in T$ is an n-accepting $\exists$-configuration then $c' \in T$ for at least one c' such that $c \vdash c'$ and c' is m-accepting for some $m < n$.

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,

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

Other complexity classes are similarly defined in terms of the time consumption on a deterministic Turing machine (DTIME($f(n)$)), time consumption on a nondeterministic Turing machine (NTIME($f(n)$)), or time or space consumption on alternating Turing machines (ATIME($f(n)$) or ASPACE($f(n)$)) [Balcázar *et al.*, 1988; 1990].

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

$$\begin{aligned}
\text{APSPACE} &= \textstyle\bigcup_{k \geq 0} \text{ASPACE}(n^k) \\
\text{AEXPSPACE} &= \textstyle\bigcup_{k \geq 0} \text{ASPACE}(2^{n^k})
\end{aligned}$$

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

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

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, the proofs of membership of a given computational problem in a certain complexity class are usually given in terms of 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 $x \in L'$ for all inputs x. 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.[1]

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].

2.5 Exercises

2.1 Show that any transition system in which the states are valuations of a set A of propositional variables can be translated into an equivalent succinct transition system.

2.2 Show that conditional effects with $\rhd$ are necessary, that is, find a transition system where states are valuations of a set of state variables and the actions cannot be represented as operators without conditional effects with $\rhd$. *Hint*: There is an example with two states and one state variable.

[1] 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.